Formalising Meaning

a History of Programming Language Semantics

Troy Kaighin Astarte

A thesis submitted for the degree of

Doctor of Philosophy

June 2019

School of Computing

Newcastle University

Newcastle upon Tyne, UK

Difficult things are difficult to describe.

Tony Hoare (in Walk 1969a, p. 9)

For Jay, who always taught me to love seeking knowledge and to question everything.

And for Cliff, who opened the door to doing this for a living.

Acknowledgements

Thanks firstly to my supervisor, Cliff Jones, for providing inspiration, motivation, and friendship; and to my extended supervisory team: Martin Campbell-Kelly, Brian Randell, and John Tucker, who provided guidance and advice. I am also indebted to a number of other helpful people: Mark Priestly, Elizabeth Scott, and Donald MacKenzie. Special thanks to Gerard Alberts, who taught me the joy of historical investigation and gave me the opportunity to pass this on.

I am grateful to everyone who shared their stories with me, providing great source material: David Beech, Rod Burstall, Hermann Maurer, Doug McIlroy, Peter Mosses, Erich Neuhold, Gordon Plotkin, Brian Randell, Joe Stoy, and Kurt Walk. I appreciate the Bodleian Library in Oxford and the Zemanek Nachlass at the Technische Universität Wien for archiving useful documents and allowing me access. Thanks as well to Gerard Alberts and Jeremy Gibbons for granting me access to the papers in their care.

I acknowledge the communities at whose events I have spoken, and whose attendees provided useful comments on my material as well as much-needed validation: the Commission for the History and Philosophy of Computing, the British Society for the History of Mathematics, PROGRAMme and the European Society for the History of Science. The Strachey 100 conference, held in Oxford in November 2016, deserves special mention: many people gave interesting and valuable accounts which informed this work.

I am deeply grateful to Joanne Allison for proofreading. Thanks also to Jay Astarte and Peter Carmichael for providing comments on some parts. And to Andrius Velykis, without whose LaTeX wizardry this document would look rather plain.

Finally, thanks to EPSRC for providing the funding for this research, and Newcastle University's School of Computing, for supporting me throughout.

Abstract

The emergence of high-level programming languages in the 1950s brought a series of challenges to the burgeoning computing community. Many of these centred around the difficulty of determining precisely the meaning of programming languages. This was a key issue for both those writing translators for languages and those writing programs in these languages. People in the computing and mathematics worlds attempted to address these challenges by creating a variety of ways to describe the semantics of high-level languages (the issue of defining syntax was more quickly resolved).

The history of the development of formal semantic description techniques is explored, with a particular focus on two centres of research: the IBM Laboratory in Vienna, and the Programming Research Group in Oxford. The stories of these two threads of development create interesting contrasts and similarities and discussing them—as well as the previous works that motivated them and the subsequent work inspired by them—can tell us much about programming languages and formal computing today, as well as the culture and context of computing in the 1960s and 1970s.

This work takes both a historical and technical approach. Looking at historical developments, considering the people involved in research, their motivations, the situations in which they worked, and how they interacted informs understanding of their technical outputs. Exploring the technical side, however, also helps uncover additional historical angles, such as interactions and influences both positive and negative.

Contents

CHAPTER 1

Introduction

Programs, collections of instructions whose purpose is to control a computer to effect some kind of change, are a central part of computing. Since the late 1940s, many different systems have been developed to provide a more efficient way to use a programmer's time than issuing commands directly in the machine's order code. These led to the use of 'high-level languages': those whose meaning was sufficiently abstract that programmers could write at a level closely attuned to their application. There is a huge proliferation of such languages: Wikipedia includes over 700 on its 'List of Programming Languages' page;[1] Landin (1966b) and Sammet (1972, p. 602) were already addressing this alarming diversity in the 1960s and 1970s. While some languages are more obviously linked to certain problem domains, determining which languages are any good is an open question and an area of ongoing research (see, for example, Ray et al. 2017). Getting to grips with programming languages is clearly vital to the creation of successful programs and always has been. One tool for assisting in this is formalism.

Formalism—the use of careful, rigorous, precise mathematical systems and notations—is an important part of theoretical computing, and has been central since electronic calculating machines came to be seen as more than mere number crunchers. Almost as soon as people began considering the theory of computing, the formal description of programming languages became a key focus. This work was influenced by ideas from logic, mathematics, and linguistics, and involved a number of people important in the wider computing scene. The relative ease of formalising syntax, the form of

[1] `http://en.wikipedia.org/wiki/List_of_programming_languages`

languages, was soon overtaken by the much tougher task of precisely defining semantics, or meaning. The complexity and depth of many programming languages resulted in increasingly intricate description techniques and ultimately meant that such efforts became confined to specialists.

Formal semantics is a very important part of the history of both formal aspects of computing, and programming languages. Work done in this area had noticeable effects on the development of early programming languages and led into many other aspects of formal computing, such as techniques for managing concurrency, and the specification and verification of programs. Many well-known historical figures added to the literature on semantics, as well as plenty of lesser-known researchers whose contributions should not be overlooked.

Studying the work of this period can tell us a lot about how early computing practitioners regarded programming languages; their motivations, the way they interacted, and their ultimate outputs make for an interesting historical examination. This dissertation considers the development of techniques for the formal description of programming language semantics from the late 1950s until the mid 1980s, with a strong focus on two key centres of activity: the IBM Laboratory Vienna, and the Oxford University Programming Research Group.

1.1 Why be interested in formal semantics?

As a way to motivate the current work, it is worth exploring why people were interested in formal semantics, and what that meant. Trying to make definitions about historical concepts is always problematic,[2] but it is important at least to delimit an area of interest. The present work focuses on people who wrote descriptions of programming languages that they saw as formal, as long as some attempt was made to describe the *meaning* of the language. These writers had different beliefs of exactly what 'semantics' meant, and these will be explored in depth across the present work. One early definition was given by Dijkstra (1961b):[3]

> As the aim of a programming language is to describe processes, I regard the definition of its semantics as the design, the description of a machine that has as reaction to an arbitrary process description in this

[2] Mahoney (1996b) particularly cautions against redefining concepts from the past in modern terminology.

[3] Dijkstra would later eschew such a view; see Section 5.6.

> language the actual execution of this process. One could also give the semantic definition of the language by stating all the rules according to which one could execute a process, given its description in the language. Fundamentally, there is nothing against this, provided that nothing is left to my imagination as regards the way and the order in which these rules are to be applied. But if nothing is left to my imagination I would rather use the metaphor of the machine that by its very structure defines the semantics of the language. In the design of a language this concept of the "defining machine" should help us to ensure the unambiguity of semantic interpretation of texts.

This idea of using a machine to define a language is at the core of 'operational' approaches to semantics. The other main approach considered in this work is the 'denotational' method, a basic definition of which is given by Plotkin (2018): "it's just the ascription of suitable abstract entities to suitable syntactical entities".

Formalised syntax is mentioned to a limited extent in the present work, because its use is often strongly linked to the semantics. Some approaches (such as van Wijngaarden's two level grammars, discussed in Section 7.1) combine syntactic and semantic description in a way that is difficult to disentangle. However, it is important to be clear that there is a large volume of work on syntax and parsing problems which is not covered here.

It is worth beginning by exploring why the area of formal semantics is interesting. Studying the motivations of historical actors is a central part of the current work. The question "why write the formal semantics of programming languages?" is addressed historically throughout, but a brief summary here sets out the motivation for considering this area.

A good initial question is why the word 'semantics' was used to refer to the meaning of languages. McCarthy claimed (in Gorn 1964, p. 136) that he introduced the term in 1961; but this seems unlikely given that Backus (1959) referred to the "syntax and semantics" of ALGOL 58 two years earlier—and at that point, McCarthy was not involved in the ALGOL project (according to Perlis 1981, p. 78). One reason could be that the use of 'language' as a metaphor for describing the encoding of programs for computers, as discussed by Nofre, Priestley, and Alberts (2014), brings in ideas of linguistics. Morris (1946) had split language understanding into 'syntax', 'semantics', and 'pragmatics', and so it was natural to apply these concepts to artificial computer languages as well. This is a view espoused by Lucas (1978, p. 3), who wrote "By viewing computers as language interpreting machines it becomes quite

apparent that the analysis of programming (and human) languages is bound to be a central theme of Computer Science".

The first computers were controlled with instructions that directly referenced machine functions and so the meaning of these notations had an obvious correspondence with physical actions. However, trying to write programs in these codes was time-consuming and the resulting texts did not offer an intuitive understanding to humans. Most programming systems were therefore a balance between precision and intuitiveness, which brought many advantages. Vaughan Pratt asked an important two-part question at the *Mathematical Foundations of Programming Languages Semantics* panel (Jones et al. 2004):

1. How much money have programming languages saved?
2. Is there a Nobel Prize in Economics for answering Question 1?

However, increasied abstraction in programming languages brought a loss of immediate correlation with machine behaviour, which clouded the meaning of programs written in such languages. Additionally, a supplementary program was then required to translate from higher-level programs into those which could be executed on machines; such programs became called 'compilers', because some early examples compiled together subroutines from a library (Hopper 1981, p. 10). Obviously, the correct behaviour of compilers was of tremendous importance, and getting this right was a serious problem in the 1960s, especially due to a lack of high quality programmers, as explained by Feeney (1981, p. 265; quoted in Peláez Valdez 1988, p. 117):[4]

> The vast increase in programming requirements brought an influx of incompetent and poorly trained programmers. A lot of good people joined too! But it was the poor ones who soaked up the machine resource. It was bad enough to have a slow, inefficient and bug-ridden daily production of software, but these characters really came into their own when

[4] Another issue was poor management and a failure to understand the difficulty of the tasks they were facing, as McIlroy (2018) explained:

> There was an outfit that could turn out FORTRAN compilers overnight, called Digitech. They had an excellent technology. I asked them if they might turn this on PL/I. They saw this as getting in on the ground floor, because if they did it, it would be the first PL/I compiler.
>
> The way they went at it is they hired a new person and just put that poor guy alone on the job. He had never written a compiler before! So he wasn't at home with the Digitech type of technology. When he finally delivered something, it was so bad, so naive, that Bob Morris looked at it with me and Bob said "We gotta do it ourselves."

> they wrote slow, inefficient and bug-ridden compilers which produced deformed object code, which confused the hapless programmer endlessly. How could he resolve whether the problem lay with his lack of skill, or the operating system, or the compiler, or even with the intermittent hardware fault?

The problem was then how to specify the meaning of a programming language such that the correctness of the compiler could be checked against this specification. If a language was defined in terms of a machine language, that would be machine-specific, and one important goal of high-level languages was to bring portability across different machines. However, writing a language definition in natural language was problematic for a number of reasons (not least of which being the choice of language given the international nature of many computing communities), as Zemanek (1972a, p. 118) explained:

> Description in natural language is never precise, complete or free from contradiction. Its terms call up connotations and associations out of control of the writer. And when describing a really big structure, it is impossible to keep all features in mind at one time as is necessary when writing about structures in natural language.

This was spotted by Herb Simon, as reported by Mahoney (1988, p. 117), who wrote "The thinking of computer designers and programmers is embodied in the way their machines and programs *work*,[5] and the languages they use to specify how things are to work are themselves artifacts. The models they use are filled with images[6] difficult or distractingly tedious to translate into words." Russell (1931) had noted similar concerns in mathematics, writing "Ordinary language is totally unsuited for expressing what physics really asserts, since the words of everyday life are not sufficiently abstract. Only mathematics and mathematical logic can say as little as the physicist means to say".

This resulted in the desire for a description technique which was abstract and mathematical in nature. Milne and Strachey (1974, pp. 10–1) wrote:

> The use of 'high-level' programming languages encourages programmers to think in terms of the abstract objects being manipulated rather than

[5] Original emphasis.

[6] Here the author is referring not just to pictorial images, but also formal languages, diagrams, and other kinds of specialised notation.

> the operations which computers perform on bit patterns. This means that we should be able to describe a program in terms of the abstract objects it uses; this is the genesis of a need for a theory of programming language semantics.

This was echoed by Marcotty, Ledgard, and Bochmann (1976, p. 274):

> It is precisely in the context-sensitive and semantic areas that formalism is needed. There is generally little argument over the precise syntax of a statement even if there is no formal description of it. All too often, however, an intuitive understanding of the semantics turns out to be woefully superficial. It is only when an attempt at implementation (which is, after all, a kind of formal definition) is made that ramifications and discrepancies are laid bare. What was thought to have been fully understood is discovered to have been differently perceived by various readers of the same description. By then, it is frequently too late to change, and incompatibilities have been cast in actual code.

Perlis (in Gorn 1964, p. 135) argued that "We are interested in semantics so that we can mechanise the process of translation on computers, and this is really the only reason we are interested in semantics in programming and computation". However, this was not the view of everyone in the field.

Aside from determining compiler correctness, the other major reason that researchers gave for wanting to formalise the semantics of programming languages was to increase the mathematical rigour of computing. Mahoney (2002) wrote that the use of programming languages was central to understanding the powers and abilities of computers, and the importance of understanding these languages led to the necessity of formal tools for discussing them—in a separate paper, he explicitly mentioned denotational semantics as part of "the mathematics that makes the [computer] theoretically possible" (Mahoney 1988, p. 118). Milner (1993) had similar beliefs, saying "this study of formal language for controlling this prosthetic limb which is the computer is just completely fundamental". Consequently, Mahoney (2002, pp. 28–9) placed semantics as one of the three main pillars of theoretical computing in the period from 1955 to 1975. Providing firm mathematical foundations for computation was seen as leading to better confidence in programs, as well as providing a justification for the existence of computer *science*.

Hoare (1985, quoted in Mahoney 2002, p. 36), argued that computing was in fact inherently mathematical: machines were mathematical objects, programs were math-

ematical expressions, programming languages were mathematical theories, and the act of programming was itself a mathematical activity. This viewpoint grew from the early view of computing machines as tools for assisting in mathematics, and led naturally to the idea of applying mathematical principles to computing. Another angle to the intersection of mathematics and computation was provided by Plotkin (2018), who simply stated that formalising semantics is fun—it provides nice mathematical problems with obvious practical connections.

Another hope for formal semantics was that it would see use in the design of programming languages; Wilkes (in Gorn 1964), for example, thought of semantics as a tool for helping improve the study of languages. Backus, speaking at the same conference, agreed, and added that formal descriptions could aid in the communication of languages:

> One purpose that one could have in describing meaningful mechanisms, for describing the meaning of programs in arbitrary new languages, is so that people can publish a description of a newly proposed language and have it made clear to the readers in a fairly transparent way what interpretation he wishes to place on the statements of this language.

Zemanek (1972a, p. 119) remarked that this should also allow areas of incomplete description:

> An essential feature of formal definition technology is the economy of expression. While always speaking precisely, it must leave it to the user how much to go into details. In other words, one must be able to speak precisely even on fuzzy objects or precise objects, the details of which are to be defined later or at another level of design; one must be able to give an exact sketch of a task to be carried out, of the essentials to be met, leaving open the free details without loss of clarity or precision.

A final and significant reason for writing formal semantics was that it created a system that enabled reasoning about a language and programs written in that language. This was a goal of McCarthy (1963, p. 6) who wrote that he wanted to be able to bring statements about programs within the realm of mathematical proof. For Strachey, the relative ease of reasoning about functions was a large motivator for developing a heavily mathematical approach to semantics based on functions. Milner, Morris, and Newey (1975) achieved some practical success in this regard,

creating a theorem prover called LCF from Scott's domain theory, originally conceived as part of the denotational semantics of programming languages.

It would be a mistake to assume that all users of formal semantics had the same requirements. Pagan (1981) noted that different approaches to semantics would likely address different needs, something Gorn (1961a, p. 336) had also realised two decades earlier:

> Different audiences, different conditions, and different purposes will find different methods of specification efficient or called for. A compiler constructor's manual, an operator's manual, or manuals for different types of users must differ. Specification of ALGOL for use by the constructor of a translator cannot conveniently be the same as the specification for use by an algorithm designer, or for use by an algorithm user.

This was echoed by Milne (2000, p. 78), who wrote "Having just one description per language would be unsatisfactory".

The core desires for formal semantics were summarised by Stoy (2016b): "we want to be confident that our computer programs do what we designed them to do". This need for confidence sat at all layers of abstraction from the high-level language down to the hardware in "a heavily nested set of confidence-inspiring procedures", a key part of which was semantics of programming languages (Stoy 2016b).

1.2 Background and literature

The history of computing is a fairly new field, emerging as an area of study during the late 20th century. As a discipline, it has its roots in two main areas: the history of science, technology, and mathematics; and reflection from computer scientists. Engaging with the history of computing takes tools and techniques from various disciplines, and while some writers have written historiographical pieces explaining how history of computing might be done, one generally learns from reading other histories. This section provides an overview of some related works in the history of computing, as well as looking at some pieces that reflect on the discipline.

Campbell-Kelly (2007) wrote about 'The history of the history of software', exploring how the field has changed over the years. He explained that, following similar trends in the history of other topics in science and technology, early histories of software focused on technical aspects, before looking at industry elements, applications

of computing, and finally institutional, social, and political issues. Campbell-Kelly presented this change as a 'maturing' of the discipline, while also clearly acknowledging the importance of all these parts in their own right. Still, he made some rather disparaging remarks about technical studies, including characterising his own early work on the programming of early machines (Campbell-Kelly 1980a,b, 1981) as "low hanging fruit".

A prolific historian of computing, Mike Mahoney wrote a number of works on the practice of the history of computing. In an early piece, Mahoney (1988) argued that history of computing must borrow questions from the history of technology, and ask whether computing moved through a small number of breakthroughs, or rather many successive small improvements. He argued that it is vital to look at collaborative structures and to stop "substituting biography for careful analysis of social progress", something unfortunately common in the history of science. One particular example he highlighted is the importance of studying industrial research labs, to look at the impact of commercial pressures, considering constraints and requirements as well as 'driving forces'. Mahoney noted that technical artefacts with a physical presence such as hardware are hard to read for historians, but software is even harder because of the absence of the dynamic aspects inherent in running programs, and the difficulty of inferring programmer and user processes from the static remnants of software such as source code. Scientific writing might seem like an easier source, but Mahoney explained that just as in software creation, looking only at finished published pieces does not give as much insight into the creation of computer science.

For the *History of Programming Languages II* conference (see below), Mahoney was on the advisory committee, and wrote a pair of short papers giving advice to authors (Mahoney 1996a; Mahoney 1996b). Here he explained how to historicise a narrative of events, citing a classic paper by Hamming (1980), whose title serves as a basic instruction to historians: 'We would know what they thought when they did it'. This means discussing the development of ideas, and trying to determine what the real questions and concerns of the time were. Mahoney (1996a, p. 25) argued that it is difficult and usually misguided to try to determine motivations of actors through the outcomes of their work, writing "That is, ideas grow roughly as trees do; Fred [Brooks] talked about the decision tree. It is important to get back to the root but not identify the root with the branch. Because the branch may actually have emerged in quite different ways than that assumption, that single backtrack, might reveal". Another pitfall to avoid, according to Mahoney, was looking back

at historical work and conflating it with current notions, by saying "Person X was really doing Y".[7]

Aside from these works by Campbell-Kelly and Mahoney, historiography of computing is fairly rare. The only other major exemplary work is that of Tom Haigh; in his (2004) piece 'The history of computing: An introduction for the computer scientist', he explained that the lack of methodological writing is something of a deliberate decision. In history, the major focus is always "developing a historical narrative" rather than putting methodology, data sources, and hypotheses up front. The paper also explained how history of computing has been done, can be done, and ought be done, highlighting particular uses in education and research. Haigh added that people who have become professional historians of computing have come from different backgrounds, such as history of science or social history, and bring commensurately different approaches. He hoped that historians of all kinds—as well as computer scientists—would become interested in history of computing and write varying perspectives.

A later paper from Haigh (2015) addressed a plea by Knuth (2014) to keep technical aspects central in histories of computing. Haigh explained that Knuth was concerned by works, especially by Campbell-Kelly (2003, 2007), that showed a trend away from technical considerations. Haigh clarified that historical works such as Campbell-Kelly's are part of the history of computing which is broader than the history of computer science that Knuth desired, but argued that there is room in the history of computing for all kinds of historical narratives.[8] Examining the technical angles of historical work, as the present work does, requires a strong background in computing as well as historical experience. This combination of skills, argued Haigh, is quite rare, and explains why there is something of a division between different kinds of historian, as well as between historians and computer scientists. Broadly speaking, this division (sometimes termed 'internalist' vs. 'externalist') is between those who look at technical aspects of historical work, and those who focus on the context in which the work was performed and its impact on the context. Haigh argued that another reason making technical histories quite uncommon is the difficulty of getting funding and academic plaudits.

[7] A good example of this kind of re-interpretation is Reynolds' 'The discoveries of continuations' (1993).

[8] This is a sentiment echoed by the historian of mathematics Grattan-Guinness (2005), who wrote that understanding *history* (what happened and why) is just as important as understanding *heritage* (impact and influence) when considering maths of the past.

Further hints on historiography for computing can be found in the literature on the history of mathematics. This is a related field, as early computers were largely considered to be mathematical tools used by mathematicians and many of the historical actors in the present account were mathematicians or had a background in mathematics. While formal description of programming languages often took inspiration and tools from mathematics and logic, it tends not to receive treatment from historians of those fields. That said, as Mahoney (2011) pointed out, knowing something of the history of maths and logic helps in understanding the history of fields they led to, such as computing. so taking direction from the history of maths can be helpful. For example, Wardhaugh (2010) explained how to engage with historical mathematical documents: which to choose, what questions to ask, how to use sources to determine context, background, and intended use, and so on.

Stedall (2012) wrote a *Very Short Introduction* to the history of mathematics, which contains much useful information and guidance—as well as plenty which is less relevant to history of computing because of mathematics' much longer past (historians of computing rarely have to worry about archaeological methods). Warnings about translating or interpreting historic ideas in today's notions still apply to history of computing, however, and may be harder to see due to the shorter distance. Stedall detailed some pitfalls for the historian to avoid: the 'ivory tower' and 'lone genius' fallacies, and Whig histories that treat developments as a series of inevitable stepping stones to success. She reminded readers of the importance of looking at the broader web of people involved in mathematics and how they interact and contribute indirectly to the development of maths. A key question highlighted is 'why study mathematics?'—it is vital to understand the motivations of historical actors. Finally, she warned against trying to identify firsts: not only are there many criteria for determining primacy, even deciding what really counts as a discovery is difficult. Stedall (2012, p. 111) ends her book with a list of questions:

> The questions asked by historians over the last 50 years have both changed and diversified. It is no longer enough simply to ask who discovered what and when. We also want to know what mathematical practices engaged groups of people or individuals, and why. What historical or geographical influences were at work? How were mathematical activities perceived, by the participants or by others? What aspects were particularly valued? What steps were taken to preserve or hand on mathematical expertise? Who was paying for it? How did an individual mathematician manage his (or her) time and skill? What were their

> motivations? What did they produce? What did they do with it? And with whom did they discuss, collaborate, or argue along the way?

These questions serve to inform the formation of goals for the present work.

Moving on from historiography to look at historical pieces similar to the current work, one notices that there are relatively few histories of formal and theoretical aspects of computing (for the reasons outlined above). Haigh (2004, p. 19) notes that software history tends to focus on software engineering and attempts of historical actors to (borrowing a term from Mahoney) form an agenda for their work. Small parts of these stories cover the attempts to use formal methods to codify an engineering discipline; while the number of such works is still relatively small, that the area of formal methods has received explicit focus indicates that it is identified as an important area.

Examples of works that look at formal computing include Jones (2003b) and MacKenzie (2001). Jones wrote an article-length piece on the history of program verification, tracing developments on assertions, axioms, program development, and correctness proof from Turing through Hoare to Aczel. This is a great resource discussing an area of computing closely related to formal semantics, although the paper takes a more 'history of ideas' approach than looking at context.[9] MacKenzie's (2001) *Mechanizing Proof* is a socio-historical book on the use of computers for verification and proof of mathematical or computer problems. His approach is much more cultural and contextual, with a heavy use of oral history interviews alongside an examination of published work. MacKenzie's background in sociology leads him to take a constructivist approach to history, with a nuanced building of narrative and exploration of various avenues of research; this particular book was a strong inspiration for the present work. A great book written by Priestley (2011) studies the impact of logic on computing in quite some detail. It has a good balance of historical and technical material and Priestley showed the value of both these information styles supporting each other. There is even a section (pp. 230–7) in which he discussed the formal description of programming languages, but it does not have the length to go into detail.

Amongst his many papers on the history of computing, Mahoney (2011) wrote an overview of theoretical computer science. He showed the field's roots in mathematics and logic, and explained why the gap between the experience of the programmer and

[9] For a very different take on these events, see Petricek (2017) in which a Lakatosian discussion of software correctness is presented. The historical rigour appears in the footnotes, which soberly provide the appropriate citations and attributions for the students' lively and irreverent discussion.

the reality of the programs suggested that a strong theory of computation was required. The paper covers switching theory, Boolean logic, computability, automata, and formal languages. It ends by covering the formal semantics of programming languages, explaining that understanding semantics is an essential aspect to really understanding computation. Mahoney (2011, p. 142) wrote that the ultimate desire in theoretical computing was "something akin to traditional mathematical physics: a dynamical representation of a program such that, given the initial values of its parameters, one could determine its state at any later point". This work is a very good overview, but is limited in scope and discusses mostly the work of McCarthy and Scott with relation to semantics. The approach is rather technical in nature, looking at the ideas without going into too much depth; there is little in the way of contextual and biographic information.

Alongside formalism, the present work is also clearly concerned with the history of programming languages. Works in this area tend towards the very internal, such as the *History of Programming Languages* series (Wexelblat 1981; Bergin and Gibson 1996; Ryder and Hailpern 2007).[10] These conferences are 'pioneer-presented' and gather first-hand accounts which are a good source for information and criticism, but not so much for historical discussion and exploration. Such pieces also tend to be noticeably biased: take for example the debate in the discussion of Naur (1981b) where Naur's view of his role as the editor of the ALGOL 60 report is contested by Bauer. About these conferences, and similarly-written works, Mahoney (1988, p. 114) wrote "While it is first-hand and expert, it is also guided by the current state of knowledge and bound by the professional culture", and so should not be taken as a totally true and accurate representation of events.

Another interesting source on the history of programming languages is the paper on early development written by Knuth and Trabb Pardo (1976). The authors looked at a number of early coding systems and programming languages, with some contextual material giving important background to the problems of early programming, which is very useful for the current account. Knuth and Trabb Pardo also use an interesting method for discussing these languages: they encode the same algorithm in each. This trick allows comparisons between the different languages and gives a good feel for the development of coding systems. However, the paper does not consider the description methods of the languages in any detail, so there is nothing

[10] A notable exception is the programming language ALGOL, which has enjoyed a relatively comprehensive treatment by historians. See, for example, the dissertation written by Peláez Valdez (1988), and the special issue of *Annals of the History of Computing* edited by Alberts (2014a).

about formal semantics.

In fact, there exists no comprehensive history of formal description of programming languages. Typical treatments of the field (beyond technical contributions) are written by computer scientists and are technical overviews or surveys with only minor historical notes; see for example Ligler (1973), Pagan (1981), Mosses (2001), and Zhang and Xu (2004). These pieces are good for gathering data and useful for getting small snapshots of how particular approaches were seen at certain times, but otherwise have little historical merit in their own right. Haigh (2004, p. 6) notes that similar historically-flavoured introductions were seen in many early computing papers and were "designed to make the unfamiliar less threatening". There are some retrospective pieces written on formal description by the people who experienced it, such as members of the Vienna Lab (Walk 2002, 2015; Lucas 1978; Lucas 1987), and Scott (2000; 2016). These provide personal reflections and recollections, and so are understandably biased, but are good source material for interpretation.

One unique resource in this area is the biographical note on Christopher Strachey by Campbell-Kelly (1985). This explores the many avenues of his life and paints a good picture of the person, as well as covering the period in which he worked on denotational semantics and sketching an overview of that subject. While it lacks technical depth and does not consider in much detail the contributions of others to the area, it remains an extremely useful source.

Another way to look at the history of computing is to consider philosophical aspects. The Commission for the History and Philosophy of Computing (HaPoC) was established in 2011, and one of its core aims is shown in this quotation from a paper delivered at their opening conference:

> One of the aims of the commission is to organize regular meetings, providing an open platform for historians, philosophers, computer scientists, logicians, programmers, mathematicians (and all other figures involved by the field at large) to discuss across their own disciplinary boundaries and to offer the open environment required to reflect on all facets of computing.
> (Mol and Primiero 2015, p. 196).

One member of the Commission who has written more explicitly about semantics is Ray Turner. His 2007 paper is a discussion of the influence of the Platonic–Formalism divide in the philosophy of mathematics on computing; a 2009 paper

explores some of the reasons for and problems with using formalism in different styles in computing. These are good solid papers but are quite short and do not explore much historical context, instead focusing on philosophical concerns. In a recently-published book, Turner (2018) provides a comprehensive introduction to the philosophy of computing. One of the parts (Part 3) of the book explores the semantics of programming languages, which is a great reference point for the basics of the various main approaches to semantics interpreted through different philosophical lenses. However it is largely presented ahistorically: Turner gives his own presentation and form of the semantics rather than the ways they were originally displayed, and there is little in the way of contextual material.

This background survey shows there is some interest in the history of computer science, and formal/theoretical computer science specifically, and that no long-form detailed history of formal semantics exists. Great role models such as MacKenzie and Priestley exist, and organisations like HaPoC indicate a growing audience for this work. Writers like Campbell-Kelly, Haigh, and Mahoney give advice on historiography. No clear consensus emerges on the 'best' way to write this kind of history: clearly multiple approaches please different audiences and a hybrid of styles could be effective. All of this means that the present work is a clear contribution to the field, and the current author has the unique opportunity and combined skill set to provide this desirable historical account.

Now is a very important time to engage with the history of computing: for a number of reasons, stories are at risk of being lost. Some of the early practitioners are still alive (this number is sadly diminishing) and have wonderful memories to share; interviews with some of these form a core part of this work's contribution. Archives of important documents exist but are generally not properly curated (the Christopher Strachey collection at the Bodleian Library in Oxford is a notable exception). For example, the work of the International Federation for Information Processing (IFIP) is very important for many stories in computing, but there are so far no complete and accessible archives. Work is ongoing to address this problem, but until such a time, drawing together the key information from these places remains difficult and important historical documents are at risk.

1.3 Goals

The main goals of the present work are as follows: to assess the motivations of the historical actors involved in the creation of formal semantics; to look at influences and collaborations between actors; and to consider the impact of formal semantics on computing more broadly. The aim is to really understand not just the technical aspects of the work, but also the people involved, and to try to get an idea of the importance of formal semantics in the wider history of computing.

Considering the motivations of workers on formal semantics, the main question to answer is why they were interested in formal semantics. What goals did they have in mind? How did these goals influence their approaches? Were the goals satisfied, in the minds of the researchers? Was there any conflict between different goals? What were their reasons for starting work on semantics, and what kind of backgrounds did they have that led them to the subject? Mahoney (1988) observed that early researchers in computing had to come from different fields, because computing as a topic of study simply did not exist. Typically these backgrounds were in mathematics or engineering, but there were plenty of other kinds of academic or industrial experience. It is interesting to examine what impact these diverse backgrounds had on their work approaches.

Considering the interactions between historical actors is an important part of history, and especially in computing, as Mahoney (1988) observed, where industrial research groups played a much larger role than in other scientific fields. On whose work did semanticists build? How and why did groups of workers form? How did these groups differ from other researchers who were working individually? What kind of influences existed from areas outside of semantics? What interactions occurred between different groups? Did these interactions lead to changes in direction or new ideas?

On the impact of formal semantics, the aim is to look at how work on formal semantics affected computing more broadly—particularly in its early stages. What hopes were there for outcomes in formal semantics from external groups and stakeholders? Were the goals achieved according to people other than those working on semantics? What was the reception from the broader computing and industrial communities—did semantics face criticism? Was a general uptake of formal semantics achieved, as part of or instead of existing language description techniques? How did ideas generated in formal semantics get used in other areas of computing?

A number of other questions are also important to flesh out the story. Some are

simple ‘What happened?’ queries to explain the different approaches and how they changed. Others are about the people involved: as well as questions mentioned above, it is interesting to look at their biographies, personalities, and so on. Finally, considering the broader context of institutions involved, funding, and other research ongoing gives greater breadth to the present account.

1.4 Approach

As mentioned in Section 1.2, no full-length historical treatment of formal semantics exists that balances contextual, biographical, technical, and industrial elements. So taking this approach presents an entirely novel piece of work. Borrowing methods such as oral history interviews from history of science and technology, and science and technology studies, put together with technical knowledge and reading, leads to this kind of multi-dimensional work. Just as understanding, for example, the technicalities of early mechanics explains the uses and development of modern mechanical engineering (Dixit, Hazarika, and Davim 2017), the current work will inform people currently working in the field of formal and theoretical computing.

This work aims to strike a balance between answering historical and technical questions about work on formal semantics; the two different aspects inform each other. History grants insight into technical decisions: why choices were made, for whom products were created, why certain collaborations occurred, and how ideas spread. Technical reading, on the other hand, provides evidence for historical conclusions, reveals hidden influences, and helps strip away surface differences in approaches. The technical nature of some discussion herein may be less accessible to those not trained in computer science, but these parts can be skipped with no great loss to overall understanding.

There are, therefore, two main audiences in mind for the present work. Computer scientists can gain insight into the origins of their field: fundamental ideas emerge with more clarity with historical distance, and as computing is a very fast-moving field, so one can learn about the present and future from looking at the past. On the other hand, historians can appreciate the material discussing the context of the people, organisations, and situations of work covered. The present account also gathers together a large number of primary sources, making this a useful secondary source.

The sources used to compile the current document come primarily from three categories: archives, published material, and interviews. As is common in history, the

availability of sources is a limiting factor: "historians write about the material found in archives that is well cared for and accessible, and tend to ignore the material that isn't" (Haigh 2004, p. 18). There was not as much archive material available as hoped, so the majority of sources used are published—although many of these are documents like technical reports which are not widely known. Using oral history interviews offers a humanising angle and different perspectives. The particular blend of sources used here is similar to that in MacKenzie (2001).

Archive work is a crucial aspect of history. Primary sources and documents contained in archives show, for example, early drafts and the working out of ideas, and correspondence and views on other people working in the area. They can be used to build up biographical information and round out pictures of events such as conferences. Minutes of meetings and other documents of organisations offer glimpses inside these communities.

The extant archives identified as relevant to the present account are those of Heinz Zemanek, Christopher Strachey, Aad van Wijngaarden, Edsger Dijkstra, Andrei Ershov, and Peter Landin. The Dijkstra and Ershov archives are available online, digitised, and are searchable to an extent. Dijkstra's has some formal semantics–related content, but Dijkstra was largely tangential to the present account, so his papers are mostly useful for correspondence. Ershov did not work on formal semantics, but was a member of many relevant IFIP communities and so has a mostly-complete collection of papers. The Strachey archive at the Bodleian Library in Oxford is not digitised, but it is the largest and richest collection of those mentioned. It has a great deal of material on his life (such as bus tickets, cheque stubs, and passport photos) and lots of letters—the academic references proved particularly useful. The archive also contains many working papers from his time developing denotational semantics, ranging from typescripts of published pieces to handwritten pencil worksheets. The Zemanek archive was at the time of research in the process of being indexed and sorted: it is huge and currently inaccessible to researchers, but thanks to the archivist Juliane Mikoletzky some documents were made available which provided an insider view of the work at IBM. The van Wijngaarden archive, held at Universiteit van Amsterdam, was made available thanks to Gerard Alberts, but it is entirely uncatalogued and unsorted, so it was only possible to extract some material. The Landin collection is promising, but is currently inaccessible: it is held by the Bodleian Library but is not indexed or archived.

Published material was available from online sources such as the ACM Digital Li-

brary, through Newcastle University Library, and also Cliff Jones' large personal collection. Following traces through citations led to many new discoveries of sources; conferences were particularly valuable given the tendency in the 1960s to record discussions as well as papers (especially the *Formal Language Description Languages* conference.) Technical Reports were a useful source for Vienna publications, but a series of Programming Research Group monographs were also studied for the Oxford story. TRs make a good complement to final published papers, often showing earlier versions or the working out of ideas. This can be combined with archive material to see the trajectory of idea development. Again, many of these reports were available from Jones' collection.

During the research period of the current work, ten oral history interviews were recorded; the full list is available in Appendix B. The author received training from the Oral History Society delivered at the British Library. This helped develop an approach to interviews as follows. First, prepare a list of questions in advance, both those that required specific answers, and general topics for discussion. Next, communicate broad topics with the interviewee, allowing them to prepare and gather materials, but not fully write out answers in advance. The idea is to try to eliminate moments of the interviewee saying something like "What was that thing? Let me try to find it..." while preventing a completely scripted discussion.

On the day of the interview, the aim was for a discussion lasting one to one and a half hours. It was important to try to get the interviewee alone in a quiet place to minimise both distractions and background noise. For the interviews, one of two main pieces of equipment were used: a Tascam DR-44WL or Zoom H1. The approach was to direct the conversation at first, but allow topics and avenues of discussion to develop organically. It was important to accept that questions would not end up being answered in precisely the same order or manner as planned. The interviewer choosing to sit back and keep themself at a minimum during interviews allowed the focus to be on the interviewee as much as possible—the interviewer should be a prompter and not an active participant. Specifically asking for anecdotes results in more human information coming out and allows the fleshing out of historical actors as real people, and facilitates getting a sense of what working in the field of formal semantics was really like.

Transcription was done in full for all interviews with the aid of the Express Scribe software, which allows control of playback speed as well as automatic stopping and starting so a comfortable pace of writing can be achieved. Paying a transcriber was

considered, but as well as being costly, the relatively high number of technical terms and given names in the interviews would have caused problems. Additionally, the time spent with the recording while transcribing allows reflection on the information and better learning from it. As well as full transcriptions, summaries were written of each interview to allow easier reference.

In addition to fully transcribed interviews, there were also detailed discussions held with Peter Mosses and Kurt Walk. These were noted but not recorded, and are referred to within the present work as 'personal communication'; also included under that banner are some useful emails exchanged. Another source was interviews conducted by others, particularly by Tony Dale for MacKenzie's book.

As with any historical work, there are a number of pitfalls to be avoided. Looking back at the work on semantics, it is tempting to talk about 'failures' of some pieces of work; but doing so clouds the view of optimism that surrounded a lot of workers at the time and ignores the fact that others can learn from apparent dead ends. This point of view is highlighted by Knuth (2014) who explained the technical utility of learning from mistakes. Mahoney (1996b) argued that this is also essential in history.

Another important mistake to avoid is trying to force ahistorical definitions on the past. Conversely, the diversity of phrasing used during the emergence of a field means that sticking solely to works described as 'semantics' would miss some crucial developments, so the present work looks at historical efforts that formalised meaning even if the term was not expressly used.

Finally, using oral history interviews as source material (as the current work does) brings well-known risks: anachronisms, misrememberings, and reconstructed memories. Mahoney (1996b, p. 832) cautions against viewing historical events in present lens and that actors may reconstruct their memories in light of later work: "Precisely because they helped to create the present, they are prone to identify it with their past work or to translate that work into current terminology". For example, they may say "of course, what I was really doing was..." which loses the important context of original work. During interviews, however, one should try not to directly contradict interviewees, as building a good rapport is vital. It is also extremely important to be supportive and understanding with interviewees who have only vague recollections left, or entirely lost memories. Many oral history interviewees are necessarily old, and directly confronting failing memory can be distressing. The interviewer should avoid pushing an agenda during an interview, but should be

critical of the source once it is produced during an historical analysis.

Bringing together the three main sources mentioned in this section leads to a reasonably comprehensive understanding of the story of formal semantics. Archives and technical reports provide information on the development of ideas, and published pieces can be read technically to show the workings of methods, as well as showing influences and determining motivations. Looking at work outside the area of formal semantics that links back to it shows the impact of the ideas. More background material comes from secondary sources such as MacKenzie (2001) and Campbell-Kelly (2003), and looking at organisational documents. The human angle is added from some archive material as well as oral history interviews, which also serve to explore motivations.

1.5 About the present work

A superficial glance will show that this essay is long and its notions elaborate. The original scope planned for the work was to cover all research on formal semantics up until about 1985; however, to address length concerns, some cuts had to be made. Rather than limiting depth, it was decided instead to narrow breadth of study, and so the main focus of the present work is on two centres of semantics: the IBM Laboratory Vienna, and Oxford University's Programming Research Group. A additional advantage to choosing these two centres as a focus is that there was obvious interaction and influence going on between the two. However, proper explanation of these stories benefits from a study of some prior work, the better to discuss influences and motivations. Some further stories occurring outside the main work of these two centres serve also to illustrate the impact of their work. These show the expansion of ideas and facilitate drawing comparisons about approaches.

The most obvious omission from the present account is the approach known as axiomatic semantics, and more generally algebraic approaches. It seemed better to omit these entirely rather than treat them cursorily; a forthcoming book based on this dissertation could include a proper treatment of this important approach to semantics.

During the PhD research period associated with this dissertation, two papers were co-authored with Cliff Jones.

- 'Formal Semantics of ALGOL 60: Four Descriptions in their Historical Con-

text' (Astarte and Jones 2018) examines four full descriptions of ALGOL 60.[11] The paper contains a historical account of the background to these descriptions, as well as more in-depth technical discussion of the approaches used (the fact that each treats the same language but in very different ways provides a good vehicle for comparison of the semantic styles). Historical material in that paper appears mostly paraphrased in the present work in various places, and is noted with citations. The technical material is not repeated here.

- 'Challenges for semantic description: comparing responses from the main approaches' (Jones and Astarte 2018) is a more technical comparison and survey of approaches to semantics, with some historical elements. Material from that paper is not repeated in the current work, but is frequently cited as a further reference.

The current work is, as mentioned, structured around the central Vienna and Oxford stories. Determining an appropriate ordering of the material was difficult: the final arrangement is broadly chronological but as plenty of work happened in parallel there is no strict progression. Back and forward references are used to aid the reader, particularly where there are comparisons between content under discussion in the immediate section and that from previous sections.

The structure of this dissertation is as follows. Chapter 1 is this introduction. Chapter 2 covers the early history and background to the main stories, setting the scene for the emergence of formal description. In particular, the history of programming languages is sketched, with particular attention given to ALGOL. Chapter 3 describes the work of some important early researchers: McCarthy, Landin, and van Wijngaarden. Chapter 4 discusses the key event that codified the field: the *Formal Language Description Languages* conference held in September 1964. Chapter 5 tells the story of the efforts of the IBM Laboratory Vienna to define the programming language PL/I. Chapter 6 explains how Strachey and others developed, partially in response to earlier work, a more mathematical approach to semantics. Chapter 7 contains four further stories, including the controversial design of ALGOL 68, IFIP's Working Group 2.2, a key discussion forum, the shift of the Vienna Group to a denotational framework with VDM, and finally SOS, a synthesis of ideas from different approaches that is probably the most widely used approach today. Chapter 8 concludes the work with historical and technical remarks, including examining the impact and criticism of formal semantics. Finally, appendices include a list of abbreviations used, a table of interviews conducted, and an index (particularly of

[11] This work was first presented by the current author in Paris in June 2016.

people involved). Each chapter includes a timeline of major events; a summary timeline is displayed in Figure 1.1.[12]

A brief note on references to parts of works: unqualified mention of 'Section', 'Chapter', or (rarely) 'Page' refers to elements of this dissertation; where references to external works are made, '§', 'ch.', and 'p.' will be used (respectively), typically within citation brackets.

[12] Adapted from `https://tex.stackexchange.com/a/198372`.

Figure 1.1: A summary timeline of major events in the history of programming language semantics.

Late 1950s	Mervyn Pragnell's reading groups attract many early computer scientists, including Burstall and Landin
Jun 1959	Strachey sets up a consulting business and employs Landin, who gets him interested in lambda calculus
Dec 1960	Publication of the Report on ALGOL 60
Jan 1960	Formal opening of IFIP
Mar 1962	Publication of van Wijngaarden's symbol manipulation approach to language description
Aug 1962	McCarthy introduces abstract syntax and interpreters
Jan 1964	Landin publishes his lambda calculus / SECD machine
Sep 1964	The Formal Language Description Languages conference
Dec 1964	The IBM Laboratory Vienna is founded
Apr 1966	Strachey sets up the Oxford Programming Research Group
Dec 1966	Publication of Vienna's first formal definition of PL/I
Sep 1967	First meeting of IFIP WG 2.2
Dec 1968	Lauer writes a definition of ALGOL 60 in the Vienna style
Dec 1968	ALGOL 68 is approved as an IFIP publication, codifying two-level grammars and introducing context conditions
1969	Burstall uses relations in a description of ALGOL 60
Apr 1969	Strachey and Scott meet at a WG 2.2 meeting and Hoare presents axiomatic semantics
Late 1969	Scott develops a logical foundation for Strachey's semantics
Aug 1970	Jones and Lucas show that VDL can be used for proving correctness of implementations
Jan 1974	Publication of Wadsworth and Strachey's paper on continuations, denotational semantics for jumps
Jan 1974	A formal description of ALGOL 60 is written by Mosses
Dec 1974	The Vienna Lab writes a VDM formal description of PL/I
May 1975	Strachey suddenly dies of infectious hepatitis
1978	A book on VDM is published, edited by Bjørner and Jones
1981	Plotkin's 'Aarhus notes' lay out SOS

CHAPTER 2

Background

Understanding the historical development of formal semantics requires some background on the state of computing as the work emerged. The present essay is not the place for a full history of computing machines, the software running on the machines, or even the languages in which that software is written. Many other such histories exist in the literature already.[1] However, a brief overview of the *context* in which work on formal semantics appeared is useful in order to better understand the motivations of the actors involved. This chapter provides just such an overview. Section 2.1 discusses the emergence of electronic computational machines as they impact the present work. Section 2.2 presents early work on the methods for writing down programs to control computers. Section 2.3 is a deeper dive into one particular language, ALGOL, which has extra relevance to the story of formal language description. A timeline of the major events detailed in this chapter can be found in Figure 2.1.

2.1 Emerging machines

The Second World War was seen as a "scientific war" and its end brought about a boom in science in the victorious countries (Campbell-Kelly and Aspray 1996, p. 65). In Britain, for example, spending on civil and military research development increased significantly towards the end of the 1960s: in 1945–6 expenditure on civil

[1] See, for example, Campbell-Kelly and Aspray (1996) and Campbell-Kelly (2003) on machines and the software industry; Wexelblat (1981) on programming languages; and Priestley (2011) for a look at the impact of logic on programming.

Figure 2.1: A timeline of important events in the background story of machines and programming languages.

R&D was £6.5 million; by 1962–3 this had risen to £150 million (de Chadarevian 2002, p. 35). Electric calculating machines had been seen as useful contributors to the war effort due to, for example, their role in anti-aircraft gun control and ballistic calculations (Randell 1975, Ch. VII). This led to a commensurate increase in the funding of research into the creation of more of these machines. This research led to the construction of computing machines of increasing power and flexibility, and to the "scientific" culture of computing dominant during the 1950s (Eden 2007).

Although there was enthusiasm for the creation of computing machines around the world, it was the United States of America that led the charge. Britain possessed a strong body of researchers but failed to convert this into commercial success and a number of business ventures ended up floundering (Hendry 1989). By the end of the 1950s, the US was in a dominant position; at this time the computer moved from being primarily a research and academic tool to a "business machine". The company embodying that shift most of all was IBM, which quickly grew to be the market leader in computers, despite its initial reluctance to get into the computing business just after the end of the war due to the instability of the market (Campbell-Kelly and Aspray 1996, Ch. 5).

The IBM computing series started with the release of the Model 701 in December 1952. IBM had been a manufacturer of other business machines (e.g. the Model 604 card punching calculator) and the inclusion of computers within this product model drove the view that they had a role in the workplace. The 701 was many companies' first computer: George Ryckman of General Motors Research Laboratory noted (in Backus 1981, Discussant's Remarks) that a 701 was GM's first computer purchase. The release in 1959 of IBM's Model 1401, however, was the commercial breakthrough, and also ushered in a new era of computer *systems*, where peripherals such as punch-card readers and printers were treated as an important part of the product. Another very important IBM system, the 704, was released in 1954, and was notable for being the computer for which a number of the first high-level programming languages were written.[2]

IBM was not the sole player in the computing field, but smaller companies had a hard time competing due to the very high initial costs of setting up a manufacturing business. One such company produced the UNIVAC line of computers, and had been set up initially by J. Presper Eckert and John Mauchly in an at-

[2]See more discussion in Section 2.2.

tempt to commercialise their early success at creating computing machines in the 1940s. They founded the Eckert-Mauchly Computer Corporation, but, struggling to survive, were bought by Remington Rand alongside the Engineering Research Associates company. Remington Rand itself was still fighting to compete with IBM, and, despite a merger with Sperry Gyroscope to form Sperry Rand, still was unable to become a leader (Campbell-Kelly and Aspray 1996, Ch. 5).

In Europe, however, the situation was much different: computers at the end of the 1950s were still largely held by scientific or research establishments. Many universities built their own computers, either directly in-house, or in spin-off companies such as Electrologica, which spun off from the Mathematisch Centrum in Amsterdam in 1956. In the UK, the government's National Research Development Corporation poured money and guidance into small computer manufacturers, but these were still unable to compete with IBM.[3] This closer tie between the creation of computers and centres of research in Europe may be a contributing factor to the differences in attitude towards theoretical computing in Europe and America.

Another part of the move towards thinking of computers as systems rather than individual machines was the growing importance of software. Initially, computers were sold purely as hardware, and users were expected to write their own programs (perhaps with the aid of a manufacturer-written manual). In the late 1950s, however, there was a shift towards software being available as a corporate product, and the importance of people who were able to produce these programs grew. The need for skilled programmers was paramount and many companies feared there was a real shortage of people with the appropriate abilities. A recruitment drive in 1956 from the System Development Corporation, the largest American software contractor, featured Tom Steel imploring readers to join their company (Campbell-Kelly 2003, p. 28).[4] The profession of 'programmer', however, was still something rather new: Edsger Dijkstra, a very influential Dutch computer scientist, noted that when he was married in Amsterdam in 1957 he had some trouble identifying as such:

> Dutch marriage rites require you to state your profession and I stated that I was a programmer. But the municipal authorities of the town of Amsterdam did not accept it on the grounds that there was no such profession. And, believe it or not, but under the heading 'profession' my marriage record shows the ridiculous entry 'theoretical physicist'!
> (Dijkstra 1972, p. 860, quoted in Peláez Valdez 1988, p. 6)

[3] An account of the early British computing industry is presented by Hendry (1989).

[4] See Page 145 for a reproduction of this advertisement.

As well as the universities performing fundamental research on computation and businesses manufacturing and selling machines, other organisations grew up around these centres and the users of computers. Some of these, like the Association for Computing Machinery (ACM) in the US and the British Computer Society (BCS) in the UK, were paid-membership bodies interested in furthering the research and professional interests of their members and of the area of computing as a whole. A particularly important organisation to the present story was the joint academic-business International Federation for Information Processing (IFIP), which hosted a vital early conference and sponsored a number of relevant working groups. IFIP began with funding from UNESCO under Isaac Auerbach in 1955, but was formally opened in 1961 (Auerbach 1986b).[5] Also relevant were organisations interested primarily in standardisation: the International Organisation for Standardisation (ISO) and the European Computer Manufacturers' Association (ECMA) provided impetus for unifying practices and processes involved in computing. Finally, 'user groups' brought together users of the same or similar machines with a view to sharing knowledge and experience of these computers (Nofre, Priestley, and Alberts 2014, p. 55). The idea was to prevent replication of work and campaign for standardisation. Examples of such groups are IBM's SHARE group (despite its capitalisation, not an acronym) and the Univac Scientific Exchange USE. One critical role these groups played was in the push for high-level programming languages, and to properly understand that desire requires a brief overview of the development of programming.

2.2 Programming languages

The very first computers, such as the Manchester Small-Scale Experimental Machine (nicknamed Baby) were operated simply with console switches. More sophisticated later machines were controlled by giving a series of 'orders' to the machine, and as these were encoded to be machine-readable, they were known as 'order codes'. Thus the task of writing a program was 'coding' (Priestley 2011, p. 157). The relative paucity of basic commands the machine could perform led to a simplicity in these order codes, as Peláez Valdez (1988, p. 4) writes:

> There was a very close correspondence between the structure of the program and the structure of the machine itself. Consequently, programmers were required to know every detail of the structure and working

[5] See Zemanek (1986a) for an extensive history of early IFIP.

> of the machine they were programming and inevitably the focus in programming was on the formulation of the problem to fit the structure of the machine; the logic of the program was totally shaped by the structure of the machine.

A further complication was that such codes were often almost unreadable to a human: as the machine ultimately accepted instructions only in binary numbers, orders were typically fed to the machine encoded as numbers. Grace Hopper (1981, p. 7), developer of early coding systems and later one of the first programming languages COBOL, noted "in the early years of programming languages, the most frequent phrase that we heard was that the only way to program a computer was in octal".

An illustrative example of these early order codes, shown in Figure 2.2, is taken from Campbell-Kelly (1980a): an algorithm encoded for the Cambridge EDSAC, a machine built in the late 1940s. The particular algorithm is 'TPK' from Knuth and Trabb Pardo (1976); it does not perform any meaningful operations but was used by those authors to showcase language features in a number of different early programming systems. The importance here is not the content of the program, but rather observation of the way in which it is written: not an intuitive notation (the annotations are Campbell-Kelly's), and the particular commands used all relate very specifically to the operations of the low-level machine architecture. The presentation of the EDSAC's order code was issued in a series of books, starting with co-authored released by Wilkes, Wheeler, and Gill (1951). According to Campbell-Kelly (1980a, pp. 10–2), this book, essentially the first textbook on programming, describing the method by which programs could be 'prepared' for the EDSAC by listing all the possible order codes, describing their effects in terms of machine operations, and providing a comprehensive series of example cases.

Machine codes like the EDSAC's dealt solely in the lowest-level operations available to the machine: loading values from registers, clearing registers, and performing basic arithmetic operations. This usually excluded 'floating point' arithmetic (the manipulation of numbers containing a decimal point whose position could be anywhere within the string of digits) which, of course, was a commonly required operation. So coders were forced to spend their time painfully writing out the same methods for handling these numbers—and other simple, repetitive tasks—in every program. A further problem was noted by Strachey (1971c, p. 2), who argued that programs written in low-level machine instructions were *too* detailed to reveal their important aspects:

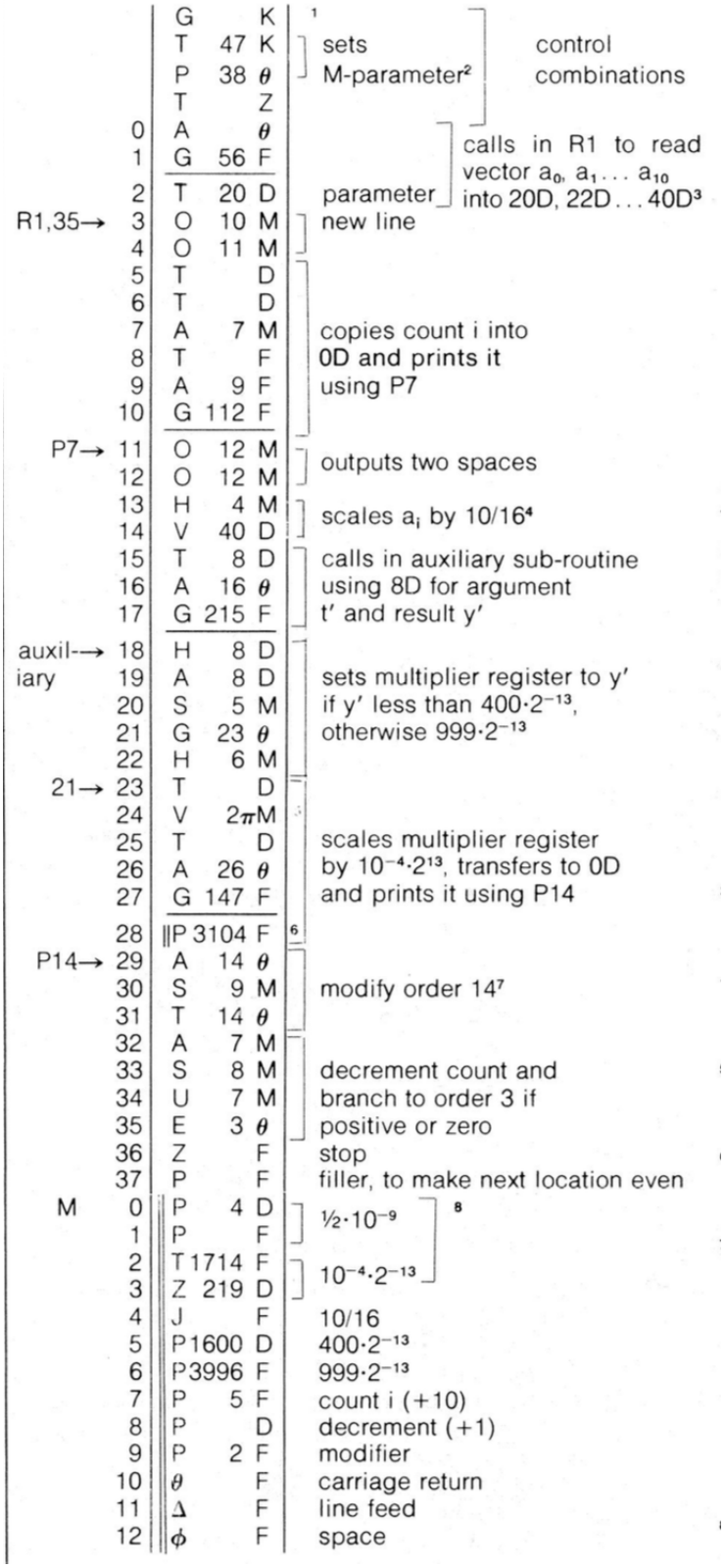


Figure 2.2: Representation of the TPK algorithm in EDSAC machine code. Taken from Campbell-Kelly (1980a); the annotations to the left and the right of the main column are his.

> But programs of this sort are too atomised to be at all convenient. Like pointilliste paintings, they conceal their form in a maze of fine detail, but without the possibility one has with pictures of standing back to get a broader view.

A great leap forward around the end of the 1940s was the introduction of stored program computers, which enabled a change in the way machines could be controlled. As instructions could be stored as data, the computer could then manipulate its own program, replacing a more abstract instruction with an appropriate series of lower-level commands, or re-ordering instructions depending on certain conditions. Early machine designers, such as Turing and Von Neumann, realised this, and began to create richer and more abstract programming systems (Priestley 2011, pp. 157–8).

One such approach grew out of the recognition that certain sequences of commands, commonly called 'subroutines', were used repeatedly. Hopper (1981, p. 8) recalls that she and her colleagues would handwrite these in their notebooks and swap them: "If I needed a sine routine ... I'd whistle at Dick and say 'Can I have your sine subroutine?' and I'd copy it out of his notebook." The next step was to store these subroutines in the computer's memory, and from there programs could be written with reference to these. Because these programs compiled subroutines from various different locations, the program used to pull them together became known as a 'compiler' (Beyer 2009, p. 223). Nora Moser, a colleague of Hopper, explained:

> To compile means to compose out of materials from other documents. Therefore, the compiler method of automatic programming consists of assembling and organizing a program from programs or routines or in general from sequences of computer code which have been made up previously.
> (Moser 1954, p. 15, quoted in Knuth and Trabb Pardo 1976, p. 51)

A variant of this approach continues today with static-compiled languages. For more on the contributions of Hopper to early coding, see Beyer (2009, Ch. 3).

Another method of handling abstract programs was to write an 'interpreter routine' which would take as input a program written in a richer coding notation and translate those commands on the fly to machine code commands (Knuth and Trabb Pardo 1976, p. 50). This was presented in the work of, for example, Wilkes, Wheeler, and Gill (1951), and the core concept is still used for many dynamic and scripting languages.

```
c@VA t@IC x@½C y@RC z@NC
INTEGERS +5 → c
    →t
  +t     TESTA Z
  −t
         ENTRY Z
SUBROUTINE 6 → z
  +tt → y → x
  +tx → y → x
+z+cx CLOSE WRITE 1
a@/½  b@MA  c@GA  d@OA  c@PA  f@HA  .@VE  x@ME
INTEGERS +20 → b  +10 → c  +400 → d  +999 → e  +1 → f
LOOP 10n
  n → x
+b − x → x
  x → q
SUBROUTINE 5 → aq
REPEAT n
  +c → i
LOOP 10n
  +an SUBROUTINE 1 → y
  +d − y TESTA Z
  +i SUBROUTINE 3
  +e SUBROUTINE 4
       CONTROL X
       ENTRY Z
  +i SUBROUTINE 3
  +y SUBROUTINE 4
       ENTRY X
  +i − f → i
REPEAT n
ENTRY A CONTROL A WRITE 2 START 2
```

Figure 2.3: The TPK algorithm encoded in Autocode.

There is an important distinction between the compilation of subroutines and the translation approach. Use of routines brings together pieces of code from disparate locations and collates it into one larger piece of code and could be performed by a simple text replacement system.[6] An interpreter allows the writing of programs which extend the repertoire of operations available to the programmer: these can be composed and manipulated at a higher level (Jones and Astarte 2018, p. 179). However, it is critical to realise that many systems employed both of these concepts

[6]Hopper (1981, p. 10) described her early compiler as "a program which, given a call word, pulled a particular subroutine. Following the call word were the specifications of the arguments and the results which were to be entered into the subroutine with no language whatsoever. The programs were butted, one bang against each other".

and combined them for best effect, especially increased efficiency, and to try to separate out the two with any great deal of clarity misses the way in which these concepts were employed historically. The key point is that coding systems became more flexible and powerful in the level of abstraction they employed, allowing more complex instructions to be written in a way easier for coders to understand.

An illustrative example of this level of code is shown in Figure 2.3, which shows the same TPK algorithm as in Figure 2.2, this time encoded in Autocode.[7] There is a clear improvement on abstraction and readability compared to machine order code. However, the system is still very machine-dependent: "Test A" near the beginning refers to a particular register rather than to some abstract variable A, and $+t$ refers to the loading of one of the machine's two accumulators. The structure of the program is built around the definition and calling of a number of subroutines, with a few control structures around.

Autocode was developed by Alick E. Glennie at a military research department housed at Fort Halstead in the UK, and was later taken up by the users of the Manchester Mark I machine. It is interesting to note that while a number of different systems grew from the Autocode idea, its origins with Glennie were not well-recognised, and Glennie's system was not mentioned by the time Tony Brooker (1958) was writing about using an Autocode system at Manchester, despite this having developed originally from Glennie's work (Knuth and Trabb Pardo 1976, p. 48). One reason for this may have been that Glennie's original papers on the topic were never published, perhaps due to the secretive nature of military research still pervasive after the finish of the war.

Glennie recognised the importance of his work, saying in a lecture delivered at Cambridge University in February 1953:

> The difficulty of programming has become the main difficulty in the use of machines ... *To make it easy, one must make coding comprehensible.*[8] This may only be done by improving the notation of programming. Present notations have many disadvantages: all are incomprehensible to the novice, they are all different (one for each machine) and they are never easy to read. It is quite difficult to decipher coded programs even with notes, and even if you yourself made the programme several months

[7] This example is taken from the talk given by John Backus (1981, p. 48) at the first *History of Programming Languages* event rather than directly from the Knuth and Trabb Pardo paper because Backus' is formatted more nicely.

[8] Original emphasis.

ago.[9]
(Glennie 1952, cited in Knuth and Trabb Pardo 1976, p. 43)

Although there was a great deal of variety in notation and precise functionality of the coding systems that emerged in the 1950s, they were all essentially variations on the same idea: a computer could be issued commands in a notation that was easier for the programmer and in some way manipulate this to create an executable program in its own order code. Glennie recognised, in a letter to Knuth written in 1965, that this idea was very present at the time:

> I recall that automatic coding as a concept was not a novel concept in the early fifties. Most knowledgeable programmers knew of it, I think. It was a well known possibility, like the possibility of computers playing chess or checkers. ... [Writing the compiler] was a hobby that I undertook in addition to my employers' business: they learnt about it afterwards. The compiler ... took about three months of spare time activity to complete.
> (Glennie, quoted in Knuth and Trabb Pardo 1976, p. 49)

Another coding system, EASICODE,[10] was also developed in spare time: at English Electric in Whetstone, Brian Randell and Mike Kelly were working on a way to get more time out of the Pilot DEUCE machine in their department. According to Randell (2016), its convoluted instruction set, complicated by the intricacies of its delay line memory system, meant that in order to cope with the computer workload, the machine had to be on all day and all night. The DEUCE was optimised at the machine code level for matrix operations and linear algebra, which meant its performance doing scalar arithmetic was sub-optimal—losing a time factor of up to 50. Starting working at night, Randell and Kelly began experimenting with a better coding system, which exploited the delay factor in the memory in order to produce better speeds. A crucial part of this experimentation was a sense of fun: "Mike and I were the only ones who really delighted in the computer, regarded it as a giant toy", recalled Randell (2016), an attitude not shared by their boss. "He was a mathematician. To him the computer, was something you used, not something you played with." Randell continues:

> And so, we had this idea for EASICODE. Nobody had asked us to do this: we started out implementing it, and our boss found out about it,

[9] These concerns may seem very strange to the programmer of today given how successfully we have eliminated them since 1953.

[10] EASICODE was published by Kelly and Randell (1958).

> and wasn't very pleased. Initially he agreed that we could work on it at night if we wanted to, and we did that, and were making fairly good progress. I'm not quite sure what triggered it, perhaps the amount of time we were spending at night on it: he banned us from doing any more work on the computer, on EASICODE. It was a pretty tense meeting. We were close to resigning or being fired, I guess. But we agreed that we would not, for one year, do any further work on EASICODE. We took all of our documentation away—didn't leave it within English Electric—and a year to the day we marched back into his office and demanded to be allowed to go back to working on EASICODE. We finished it fairly soon afterwards, and the first use of it was by one of our customers, one of the engineers. His application, which he'd already tried doing using one of the other interpretive systems, rapidly provided evidence that he had saved more computer time than we had spent on the entirety of building of EASICODE.

The presentation of EASICODE, by Kelly and Randell (1958), describes the system as a 'translation programme' [sic] and, like the EDSAC order code, it is described in terms of machine operations. An additional level of abstraction is present, however, by the fact that the code calls DEUCE orders—described as 'functions',[11] and labelled by numbers. The presentation of an EASICODE program is tied to its formatting on punch cards: columns have a meaning, with the first for the DEUCE function to be called, the second a memory location to be read, and so on. The report does not use any formalisation and is much more like a coder's manual.

Randell and Kelly's boss might not have realised the significance of EASICODE, but the ideas were gaining traction in the growing computing community. One person who was aware of the importance of the work on coding systems, and Autocode in particular, was Christopher Strachey (1952),[12] who wrote favourably about the system, although he too failed to attribute it to Glennie. Strachey was writing about the importance of paying attention to non-mathematical problems and how they could be handled by computers, pointing out that the large part of the programmer's work was tied up in working out how to pose their problems in a way the computer could handle, with an additional toll posed by the need to encode all the control and input/output management instructions. Strachey gave an example showing that a program for the integration of a non-linear differential equation took about 400 instructions total, of which only 75 were actually involved in the math-

[11] In the sense of machine functionality rather than the mathematical meaning.

[12] This paper also presented an early draft of Strachey's famous draughts-playing problem; something Glennie mentioned as being an interesting possibility in the quotation on page 35. Much more is written about the contributions of Strachey in Chapter 6.

ematical calculation. So systems which could ease the load of the programmer in this regard could be very beneficial indeed.

John Backus, who worked on one such system as well as on many high-level programming languages later, had the following to say on the subject, in an interview from 1979:

> Much of my work has come from being lazy. I didn't like writing programs, and so, when working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs for the 701. And that wound up as something called Speedcoding.[13]
> (Stegmann and Backus 1979)

As computer systems grew in power throughout the 1950s, the arguments against coding systems—that compilation took up valuable machine time and the object code provided was not as efficient as that produced directly by a good programmer—began to lose impact. Peláez Valdez (1988, p. 15) explains that the turning point came with the release of the IBM 704, a machine not only with floating-point arithmetic built into the hardware (obviating the need for a very commonly-used kind of subroutine), but sufficiently increased memory and power to cope with a considerably more abstract programming system. John Backus (1980, p. 130, quoted in Peláez Valdez 1988, p. 15.) provided an argument for just such a system that appealed to the "economics of programming":

> FORTRAN did not really grow out of some brainstorm about the beauty of programming in mathematical notation; instead, it began with recognition of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became cheaper this imbalance would become more and more intolerable. This prosaic economic insight, plus experience with the drudgery of coding, plus an unusually lazy nature led to my continuing interest in making programming easier.

A further motivation for the development of languages with a greater level of abstraction[14] was provided by user groups, as mentioned at the end of Section 2.1. An important function of these groups was to enable the sharing of work: standard

[13] The Speedcoding system was presented in Backus (1954b).

[14] The term 'high-level language' came to be the popular phrase to refer to such languages, although this only really began to take off in the mid-1960s.

libraries, commonly-used routines, and so on; this necessitated a common form of notation and sufficient compatibility between different models of machine. This case is made by Nofre, Priestley, and Alberts (2014, p. 56) and they point out the role of these groups in pushing the use of the language metaphor to describe programming systems, noting that USE was particularly active in this area: "in December 1955, the first meeting of USE agreed on the adoption of a 'common programming language for exchanged programs' as one of the top priorities of the newly established cooperative organization."

Although there were a number of different coding systems around at the end of the 1950s, some of which have been mentioned already in this section, the one which achieved the greatest success was FORTRAN. At the urgings of IBM's Programming Research Group, the designers of the IBM 704 incorporated time-saving features from Speedcoding, such as floating point arithmetic, directly into the machine's hardware. "From then on," said Backus, of the Programming Research Group, "the question became, what can we do for the poor programmer now? You see, programming and debugging were the largest parts of the computer budget, and it was easy to see that, with machines like the 704 getting faster and cheaper, the situation was going to get far worse" (Stegmann and Backus 1979).

The next step was a translation system that would allow writing of programs in a notation not too dissimilar from standard mathematics and was first described, as FORTRAN, in a preliminary report (Backus 1954a). According to Backus (1981, p. 27), the core concepts came from earlier ideas by Laning and Zierler; Priestley (2011, p. 197) added that as well as allowing translation of mathematical concepts, FORTRAN brought in the benefits of control structures that were well-recognised in autocoding. The program which took FORTRAN input was always referred to as a 'translator' rather than 'compiler', although by 1981 when Backus was reflecting on the history of FORTRAN, the term 'compiler' was firmly cemented as the universal (Backus 1981, p. 33). It is interesting to note that, despite his key role in the development of FORTRAN, by the late 1970s Backus was becoming increasingly dissatisfied with what he called 'the von Neumann style' of programming and was moving towards a functional approach instead. This can be seen in interviews with him, such as that of Stegmann and Backus (1979) and his Turing Award acceptance speech (Backus 1978).

FORTRAN, then, marked the beginning of a real shift in programming towards compactness and abstraction as compared to programs written in either machine

```
1         DIMENSION A(11)
2         READ A
3     2   DO 3,8,11 J=1,11
4     3   I=11-J
5         Y=SQRT(ABS(A(I+1)))+5*A(I+1)**3
6         IF (400. >=Y) 8,4
7     4   PRINT I,999.
8         GO TO 2
9     8   PRINT I,Y
10   11   STOP
```

Figure 2.4: The TPK algorithm presented in FORTRAN O.

order code or autocode. This is illustrated in Figure 2.4 which shows the TPK algorithm seen in Figures 2.2 and 2.3 encoded in FORTRAN O, the notation of the 1954 Preliminary Report. This example is taken directly from Knuth and Trabb Pardo (1976). Compared to the other TPK encodings shown in this Section, it is clear the level of abstraction is drastically improved; the modern reader will likely be able to understand the majority if not entirety of this program without any explanation.

Although the Preliminary Report for FORTRAN was published in 1954, work on the translation system carried on through to 1956, and the language was officially published in a reference manual from October of that year (Backus et al. 1956) as well as an academic paper the following year (Backus et al. 1957). In the reference manual, the language is presented using informal textual descriptions and examples, much like one would expect in a mathematical textbook. The paper is essentially an extended series of example FORTRAN programs and snippets. The language used in these works is deeply rooted in machine notions: the meaning (the word 'semantics' is not used) of the FORTRAN programs and constructs is described in terms of the machine (specifically the IBM 704).

The definition style reflects FORTRAN's status as a kind of intermediary between the coding systems of the early 1950s and the high-level languages that came at the end of that decade and into the 1960s.[15] Coding systems generally had very clear connections to their target machines and were notations whose primary purpose was to save the programmer time. The 'meaning' of such languages tended to be fairly easily understood (even if the notation was somewhat baroque) as it was built up from the knowledge of the machine in a bottom-up fashion. This can be seen in the method of presentation of the EDSAC order code which simply lists each order's effect on the machine's components, and EASICODE's description in terms of the DEUCE's built-in functions. Walk (2002, p. 80) argued this point:

> With machine and assembler languages, whose structure was fairly simple, programs consist[ed] of equally formed statements whose meaning was determined by the status and the changes of the hardware components they referred to.

The emerging high-level languages represented a significant departure from this, although as the shift happened gradually it was not immediately obvious. The greater abstraction of FORTRAN and its successor languages allowed much easier formulation of programs: meaning was constructed in a top-down approach from the knowledge of mathematics. However, this led to a disconnection from understanding flowing from clear correspondence with machine instructions. This was to cause some trouble, especially when attempting to prove correctness of programs.

FORTRAN was, of course, not the only higher-level programming language emerging towards the end of the 1950s. COBOL, an attempt to create a FORTRAN-like language for data processing rather than mathematics, was built out of an earlier system called B-0. A good history is given by Sammet (1981); it will not be covered here because its approach was neither algorithmic nor algebraic, and therefore not treated in the same way by theorists interested in programming languages. LISP, a purely functional language for manipulating lists of data is relevant, but is covered in Section 3.1.

Not only was FORTRAN not the last word on programming, but the proliferation of different coding systems and languages was causing concern for portability and

[15] This should not be taken as suggesting that FORTRAN only served to prepare the world for the languages with greater abstraction which came later. FORTRAN continued to enjoy much success well into the 1970s and '80s and some systems today still run FORTRAN programs. Although FORTRAN was undoubtedly deeply influential on later events, its importance as a system in its own right cannot be overlooked.

communication; according to Endres (2013, p. 3), "IBM offered six to eight different language processors for each of its large machines". An issue of the *Communications of the ACM* from January 1961 had an image of the Tower of Babel on it, reflecting the "confusion and lack of communication that was felt to exist at the time" (Priestley 2011, p. 204). These concerns were not new: the desire for a universal (i.e. cross-machine) programming language was discussed at a symposium on automatic coding organised by the Office of Naval Research in May 1954. Two people who voiced their worries were John W. Carr and Saul Gorn: they had been working on various experiments towards a universal code (Nofre, Priestley, and Alberts 2014, p. 52). It was in this atmosphere of desire for real universality (FORTRAN, for example, was very strongly tied to the IBM 704 in its initial incarnations) that the first seeds were planted in 1955 for a truly machine-independent, highly-abstract, algorithmically-oriented language. This language became very successful and influential and played a particularly important role in the story of programming language semantics.

2.3 ALGOL

The programming language known most commonly as 'ALGOL'[16] (formed from 'algorithmic language') is more correctly a family of programming languages, consisting of ALGOL 68, ALGOL 60, and ALGOL 58, as well as many variants of those languages. All of these languages have some relevance to the history of semantics, but ALGOL 60 is particularly important due to its use as a model language for many semantic descriptions. This is largely due to its prominence as the language on which the field of programming language studies was founded; Priestley (2011, p. 229) argues that "the ALGOL 60 report in particular [was] a paradigmatic achievement". Other more intrinsic properties of ALGOL 60, such as its favourable size to expressive power ratio, also contributed to ALGOL 60's success as a tool for demonstration of semantics. Consideration of these features and ALGOL 60's role in the history of semantics begins in this section with the origins of ALGOL 58; ALGOL 68 is discussed in Section 7.1. The history of ALGOL presented here is only an overview; further details can be found in the *History of Programming Languages* papers of Perlis (1981) and Naur (1981b). The special issue (Vol. 36, No. 4) of *The Annals of the History of Computing*, guest edited by Alberts (2014b), provides some deeper historical reflection.

[16] Various different forms of capitalisation were used in various publications. The present work will use all capitals unless quoting a source.

In the middle of the 1950s, following the initial release of FORTRAN, desire grew up amongst both Europeans and Americans for a truly universal programming language—FORTRAN having been designed specifically for the IBM 704 and marketed as a product of that machine. The ACM, under the presidency of John W. Carr, mentioned earlier as an advocate of universal programming languages, set up a committee in 1957 to study the possibility of such a language. This group included academic, governmental, and industrial representatives; amongst the members were John McCarthy, Alan Perlis, and Backus (Perlis 1981, p. 77). The language, obviously, could not be an order code for a particular machine; and so the terms 'algebraic' and 'algorithmic' were used to describe the desired language, reflecting the greater abstraction needed.

In Germany, the *Gesellschaft für Angewandte Mathematik und Mechanik*[17] (GAMM) had also been working on the creation of a universal algorithmic language. In Europe, a continent still deeply divided along political lines and with many spoken languages, there was a desire to ease the communication of programs: although there were many embryonic high-level languages under development, none had successfully become a standard. Another reason for the Europeans to develop a new universal language was to resist the dominance of FORTRAN as an IBM product. Following a conference in Darmstadt in 1955, a subcommittee of GAMM was formed to address this need (Rutishauser, quoted in Naur 1981b, p. 93). So when in 1958 an ACM subcommittee was also formed for the same purpose, GAMM sent a delegate—Friedrich Bauer—with a proposal that a joint algorithmic language be created. A joint ACM/GAMM meeting was held in Zurich at the end of May 1958 and it was agreed that an 'International Algebraic Language' should be formulated. There was less enthusiasm for the new language amongst the American cohort, due to the greater level of FORTRAN acceptance in their country, but they recognised the language would be of value even if it were only used in Europe (Perlis 1981, p. 78).

A 'Preliminary Report' on the language discussed at the meeting, authored by Perlis and Samelson (1958), presented the first look at the 'International Algebraic Language' (IAL). It was anticipated that there would be multiple varieties of the language right from the start, with the report identifying three levels: the 'Reference Language', which would be the defining standard; the 'Publication Language', to be used for the dissemination of programs, which would be almost identical to

[17] Society for Applied Mathematics and Mechanics

the Reference but would allow regional variations in character sets;[18] and a series of 'Hardware Representations'. This approach would allow for variations in local implementation such as built-in operations or word length while still maintaining a universally-agreed standard that would make arguments over local style irrelevant.

The Preliminary Report described the language informally, primarily with the use of examples. The term 'syntax' is used only once and 'semantics' is not used at all; 'meaning' appears in a few places but does not connote anything rigorously defined. For example, arithmetic expressions are described as follows: "The operators $+$, $-$, $\times$, / appearing above have the conventional meaning" (Perlis and Samelson 1958, p. 12). The language was thus intended to be understood intuitively: the purpose of the document was simply to introduce the core concepts of the language. Certain features, such as lists, trees, and recursion, were left out of the language because a sufficiently precise definition could not be agreed upon in the time permitted (Perlis 1981, p. 79). This is in keeping with the preliminary nature of the report: the introduction states "It is anticipated that the committee will prepare a more complete description of the language for publication" (Perlis and Samelson 1958, p. 8) .

That description was provided in a paper entitled 'The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference'. Written up by Backus, and presented at the International Conference on Information Processing in June 1959,[19] it is noteworthy for featuring the terms 'syntax' and 'semantics' so prominently. Previous language description documents such as the Preliminary Report, or the FORTRAN Programmer's Reference Manual, had not used these terms. The inclusion of such terminology, common from this point onwards when discussing languages, is likely due to the shift in thinking about programming systems towards a language metaphor, described by Nofre, Priestley, and Alberts (2014). The concepts of syntax and semantics, taken from linguistics and the study of formal languages, would seem obvious considerations when regarding a 'language'; and this may provide some explanation for their introduction.[20]

[18] An attendee at the Zurich meeting, Joseph Wegstein (in Naur 1981b, Appendix 4), later noted wryly that this particular requirement was introduced following "two full days of sparring" after which "the meeting came to a complete deadlock, with one European member pounding on the table and declaring 'No! I will never use a period for a decimal point.' ".

[19] This conference is itself important as its success allowed the creation of IFIP.

[20] Not everyone liked this anthropomorphising of coding systems, however; Stoy (1977, p. 9) argued that as computing languages are carefully constructed to avoid the ambiguities and complexities of human language:

> This may indicate that 'language' is the wrong word to use for objects of our study, and that perhaps the word 'notation' would give a more accurate impression of what

The language described in Backus' paper is the 'Reference Language' and the description was given in two parts: first, a longer informal description, and then a shorter formal section. The first part uses natural language and examples to describe the syntax and semantics of IAL, although there is clearly an effort to be both more specific with the descriptions and more comprehensive in coverage when compared to the Preliminary Report. Backus (1959, p. 12) noted specifically in this paper that the informal description is sufficient in achieving one goal of the Zurich conference: "to provide a means of communicating numerical methods and other procedures between people".

It is the second part which is of most interest to the current story: presented there is a formal description of "the set of legal IAL programs", also referred to as the language's syntax (Backus 1959, p. 13). The notation used here later became known as Backus Normal Form or BNF [21] and is a set of rules for generating the allowable texts of programs, and no other texts. These rules are context-free in the sense that there is no reliance on the meaning of the symbols for determining validity, only the identity of the symbols and the manner of their combination. Lucas (1978, p. 6), in a brief history of formalism in programming languages, emphasised the importance of BNF not just to language description, but also the study of formal languages in general: the set of texts definable by BNF overlaps nearly perfectly with the class of context-free Chomsky grammars.

The motivation for this syntactic system was to ease the task of the implementer of the language, the writer of the translating program which will convert programs written in IAL to those executable on the machine. If true universality is to be achieved, argued Backus (1959, p. 13), "there must exist a precise description of those sequences of symbols which constitute legal IAL programs. Otherwise it will often be the case that a program which is legal and translatable for one translating program will not be so with respect to another".

The formalised syntax went some way to accomplishing the goal of universal implementability, but was not on its own sufficient. The paper went on to explain that some formalising of semantics was necessary as well: "for every legal program there

---

we are about: we do not normally talk about 'the language of tensors', or 'Leibniz's language for the integral calculus'. But we are bedevilled by over-inflated jargon on computing (usually implying unwarranted anthropomorphism), and we must learn to live with it.

[21] See Knuth (1964a) for a discussion of this name.

must be a precise description of its 'meaning', the process or transformation which it describes, if any. Otherwise the machine language programs obtained by two translating programs from a single IAL program may behave differently in one or more crucial respects" (Backus 1959, p. 13). According to Backus, no fully formalised description of a language that met both of those criteria had been published. Indeed, IAL did not either: despite the recognised need for formal semantics, it was not included in the paper. A note at the end indicates that there was an intention to write something and even include it in a subsequent publication, but such a description never emerged. This may be due to the flurry of suggestions for improvement that began to appear as soon as the Preliminary Zurich Report was published at the end of 1958. It was obvious a new version of the language was required.

There were two main places in which these discussions took place: The *Communications of the ACM* (*CACM*), and a new venue, the *ALGOL Bulletin*. This latter was set up by Peter Naur, a Danish astronomer and mathematician, who was frustrated with the lack of a decent venue in which to have long-form written discussions of the language (Naur 1981b, p. 98). This was particularly relevant given the rapidly growing enthusiasm for the new language in Europe after the publication of the Preliminary Report (Perlis and Samelson 1958) in the *CACM* and a later entry in *Numerische Mathematik* (Perlis and Samelson 1959).[22] Both the *CACM* and *ALGOL Bulletin* soon contained many proposals for changing and improving the language from a number of various diverse interested parties. Contributors included the original authors of IAL and those that later went on to design the new language, but also various others who had roles to play in semantics: Edsger Dijkstra, Viktor Kudielka and Heinz Zemanek (both later of the IBM Laboratory in Vienna; see Chapter 5), and Peter Landin (see Section 3.2). By June 1959—the publication of Backus' report—it was already clear that the best solution was to develop a new language based on the principles of IAL (Naur 1981b, p. 96).

In any case, IAL had never been intended as a finished language in its own right, but rather a framework around which a further language could be built (Perlis 1981, p. 82). Various organisations began adapting IAL for their own purposes, creating "dialects" such as NELIAC and JOVIAL (Priestley 2011, p. 207). IBM's usergroup SHARE created an IAL subcommittee and began work on an implementation at the beginning of 1959, but this failed to catch on properly due to the enormous effort

[22]Interesting differences in the presentation and terminology of these two reports is discussed by Durnova and Alberts (2014).

already being expended to implement FORTRAN. There was concern that any implementation of IAL would serve only to undermine the growing use and utility of FORTRAN, something neither IBM nor SHARE wanted to see (Perlis 1981, p. 83). According to Nofre (2010), arguments also ensued about whether the universality project should really be promoting one single programming language, or whether a diversity of inter-translatable languages and a translation system was a better way to achieve the goal. With the US contingent more firmly in the latter camp, this paved the way for a drop in enthusiasm and uptake in America for the new algorithmic languages.

Although IAL was not so far succeeding terribly well at achieving the goal of a widely-implemented universal language, it was seeing a lot of use in another area: the publication of algorithms. The *CACM* saw a lot of discussion of algorithms and in 1960 opened a section specifically for this purpose. The vast majority of these were published in some variant of IAL or one of its successors.

At the same time, a number of discussions were ongoing in the same forums about the necessary improvements to the language, and these were taken on by the ACM and GAMM subcommittees. A preliminary meeting was held by GAMM in November 1959 in Paris to make the decision about who should attend the next meeting, at which a new version of the language would be developed; the ACM committee also chose delegates around the same time. The defining meeting for the new language was held in Paris, in January 1960, and was hosted by IBM World Trade Europe. In attendance were a number of the participants who had been at the Zurich meeting, including Backus, Katz, Perlis, Wegstein, Bauer, Rutishauser, and Samelson. Also invited were some new delegates, including Naur, Green (of IBM), Turanski (of Remington Rand),[23] Woodger (at the National Physical Laboratory in the UK), Vauquois (of the Institut Fourier in France); and some others whose names will become important in the present story of semantics: John McCarthy (at that time at MIT), and Adriaan van Wijngaarden (at Mathematisch Centrum in the Netherlands). The atmosphere at this short meeting, held 11–16 January, was electric: Perlis (1981, p. 88) later described it as "exhausting, interminable, and exhilarating", painting a picture of a dedicated and competent group working together to create something marvellous.

Some photographs from the event can be seen in Figures 2.5, 2.6, and 2.7. These

[23]Sadly killed in a car accident a few days before the meeting; the subsequent Report is dedicated to his memory.

Figure 2.5: The attendees of the January 1960 ALGOL meeting (seated clockwise from left): Julien Green, Klaus Samelson, Charlie Katz, Peter Naur, Mike Woodger, Joe Wegstein, Bernard Vauquois, Adriaan van Wijngaarden, Alan Perlis, Heinz Rutishauser, Fritz Bauer, and John Backus. Photograph was taken by John McCarthy.

Figure 2.6: John Backus (left) and Fritz Bauer discuss the syntax of labels.

photographs are taken from a presentation by Naur (1981b, pp. 148–154) about the history of ALGOL. Naur also commented that the vote shown in the first picture was "the one and only unanimous vote of the meeting. And the issue of this vote was: 'Do you agree to have your picture taken?' ".

It was decided that rather than attempting to extend IAL, a new language should be built from the ground up, with IAL's principles in mind. This language would be

Figure 2.7: From left: Fritz Bauer, Adriaan van Wijngaarden, Peter Naur, Klaus Samelson.

called ALGOL 60. The shift to the name 'ALGOL' from 'IAL' occurred in 1959; this can be seen in the different names of the Reports presenting the language: Perlis and Samelson (1958) use the description 'International Algebraic Language'; but by 1959 the name 'ALGOL' is used. Perlis (1981, p. 79) recalls that 'ALGOL' was originally proposed at the Zurich meeting but was rejected "perhaps because it did not emphasize the international (!)[24] effort involved". However as work progressed towards a new version of the language, 'ALGOL' came back into parlance, with 'IAL' derided as "'unspeakable' and pompous". The new name was then applied retrospectively to IAL, terming it 'ALGOL 58', while the new language was to be called 'ALGOL 60'.

The output of the Paris meeting was the 'Report on the algorithmic language ALGOL 60' and was edited by Naur, but had the full committee's names on it (Backus et al. 1960). Formalism was again used for the syntax, although Naur thought this would not be an automatic decision: he believed he would have to persuade the committee, having realised the importance of formalism after seeing the large number of problems caused by the inconsistencies and omissions from the 1959 IAL definition. Naur (1981b, p. 99) wrote:

> All this indicated to me the crying need for integrating the formulation of a precise and complete language description into the actual language

[24] Original sarcasm.

> development process. I was led to the conviction that the formulation of a clear and complete description was more important than any particular characteristic of the language.[25]

To aid in his argument that formalism was not only necessary, but also feasible, Naur prepared some fragments of a potential language and brought these to the Paris meeting (Naur 1981b, p. 100). They were composed from the suggestions which had been published in the *ALGOL Bulletin* and other places, rather than being Naur's own inventions (Samelson, in Naur 1981b, Appendix 7). This served as a starting point for the new language, making Naur the obvious editor of the new Report.[26]

The Report was indeed written using BNF for the description of syntax. This use of formality was described by some authors in a way that seemed to draw from notions of formalised syntax for logic: very careful definition of terms, and composition of more complicated concepts from basic ones. The report indicates that (in contrast to IAL, which used more arbitrary identifiers) "the symbols used for distinguishing the metalinguistic variables [...] have been chosen to be words describing approximately the nature of the corresponding variable" (Backus et al. 1960, p. 109). This was intended to provide an "immediate link between syntax and semantics" (Naur 1981b, p. 115).

Duncan, in a rare serious point made in a speech at the end of the *Formal Language Description Languages* conference, commented that this also served to establish a tie between the formal and informal parts of the description. A word like 'label' introduced as a formal metavariable within the BNF could then be used informally within the semantics description; variations on it like 'labeled' and 'labeling' could also be used. This point illustrates another crucial role for formal syntax systems: providing a reference point for semantics, delineating the particular parts of a language so that each can then be associated with a particular meaning.

[25] It is interesting to compare this very positive attitude towards formalism with the very negative position Naur took later. In Naur (1981a)—published the same year as the source of the above quotation—Naur tore into a formal description of ALGOL 60 and even discredited large parts of the idea of formalism in general. This negative reaction towards formal methods is discussed further in Section 8.1.

[26] This version of the story, however, was challenged by Bauer (in Naur 1981b, Appendix 5), who did not think there was any real opposition to using BNF to formally define the syntax of ALGOL 60, and described Naur as "running into open doors". Furthermore, Bauer was concerned by the central role Naur was assuming, warning him he would become "the pope", a role in which he would have preferred to see Rutishauser. Ironically, for a very long time Bauer himself was known as *Der Papst* (the Pope) in Germany for his central role to informatics in that country. In the UK, he was known—similarly affectionately—as 'Uncle Fritz'.

The recursiveness of the syntax definition allowed a powerful and expressive language: because each statement could be a compound statement or a block, nesting was easy to achieve and allowed an arbitrary complexity of program to be delineated relatively simply (Priestley 2011, § 8.6). The semantics, however, was not formalised,[27] although the presentation did show a good deal of care was taken: a manner reminiscent of a mathematical explanation was used.[28] It is worth mentioning one particular aspect of the Reports' approach, however, which is the so-called 'copy rule'. The idea is fundamentally simple and had been used by mathematicians for decades in any situation involving bound variables: copy the identifiers from their various locations into the target phrase and, if there is an identical name found, simply rename one of the variables. This is equivalent to the notion of α-reduction in lambda calculus. The intuitiveness of the idea belies the complexity underlying it and thus, while some of the formalised semantic approaches discussed in this story apply this principle, many avoid it by using other methods.

Response to the informal approach to semantics was mixed: at a *Working Conference on Mechanical Language Structures* (WCMLS) held in Princeton in August 1963, Gorn (1964, p. 134) wrote:

> It seems to me that much of the semantic intent in ALGOL could have been specified in other than natural language and that the failure to do so caused confusion.

On the other hand, Perlis (1981, p. 89) (with the benefit of hindsight) argued that the incompleteness was something of a virtue:

> The authors wisely under-described the semantics, choosing an incomplete description, possibly leading to ambiguity, in preference to inconsistency. They can be forgiven for not being precise. To do so would have required agreement on a semantic model in which machine and implementation independence would have been preserved. At that time, no-one knew how to define a semantic model with those properties. As a consequence, they underestimated the difficulties in getting efficient implementations of the for statement and evaluation of call-by-name parameters.

[27] Bauer (in Landin 1966c, p. 292) noted that during the "chaos" of trying to fix the semantics of procedures, there was a group advocating the use of lambda calculus in this context. This did not happen, but it is interesting to see that there was already in 1960 a school of researchers thinking about using lambda calculus in programming languages.

[28] For more discussion of the way in which semantics was presented in the ALGOL 60 Reports, see Astarte and Jones (2018, pp. 81–4).

Defining method was not the only area in which there was disagreement over ALGOL 60. One of the biggest conflicts in the author group was over what Samelson (in Naur 1981b, Appendix 7) termed 'flexibility vs. power'. On one side were those who wanted a language with built-in clarity and rigour (and corresponding language restrictions); this was largely the GAMM contingent: Bauer, Rutishauser, and Samelson. The other argument was for a more complex language that could be better exploited by ingenious programmers, represented by Naur, Perlis, and van Wijngaarden. This divide continued throughout the history of ALGOL, and flared up again during the development of ALGOL 68 (see Section 7.1).

A new and unique property of ALGOL was its definition by document, in contrast to earlier languages like FORTRAN for which the compiler was the ultimate reference. Zemanek, in an interview with Aspray (1987, p. 44) explained:

> In other words, when you had a problem, in the case of ALGOL, you went to the document and looked at what the definition says, and that you have to fulfil with the compiler. In the case of FORTRAN, when you had an argument, you would have to look at what the compiler answered and the compiler was right. Everything you wrote down had to conform with what the compiler did.

Using a document as the arbiter, however, could and did cause problems: being written in natural language, there remained ambiguities and omissions.

The ALGOL 60 Report, then, was not the ultimate language standard. Both Europeans and Americans formed groups to continue the maintenance of the language, but this led to fears of two dialects of the language emerging and breaking the all-important universality goal. A meeting was held in Rome in April of 1962 at which a Revised Report was put together to correct errors and ambiguities in the initial Report, which were inevitably present given the informality of the definition approach. The publication of this Report (Backus et al. 1963b) in *CACM* came in 1963, and again the changes were in part inspired by those submitted to the *ALGOL Bulletin*. Concurrent to the building interest in ALGOL maintenance, IFIP had founded its second Technical Committee, commonly known as TC-2, its purview to be programming languages (the first Technical Committee was concerned with terminology). Given the desire to maintain a properly international group in charge of ALGOL, at the Rome meeting it was decided to hand control of the language to IFIP. The first Working Group of TC-2 was WG 2.1, on 'Algorithmic Languages', with membership being composed primarily of those present at the Rome meeting. This included the

majority of the original ALGOL writers, with a few new inclusions—notably Peter Landin (see Section 3.2). This new body overseeing the language led Naur to decide the *ALGOL Bulletin* was no longer necessary, and cancel its distribution (Priestley 2011, p. 227–228).

The 1962 Revised Report did not mean the language was without any controversial elements. One particular sticking point was the **own** declaration for variables, which would cause the value of that variable to be maintained should the block be entered another time. Generally regarded as an unnecessary complication, many later semantic descriptions of ALGOL 60 chose to simply ignore it. McCarthy recalled in 2004[29] that he had a view of blocks in programming languages as a way to potentially combine programs, possibly even at the machine level, changing names of variables as appropriate.[30] However, the European view was that a variable defined in a block should lose its value when the block is left so the memory could be freed. The inclusion of **own** variables was an attempt at a compromise.

Another tricky element was the inclusion of explicit recursion, which was seen by the Americans as sneaked in by the European contingent at the last minute—the crucial sentence "Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure" (Backus et al. 1963b, § 5.4.3) was added rather late in the process. However, this narrative was challenged by van den Hove (2014), who argued that recursion is an inherent part of the language and its structural definition, and even without that sentence, an implementation which allowed recursion would have to be considered correct. Further complications in ALGOL 60 were caused by the various parameter mechanisms, which provided different ways for the evaluation of parameters at procedure calling time.[31]

Despite these difficulties, ALGOL 60 became and remained a very highly-regarded language. Tony Hoare (1973, p. 27) wrote in a paper 'Hints on programming language design': "Here is a language so far ahead of its time that it was not only an improvement on its predecessors, but also on nearly all its successors". Dijkstra

[29] At the 20th *Mathematical Foundations of Programming Semantics* (MFPS) conference; a panel was held on 'Reminiscences on Programming Languages and their Semantics' featuring Cliff Jones, John McCarthy, John Reynolds, and Dana Scott.

[30] Strachey later acknowledged (in Milne and Strachey 1974, p. 25) that this concept of scope was one of the most important contributions of ALGOL 60, and that the general feeling was that McCarthy was the main force behind this.

[31] For a longer discussion of the relevance of these quirks to semantic description, see Astarte and Jones (2018, pp. 78–81).

(2001a, pp. 5–6) noted that the publication of ALGOL 60 was "an event that [his] colleague F.E.J. Kruseman Aretz for good reasons has chosen to mark the birth of Computing Science. The reasons were good because ALGOL 60 introduced a handful of deep novelties, all of which have withstood the test of time." These were its formal definition, the block structure and recursion, Boolean values as base types, and machine independence.

ALGOL was always intended to be continually improved and updated: although calling it a "rounded work of art" already at the time of its creation, Perlis (1981, pp. 88, 91) joked "It was a noble **begin** but never intended to be a satisfactory **end**". Naur (1981b, p. 117) noted that the language also prompted a great deal of research in formal languages, but felt that this distracted from focus on the language itself, to its detriment. Dijkstra (2001a, p. 5), however, saw this as an advantage, writing

> More than anything else, ALGOL 60's formal definition has made language implementation a topic worthy of academic attention, and this was a crucial factor in turning automatic computing into a viable scientific discipline.

Kurt Walk (2002, p. 78), of the Vienna Laboratory, agreed, writing that in the early 1960s, his group understood the importance of programming languages:

> The programming language here was the sole mediator between humans, their problems, and the computer. This made programming languages and their compilers the central subjects of interest and research in (the software section of) information technology. Language design and language definition methodology were the key disciplines.

Priestley (2011, § 9.3) agrees with this perspective, arguing that the publication of ALGOL as a machine-independent language meant its meaning had to be described in much more abstract terms, paving the way for deeper study of the area of abstract programming language description and many formalisms thereof. Alberts (2014b, p. 3) argues that the introduction of ALGOL established a new "research agenda".

The machine-independence property of ALGOL was particularly important in Europe, with its greater proliferation of computers than America—and it had the advantage of not being an IBM product. Indeed, IBM, through SHARE, actively avoided providing support for ALGOL. Although ALGOL is generally considered as

failing to break through into industrial success, as it had to compete with the IBM-backed FORTRAN, it had a huge impact on theoretical computing. New techniques were required for the implementation of ALGOL, given its lack of reference to machine in the publication, and its new features such as recursion; this led to concepts now central in computing, such as the stack concept and associated mechanisms such as Dijkstra's display. This was a crucial factor in ALGOL's impact, as Mahoney (1988, p. 112) notes: "ALGOL might have remained a laboratory language had it and its offspring not become the vehicles of structured programming, a movement addressed directly to the problems of programming as a form of production."

ALGOL 60 may not have been widely implemented across the world, but a number of very influential computer scientists started out their careers providing an implementation for one commercial enterprise or another. Dijkstra and Jaap Zonneveld wrote a compiler for the Electrologica[32] X1 before 1960 was out: no report was published right away, but the full code (and associated commentary), can be found in Kruseman Aretz (2003).

Another commercial compiler was created by Brian Randell and Lawford Russell at English Electric. Randell had attended a school on ALGOL 60 in Brighton in 1961; he had been considering writing a new high-level programming language for English Electric's forthcoming KDF9 computer but decided to implement ALGOL 60 instead after Brighton (Randell 2010). This was done following a visit to Dijkstra to learn about the X1 compiler. Randell (2016) explained that his report on the trip was rather influential: "we wrote up those discussions in a very lengthy technical report (Randell and Russell 1962), and for some years Dijkstra didn't document his compiler, and tended to refuse other requests to go and interview him, by saying 'No, apply to Brian Randell, for a copy of the trip report.' " The authors also wrote about their implementation in a book (Randell and Russell 1964), which was long regarded as *the* ALGOL 60 implementation book.

Another attendee at the Brighton school was Tony Hoare, then working for Elliott Brothers. According to Hoare, the tutors were illustrious indeed: Dijkstra, Landin and Naur. Hoare, too, had been tasked with developing a high-level language for Elliott's new 503 computer, but decided instead to implement ALGOL 60.

As already mentioned, the publication of ALGOL 60 was instrumental in the foundation of the field of study of formal languages. ALGOL played an important role

[32]Electrologica was a company spun out of the Mathematisch Centrum in Amsterdam to build computers designed at that institution.

in the development of formal semantic description techniques in particular: most of the people mentioned in the present story used either ALGOL or an ALGOL-like smaller language to illustrate their approaches to semantics. More detail is given in the respective sections, but the list includes McCarthy, Rod Burstall, Jacobus de Bakker, van Wijngaarden, Landin, the Vienna Laboratory, and the Oxford Programming Research Group. Reynolds (1981, p. 3), in a paper entitled 'Essence of Algol' also noted the importance of ALGOL and its descriptions, writing "Among programming languages, Algol 60 has been uniquely influential in the theory and practice of language design. It has inspired a variety of models which have in turn inspired a multitude of languages."[33]

The ALGOL story does not end with the 1962 Revised Report. Working Group 2.1 continued to work on the language, despite the somewhat hopeful claim early on in their founding that "a properly defined language does not require maintenance" (Utman 1962b, p. 2). Work on a new version of ALGOL continued throughout the 1960s: initial plans called for a 1965 and a 1970 version to be called ALGOL x and ALGOL y respectively. Interestingly, WG 2.1 minutes show a unanimous consensus that any new versions should be given a full formal definition—but there was no such agreement on which method should be used (Utman 1964a). Ultimately, the proposals for new versions resulted in the highly controversial ALGOL 68 language, whose complexity in both content and mode of presentation caused a deep rift in WG 2.1 and the computing community at large. That story is considered later (in Section 7.1); for now, let us return to the end of the 1950s and begin by looking at some early advocates for formal semantics.

[33]Page references are made to the most readily-available version of this paper, which is hosted on the CMU website.

CHAPTER 3

Before the storm: early semantics pioneers

In this chapter, the work of some researchers who worked in the field of formal language descriptions is discussed. A brief background on each practitioner is laid out, especially as it pertains to their relevant work, and an outline of their contributions is made. Of particular interest are the motivations of each researcher for embarking on work in this area. Section 3.1 covers John McCarthy; Section 3.2 discusses the work of Peter J. Landin; and Section 3.3 explores the contributions of Adriaan van Wijngaarden. See Figure 3.1 for a timeline of this period.

3.1 John McCarthy

One of the first people to work on formalising concepts in programming in a more abstract way was John McCarthy (1927–2011). Born in Boston, Massachusetts, McCarthy attended the California Institute of Technology, being awarded a BS in Mathematics in 1948, before moving to Princeton to study a PhD in Mathematics, graduating in 1951 (biography in McCarthy 1981). He then worked for a short time at Dartmouth College, organising a famous conference in artificial intelligence in 1956, and then spent six years at the Massachusetts Institute of Technology. In 1962 he moved to Stanford where he remained until his retirement in 2000. McCarthy was well known for his contributions to AI and early concepts in time-sharing sys-

Figure 3.1: A summary of some key events relating to early semantics pioneers.

Feb 1946	Mathematisch Centrum is founded and van Wijngaarden becomes the head of computing
1956–1958	Creation of LISP ideas at Dartmouth and MIT
Late 1950s	Pragnell's logic and maths reading groups attract many early computer scientists, including Burstall and Landin
1958	Landin learns about LISP and becomes interested in extending McCarthy's work
Jan 1960	Landin begins working for Strachey's consulting business
Jan 1960	Van Wijngaarden joins the Paris committee on ALGOL 60, pushing a 'generalist' approach
Mar 1960	Release of first LISP implementation, initially created by Steve Russell
Apr 1960	Publication of McCarthy's first paper on LISP, using a deliberately abstract notation
Mar 1962	Van Wijngaarden presents a symbol manipulation approach to language description
Aug 1962	McCarthy presents an abstract approach to language description
Jan 1964	Landin publishes his lambda calculus basis for language translation with interpretation by an abstract SECD machine
Sep 1964	At the *Formal Language Description Languages* conference, McCarthy shows his ideas applied to a subset of ALGOL 60, van Wijngaarden elaborates on his approach, and Landin presents a formal definition of ALGOL 60
Jan 1966	Landin applies his approach to language design
Dec 1967	McCarthy and Painter publish a piece on verifying compiler correctness using semantic techniques
1969	Burstall and Landin work on an algebraic approach to program description and proof

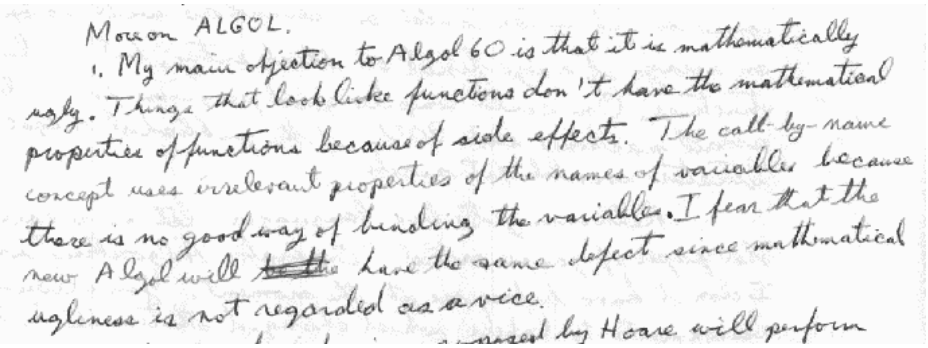


Figure 3.2: John McCarthy explains his main concern with ALGOL 60 to Andrei Ershov (McCarthy 1965, p. 7).

tems, but it is his interest in abstraction in programming languages that is relevant here—both in the creation of a language, LISP, and in formal description.

3.1.1 LISP: abstraction in programming languages

As has already been mentioned, McCarthy was part of the team that developed ALGOL 60.[1] Despite his involvement, McCarthy still had a number of problems with the language as it emerged. Chief among these was that ALGOL was not sufficiently mathematical. McCarthy explained this in a letter to the Soviet computer scientist Andrei Ershov in 1965 (see Figure 3.2 for the handwritten version):

> My main objection to Algol 60 is that it is mathematically ugly. Things that look like functions don't have the mathematical properties of functions because of side effects. The call-by-name concept uses irrelevant properties of the names of variables because there is no good way of

[1] See Section 2.3.

> binding the variables.[2] I fear that the new Algol will have the same defect since mathematical ugliness is not regarded as a vice.
> (McCarthy 1965, p. 7)

ALGOL was not McCarthy's first foray into the creation of programming languages, however: by the time of the Paris 1960 meeting, he had already published the first material on the language LISP.[3] This language, whose name stood for LISt Processing language, comprised not only a notation for programs, but also a 'programming system' which incorporated a very early form of garbage collection (McCarthy 1960). The language was characterised by its use of lists as the main data representation structure and conditional expressions used to define recursive functions. The truly crucial novelty in LISP, though, was its application as a *symbolic* language, rather than numerical like, say, FORTRAN. That is to say, LISP programs were intended to perform some kinds of action based purely on the symbols involved rather than any notion of their value; and typically, LISP programs were not primarily for the calculation of numerical results.

LISP was developed as part of McCarthy's early research into AI, and the necessity of finding a way to describe the action of a computer attempting to simulate intelligent activity as discussed at the Dartmouth Conference. The inspiration came from Newell, Simon, and Shaw's Information Processing Language (IPL), also based on lists, which had been developed following the desire for a language to express notions of human thought and complex system decisions (Priestley 2011, p. 221). McCarthy (1981, pp. 185–6) recalled later than he and Marvin Minsky had asked their research laboratory head for money for an AI project, which they were duly given along with some graduate students. With this extra staffing, it seemed natural to develop a programming language at the same time. Some of the ideas which later appeared in LISP can be seen in McCarthy (1959), 'Programs with Common Sense', which described an 'advice taker' program formulating declarations about the world and making decisions based on those; the declarations and knowledge are represented within the machine as lists and operations work on these.

The LISP programming system was ultimately developed for an IBM 704 machine, like FORTRAN before it, although unlike FORTRAN, it was not tied to the machine as a product. LISP included lambda notation for writing functions, later a

[2] Note that this is contrary to the position John Reynolds would later take: Reynolds (1981) rather argued that call-by-name was the core mechanism of lambda calculus and also a critical part of the 'essence' of ALGOL. For more discussion, see Sections 3.2 and 6.6.

[3] Later styled also as 'Lisp'.

key concept in denotational semantics (see Chapter 6), but no more of Church's calculus, such as the rewrite rules. McCarthy (1981, p. 176) admitted later that he didn't understand the majority of lambda calculus, and in any case, using conditional expressions instead of higher-order functionals was easier to implement on a computer. Daylight (2012, pp. 27–8) concurred with this perspective, explaining that the desire to use lambda calculus concepts was secondary to McCarthy's goal of finding a way to describe recursion. In fact a later commentator on the use of lambda notation in LISP described the binding rules used as "a bug", following as they did dynamic rather than lexical scoping (Moreau 1998, pp. 233–4). Mahoney and Haigh (2011, p. 143) suggested that lambda calculus had been somewhat abandoned since its initial goal (providing a type-free solution to Russell's Paradox) was unsuccessful and described McCarthy as "reviving" the concepts. However, McCarthy's tone when introducing the notation does not suggest it to be obscure, and peers of McCarthy such as Landin and Scott described learning lambda calculus during their university educations.

In later reflection, McCarthy made much of LISP as a language of inherent simplicity: it has only a small number of core concepts, plus some compositional tools, and yet has remarkable power (McCarthy 1980). Landin (1966b, p. 160) also praised the "logical properties lying behind the notation" of LISP, believing these to be its most important contribution. He commented specifically on the way that equivalence of LISP program pieces could be determined, and the way that it facilitated the denotation of expressions similar to those used in mathematical and logical systems.

The first publication of the LISP programming system was in a *CACM* article entitled 'Recursive functions of symbolic expressions and their computation by machine, Part I' (McCarthy 1960) and the opening clearly links the resultant language with its development for the 'advice taker'. The *CACM* paper was not intended to be the definitive standard for the language but rather to show how LISP could be used both for programming computers and recursive function theory. McCarthy used a deliberately abstract notation to avoid machine dependence, which represented data and instructions as *symbolic expressions* and *functions*, or S-expressions and S-functions. A more FORTRAN-like syntax was planned but never ultimately emerged, as users learned instead to embrace the S-expression style (McCarthy 1981, p. 179).

The core of the LISP programming system was the pair of universal S-functions *apply* and *eval* which together provided a way of manipulating S-expressions and S-functions. *apply* was used to apply functions and is not itself terribly exciting; more

interesting was the *eval* function which evaluates expressions given a list of prior assignments. *apply* simply creates an expression form of the function call which can then be passed to *eval*. The explanation given for *eval* is as follows:

> *eval*$[e; a]$ has two arguments, an expression e to be evaluated, and a list of pairs a. The first item of each pair is an atomic symbol, and the second is the expression for which the symbol stands.
> (McCarthy 1960, p. 18)

There is an important point to be made here about this "list of pairs a". It essentially represented the storage of the computer in an abstract way by providing an association of variables with values. This is a crucial concept in formal semantics; it is most usually referred to as 'state' (indeed, this was a term McCarthy himself used in subsequent papers). Precisely how the state is crafted and what components it contains varies hugely across semantic styles, but the key elements are the association of variables and values, and universal accessibility. In LISP, a was passed around between functions, in the style now popular in functional programming, rather than being automatically available to every call, but this essentially captured the same behaviour as a globally-accessible metavariable.

McCarthy added a remark about this list a: its use could be avoided by careful substitution of arguments for variables in the application of S-functions, essentially mirroring the 'copy rule' used by mathematicians and logicians. This approach was also used in the description of procedures in the ALGOL 60 Reports. McCarthy did not recommend this style, however, writing "unfortunately, difficulties involving collisions of bound variables arise, but they are avoided by using the list a" (McCarthy 1960, p. 18).

An interesting point about the *eval* function is that it was not originally intended for use in implementation. McCarthy commented at the MFPS panel discussion (Jones et al. 2004) that *eval*'s original purpose was as a universal function to be equivalent to a Turing Machine and was included to impress logicians. *eval* required notation that allowed the representation of LISP programs as LISP data, for which purpose the S-expression style was created. It was never McCarthy's intention that this would be the final form of the language; indeed, as mentioned earlier, a more FORTRAN-like syntax was planned. However, a technician working with McCarthy, Steve R. Russell, noticed that *eval* could be encoded as a machine program and then used as a LISP interpreter (McCarthy 1981, p. 179). Meyer (2011) later wrote how

impressive it was that the interpreter for LISP, written in LISP, was so compact as to fill only half a page. McCarthy, however, was not satisfied with only presenting LISP in one style, and gave a few different forms of the same kind of expression in the 1960 paper, demonstrating his interest in problems of representation in programming languages.

3.1.2 Mathematising computing

Another area of interest for McCarthy was the linking of computation with mathematics to bring a rigorous foundation to computing,[4] as seen in his intention to make LISP useful for those who studied recursive function theory. McCarthy (1963) explained that in mathematically-based sciences, important properties were built from basic assumptions; for example, Newton's laws of motion allow the deduction of Kepler's laws of planetary bodies. The historian of science Mike Mahoney (2002) wrote about this, reporting that McCarthy was interested in considering what similarly basic notions existed in computing, and to what deductions they could lead. A further reason to build a link between computation and mathematical logic was to increase the use of logic in programming. McCarthy's aim was not, however, to prove equivalence of every model of computation; as he clarified in a later discussion:

> What I was really arguing against was the sort of model that says Turing machines and recursive functions and productions systems and ALGOL are all the same thing. They have certain relations, but they are not the same thing.
> (McCarthy 1966, p. 9)

A goal McCarthy often stated was a desire to bring computation within the remit of mathematical proof: In a 1990 interview with Mahoney, McCarthy said he thought a situation would emerge where "no-one would pay money for a computer program until it had been proved to meet its specifications." (McCarthy, quoted in Mahoney 2002). In that same piece, Mahoney (2002, pp. 29–32) also highlighted the link between the scientific nature of computing, and formal semantics of programming languages. As the science seeks to understand the limits of computation and computing, the understanding must also encompass programming languages—and of course models with formal properties are one of the chief tools of science.

[4] This was a prevalent theme of the 1960s; during this time, the computing discourse was dominated by a mathematical view of computation, as contrasted to the scientific view of the previous decade (Eden 2007).

McCarthy's first paper with a clear intention to link mathematics and computing was 'A Basis for a Mathematical Theory of Computation' (McCarthy 1961), first presented at the May 1961 Western Joint Computer Conference[5] and reprinted in 1963. The paper began with a series of open and interesting questions in the area, many of which overlap with the motivations for studying semantics given by various practitioners.[6] These included: a way of showing equivalence of computational processes and methods for transformation between these (for example, from non-computable to computable functions; the reduction or even elimination of the need for debugging, using formal proof techniques instead; and proving the correctness of compilers. S-expressions, as seen in the LISP paper, were re-introduced, as a way to move towards achieving these goals. McCarthy then introduced a new method for proof by induction, called 'recursion induction'; the idea is to recurse over the branches of conditional expressions Finally, McCarthy wrapped up with some desires for union of computation and mathematical logic: "It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last" (McCarthy 1961, p. 42; the conclusion was added in the 1963 reprint). These kinds of mathematical goals continued to permeate all of McCarthy's work.

A later paper, 'Towards a Mathematical Science of Computation' (McCarthy 1963), showed this goal developed a little further. One particular passage near the end mirrors an important goal for semantics: the creation of an appropriate mechanism and system under which correctness and other properties can be stated and proved. McCarthy (1963, p. 6)[7] wrote:

> Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs.

The paper was presented at IFIP's first officially titled[8] International Congress on Information Processing in Munich in 1962, and covered similar concepts to the 1961

[5] This conference series—held twice yearly, once on the West and once the East coast of the USA—was mentioned as the inspiration for IFIP by its founder Isaac Auerbach (1986b, p. 42).

[6] See Section 1.1.

[7] Note: references to this paper are given with page numbers reflecting the later, LaTeX'd republication; this is because this version is much more easily accessible to the reader.

[8] The UNESCO-sponsored International Congress in Information Processing in 1959, at which Backus presented the formalised syntax of IAL, is frequently considered to be the first ICIP, but technically it pre-dated IFIP.

paper, but with the introduction of some new notations and concepts. Critically for the present story, this included formalisation of programming language syntax and semantics. Indeed, McCarthy (1963, p. 19) unequivocally stated "Programming languages must be described syntactically and semantically".

3.1.3 Describing programming languages: syntax

McCarthy noted that work on syntax up to that point had been mostly concerned with defining valid programs—delineating the strings that comprise legal programs. Considerably less attention had been given to syntax's link with semantics, argued McCarthy, explaining that syntax also plays a crucial role in allowing the categorisation and recognition of program components. He called the first kind of syntax 'synthetic' because it describes how to build programs up from the composition of basic parts, and the second 'analytic' because it allows you to take a program apart to see how it is composed. Analytic syntax, McCarthy claimed, is more abstract than synthetic because it only concerns the identification of which parts are involved in a particular construct, and not their concrete representation. This distinction is less obviously useful on small examples like those in McCarthy's paper, but became much more relevant when larger languages began to be considered—especially those, like PL/I,[9] which allowed different syntactic representations of the same construct.

The description of analytic syntax was given in terms of a hypothetical translator, which might be part of an interpreter, or a semantic function attempting to ascertain the meaning of a program part. Whatever the reason, the translator needed to be able to identify and pick out elements from the program in order to correctly translate them. McCarthy also described how to use synthetic syntax with *mk* functions to construct program parts from their components. McCarthy explained an important property of abstract syntax that follows this style: it must be 'regular', which is to say that objects constructed with the synthetic syntax should fit the analytic syntax.

In a later publication, McCarthy (1966, p. 10) explained the connection between his analytic and synthetic syntax, and a concrete formalism like BNF. Analytic syntax tells you that a sum is made of two summands, but nothing further. Synthetic syntax explains that you can combine terms into a sum. BNF goes further and indicates that in order to actually make that sum, one must write one summand followed by a

[9] See Section 5.3.

plus sign, and then the other summand. These different levels of abstraction would be useful for different purposes.

3.1.4 Describing programming languages: semantics

The next section of 'Towards a Mathematical Science of Computation' discussed the semantics of programming languages—the earliest of McCarthy's works to do so. His statement of intent clearly shows that for him, semantics is about understanding meaning:

> The semantic description of the language must tell what the programs mean. The meaning of a program is its effect on the state vector in the case of a machine independent language, and its effect on the contents of memory in the case of a machine language program.
> (McCarthy 1963, p. 19)

The 'state vector' mentioned was defined a little earlier in the paper, and was a development of the idea of the *a* parameter passed to the *eval* function in the LISP paper (McCarthy 1960). Its purpose was to enable the semantics of programming languages to be described in a manner similar to this function:

> In order to regard programs as recursive functions, we shall define the state vector ξ of a program at a given time, to be the set of current assignments of values to the variables of the program. In the case of a machine language program, the state vector is the set of current contents of those registers whose contents change during the course of execution of the program.
> (McCarthy 1963, p. 11)

The phrase 'set of current assignments' is a little confusing, but the essential idea was to provide a one-to-one mapping between variables (or machine registers) and their contents (or values).

The first illustration of the state vector concept was given not in terms of a whole programming language, but instead mathematical recursive functions, flowcharts representing these functions,[10] and fragments of ALGOL-like programs with the

[10] The use of flowcharts as an introductory illustrative tool was later used by Strachey and co. in the explanations of denotational semantics; see Section 6.4. Flowcharts also have a pedigree in early computer science: Goldstine and von Neumann (1947) used flowcharts to prepare routines for coding; Turing (1949) used flowcharts decorated with assertions to aid in the task of proving program correctness; and, later, Floyd (1967a) also used assertions on flowcharts in a more rigorous way.

same meaning. The correspondence between these was established with reference to the state vector, typically called ξ by McCarthy. This required the definition of auxiliary functions for state vector lookup and assignment.

With the concepts of analytic syntax and state vector defined, McCarthy was able to describe the semantics of his "Algolic" mini-language. The meaning was given using a simple function:

$$\xi' = algol(\pi, \xi)$$

This says that the meaning of the program π is the value of the state vector ξ' after execution of the program in initial state ξ; so this general function could be used to give meaning to any program written in the language. This was an extremely important concept, and the core of what was to become known as *operational semantics*. However, as Meyer (2011) noted in his obituary of McCarthy, this also "came remarkably close to denotational semantics"[11], as meaning was described in terms of a function which transforms a state before computation into a state after computation.

Another idea described in the same paper is the notion of a translator function from a high-level language to a machine code, and the formulation of a correctness equation for such a translator. Note, however, that McCarthy was not interested in using translation as a tool for semantics; at the 1963 *Working Conference on Mechanical Language Structures*, he said (in Gorn 1964, p. 134) "To describe semantics by means of a translation rule is an incorrect thing to do. You use a language to describe semantics." The reason he gave for this view is that a translation rule is too vaguely-defined and does not capture the important properties of the language. For that, a formalism is required which then allows mathematical tools to be applied.

Indeed, McCarthy's presentation in the 1962 paper was notably mathematical in its style of presentation. Core concepts were introduced, some simple examples given, and then properties stated and proved. This paper did not include any full language description; its purpose is more like a demonstration of tools. McCarthy (1963, p. 22) stated "We expect to publish elsewhere a recursive description of the meaning of a small subset of ALGOL", and that did follow in 1964, providing a fuller demonstration of the power of the approach. McCarthy also developed the translation correctness notion further in joint work with James Painter later in the decade.

Both of these works will be discussed further, but first it is worth taking the time to take a brief look at the style of formalism employed by McCarthy, compared with

[11] See Chapter 6.

that of John Backus. Backus had worked on FORTRAN, a numerical language created with the aim of addressing practical mathematical applications. His syntax formalism, BNF, is a concretisation of what McCarthy called 'synthetic' syntax: McCarthy (1963, p. 19) commented that it is "better for translating into ALGOL than it is for the more usual problem of translating from ALGOL". In contrast to Backus, McCarthy was more interested in abstract mathematics, formal proof and recursive function theory especially. McCarthy's programming language, LISP, was a symbolic processing language designed to address these concerns—as well as the concepts of artificial intelligence. The syntax approach proposed by McCarthy had a much more abstracted view, and it is clear to see this interest in abstraction and simplicity ran through all of his work, from LISP to programming language semantics.

However, that does not mean that McCarthy's approach was not practical. Indeed, the summary of the *Working Conference on Mechanical Language Structures* (WCMLS) held in 1963, written by Saul Gorn not long after the publication of McCarthy's 1962 paper, noted that while the main talking points of the conference tended towards semantics as the event progressed, only McCarthy was showing a practical and fully-formed approach to semantics (Gorn 1964). It is interesting also to observe that none of the papers in the proceedings directly concerned semantics, so the conversation turning to this topic indicates its importance in the computing field at the time. McCarthy had not presented a paper at the conference, so the mention of his work is likely a reference to the 1962 paper discussed here.

Gorn, meanwhile, was also arguing for an abstraction of programming, but his idea was quite different to McCarthy's: Gorn proposed that whenever one thinks about the operations of programming languages, they are in terms of some conceptual machine, which he called a 'background machine'. It may be more simple or more complicated based on the particular features of the program and which parts of them are of interest, but in whatever case, it should be specified alongside the language in order to best ensure that the operations of the language are fully understood. This contrasts sharply with McCarthy's perspective, which was to specify as little as possible, taking a very machine-independent view. The tension between the somewhat opposing desires for abstract and full specification is one which crops up many times in the story of semantics.[12]

The WCMLS discussion also contains a good summation by McCarthy of his ap-

[12]See, for example, the axiomatic model for storage used in Vienna operational semantics, discussed in Section 5.5, or the abstract model for storage used in denotational semantics (Section 6.4).

proach to semantics.

> I want to say that what I mean by semantics is [...] an attempt to make a correspondence to what is done in mathematical logic when one discusses the semantics of formal systems. [...] What is appropriate for the semantics of a program in a programming language is the effect of this program on some kind of a state vector describing the state of a computing process. [...] What is meant by the semantics of a program is this function taking an old state vector into a new one.
> (McCarthy, in Gorn 1964)

As this quotation shows, the core of McCarthy's approach is very simple. Only a very small section (13) in McCarthy's 'Towards a Mathematical Science of Computation' was required to establish a lot of the very important concepts in semantics, operational semantics in particular. McCarthy's semantic function bore a strong similarity to the *eval* function, showing McCarthy's ideas developing from their earlier forms; what the approach really needed was a larger definition to showcase its utility. That came in September 1964 at the *Formal Language Description Languages* (FLDL) conference, held in Austria.[13]

McCarthy's paper was the first given at the conference, and was entitled 'A formal description of a subset of ALGOL' (McCarthy 1966). The use of 'subset' is important: the only statements allowed in 'Micro-ALGOL' were assignments and conditional jumps of the form **if** x **then go to** a. McCarthy (1966, p. 1; original emphasis) described this language as "a language for programming *about*, not for programming *in*". This is not the obvious choice for a demonstration language, which would more typically include looping and other control structures while skipping the (trickier) jumps.[14] This choice of language subset makes for a slightly more complicated but perhaps more convincingly useful semantic description, especially considering how much contortion and complication is required to handle jumps in other semantic approaches. McCarthy did not include procedures in the 1964 paper, but did address the topic at WCMLS the previous year:

> I think that one can hint at what is meant by the semantics of a procedure call and this is in terms of the effect of the procedure call on the state vector, namely: Where is the computer now? Well, it is in a

[13] See Chapter 4.

[14] McCarthy always was a supporter of **goto** in programming languages, believing there to be a nice correspondence between programs with jumps and abstract state machines, as indicated in a comment made at MFPS (Jones et al. 2004).

> procedure which has been called by name with the following parameters so that the state of operating ALGOL under this procedure is involved. I think if one wants to complete this description of the semantics of ALGOL then one has to complete a description of the current state of the ALGOL program.
> (from Gorn 1964, p. 135)

McCarthy's FLDL paper began with a brief recap of the approach to semantic definition described in his 1962 paper, before giving a short informal description of Micro-ALGOL and an example of a program. The formal description began with the abstract syntax; in this case, only analytical is given. The use of abstract syntax was justified as follows: "Questions of notation are separated from semantic questions and postponed until the concrete syntax has to be defined" (McCarthy 1966, p. 6). Synthetic syntax was not needed because the ability to construct parts of the language is not necessary; precisely *how* the analytical syntax can recognise or deconstruct elements was also not required in this context. If a parser were to be built, it would be important to include that level of detail, but for the purposes of a human reader understanding the language, simply the existence of identification and selection functions is sufficient (McCarthy 1966, p. 10).

The concept of the state vector ξ was used again as the core of the semantics of Micro-ALGOL. There was an extension to the state vector this time, however: the statement number *sn* was considered to be part of ξ at all times as a 'pseudovariable' used to keep track of the statement to be executed next. McCarthy kept the state deliberately simple to maintain simplicity in the presentation of the method; the language subset was chosen for precisely this reason. McCarthy (1966, p. 11) noted that he "carefully threw out anything that raised any complications in terms of the state," adding that the program counter too "really ought to be treated separately from the other things". This came up in discussion with Strachey, who had queried the limited nature of the language and the lack of separation in the state.

The semantics of the Micro-ALGOL was defined by a series of functions. The first of these is a $value(\tau, \xi)$ function used for the evaluation of expressions. Clever use of recursion here allows the evaluation of arbitrarily nested expressions, matching the structure of the abstract syntax. With an evaluator function for expressions defined, the overall semantic function could be written: $micro(\pi, \xi)$. It takes a program and a state and then repeatedly checks the first statement, ending the interpretation and returning the current ξ if the statement is final; then the current ξ is the returned

value. Otherwise, the type of the statement is determined and interpreted. An assignment results in an adjustment of the state and a goto causes the test to be evaluated and a potential adjustment of the statement number. Performing a jump does not result in a modification of the state vector as it applies to values; however, the inclusion of the statement number as a simple variable clouds this somewhat. This property that state values remain unchanged except when assignments are made is a critical notion in the semantics of languages without side effects; proving that this holds true is often an important correctness or safety proof.

As given, *micro* looks at first glance to be a function of only a program π and a state vector ξ; however, the statement number n and the current statement or term τ are brought in through lambda abstractions. No real use is made of the higher-order functional notions in play here and *micro* could just as easily be a function of four parameters. It is important to note that this use of lambda notation is very different to that used in the denotational style of semantics[15] in which the meanings of programs are functions themselves, not the semantic objects resulting from functions as in McCarthy's approach.

The inclusion of the program π in the *micro* function meant that the entire program text could be available at any time to the interpretation function which then picked out the current statement from the text.[16] This was required in McCarthy's style to enable jumps to be made backwards: a statement's interpretation comes from its text, so this text must be saved for potential re-use. McCarthy was, however, careful to keep the program and state separate. His desire was to "put everything that remains constant in the program and put the things that remain variable in the state" (McCarthy 1966, p. 10).[17] This is why the *micro* function had fewer parameters than might seem necessary.

A notable aspect of the presentation of the semantic function *micro* is its similarity to the *eval* function described in the paper presenting LISP (McCarthy 1960). Like the earlier *eval*, *micro* was defined recursively by cases using conditional expressions; it too operated on a text and a construct associating variables with values. In this way we can see McCarthy applying the ideas he had developed for LISP—originally

[15] See Chapter 6.

[16] This approach is in direct contrast to later styles of model-based semantics that use nested functions to break down the size of the program under consideration with each nested call.

[17] This remained a concern for all kinds of semantic styles; precisely how the split is made depends on the language to be defined. McCarthy noted that self-modifying programs might need the program text to be a part of the state.

intended as a way to describe the kind of computations needed in AI—in other areas. The approach treated the description of programming language semantics in the same way he had hoped to treat computation, as discussed in 'Towards a Mathematical Science of Computation' (McCarthy 1963). As mentioned previously, *eval* was not initially intended to become a full-blown interpreter for LISP, and it is interesting to wonder whether McCarthy would have seen the idea's application for semantics had it not become so strongly associated with interpretation. Another way in which McCarthy's Micro-ALGOL paper has connections to publications on LISP is in the inclusion of two 'concrete realizations' of abstract Micro-ALGOL. These are alternate notations for syntax and implementations of the interpreting function. One of these forms is S-expression, showing the strong links to the LISP way of thinking about programming. These attempts of McCarthy to link with LISP were also noticed by Priestley (2011, p. 235), who saw this as part of McCarthy's desire to frame programming language theory within the bounds of recursive function theory.

The remarks with which McCarthy closed the paper conceded that the semantics of full ALGOL would be considerably more complicated than those of Micro-ALGOL, and that the crucial task would be the definition of the state. McCarthy argued in favour of the intuitiveness of his definition approach, claiming that programmers think about statements naturally in terms of their effect on a state. The ideas presented in the paper, according to McCarthy, also fit with the work of Tarski and others on the notions of semantics in mathematical logic; this shows McCarthy once again striving to place himself firmly within the bounds of mathematics as applied to computation (McCarthy 1966, p. 6).

As will be described later, the FLDL conference provides a great source for historical work not only through the papers included, but also the transcription of discussion session comments. The reception to McCarthy's paper is interesting. Some participants, such as Ingerman, were confused by the notion of "statement" due to the extreme simplicity of the language, and the lack of clear distinction between statements and terms. Gorn thought there should be more focus on the "action" of the program, making the point that the meaning of a program being only the final state does not show the meaning of, for example, an infinitely looping program. To Gorn (quoted in discussion of McCarthy 1966, p. 11), "It's the action that occurs going from the initial to the final state which is the meaning we all have in mind". McCarthy's response was that in his language, such a program has no meaning, and as such is not defined by the interpretation function. A final

point was made by Samelson, who argued that McCarthy's method, while clever, is merely providing a translation to a new notation, the meaning of which then surely also needs description. McCarthy had already addressed that point in his paper, quipping "Nothing can be explained to a stone; the reader must understand something beforehand" (McCarthy 1966, p. 7). He agreed that it was a good point, and that there are indeed philosophical queries about the utility of such formalism, but following Tarski mathematicians had become sufficiently interested in the questions that could be asked and answered in such an environment that they drowned out the philosophers objecting. The same could well happen in computing.

3.1.5 Correctness of compilers

One such application was of particular interest to McCarthy towards the end of the 1960s: proving the correctness of language implementations and compilers. This motivation was stated at the end of both 'Towards a mathematical science of computation' and the Micro-ALGOL paper (McCarthy 1963, 1966). McCarthy's ideas for approaching the problem were briefly outlined in the earlier paper. If the language had already been modelled, in the style discussed, then the machine's operations—the compiled programs—could also be represented in a similar way: by a function "giving the effect of operating the machine program on a machine vector" (McCarthy 1963, p. 22). Assuming an appropriate equivalence relation could be defined between machine and abstract state vectors, the correctness of a translation could be formulated: the state produced by the semantic function applied to an abstract program should be equivalent to that produced by the machine operation on the translated program.

A lengthier treatment of this idea was presented in a paper published by McCarthy and Painter (1967), 'Correctness of a compiler for arithmetic expressions', Painter was a PhD student of McCarthy. This used the same principles of abstract syntax, state vector, and semantic function as in McCarthy's earlier work and the language defined was even more limited in scope than Micro-ALGOL: only mathematical expressions composed of additions of constants and variables. The machine language was similarly pared-down, with a simple series of load and store operations; the authors acknowledged that without a jump instruction there was a "severe restriction on the generality of [their] results which [they] shall overcome in future work" (McCarthy and Painter 1967, p. 2).

Full abstract syntax was described for both the 'source language' and 'object language' (terms that became the norm in discussion of compilers). No formalised concrete syntax was provided, because "no commitment to a particular notation is made" (McCarthy and Painter 1967, p. 2).[18] This somewhat dodged the issue of catching errors such as type mismatches, because such a small language could not contain any. Object programs were written in a LISP-like style, as lists of instructions joined together with concatenation. Semantics for both languages was given in the classic McCarthy style with state vector lookup and alteration as the only core concepts. The compiler itself was defined with a LISP style recursive function, and used an assumed mapping between abstract and machine states.

The correctness of the compiler was formulated in terms of equality of states produced from the source language and object language semantic functions, with the exception of the accumulator and temporary register elements in the machine representation. The proof was made by induction on expressions, with the hypothesis that if the correctness holds on the two parts of a sum, then it holds for the sum. The remarks after the proof note that adding other basic arithmetic operations would be trivial, but the introduction of other concepts such as assignments, conditionals, and jumps would be much trickier. The authors acknowledged that "a complete revision of the formalism will be required" (McCarthy and Painter 1967, p. 10).

This revision was provided in Painter's thesis, 'Semantic correctness of a compiler for an ALGOL-like language' (Painter 1967a). The language under definition was named 'Mickey' and is an extension to McCarthy's Micro-ALGOL language to include input and output as well as Boolean and conditional expressions. The inclusion of I/O required input vector modelled acts as a straightforward list of values; an output string was also maintained as part of the state. Modelling jumps meant that the whole of the program text was kept throughout the entirety of the interpretation, as McCarthy's Micro-ALGOL, and the current statement was picked out using a statement counter.

It is interesting to see that Painter used essentially all the same ideas from McCarthy's earlier works, but provided a much more rigorous and fully formal treatment. For example, while McCarthy did not say much about the structure of state vectors and the precise workings of the c and a functions for reading from and updating the state, Painter wrote a complete specification of these. There was nothing

[18] Page references to republished version, again for ease of access.

terribly new in Painter's work, but it went a long way to showing that McCarthy's ideas were realisable.

Another point worth mentioning is the size of the document. Micro-ALGOL's semantics takes up about half a page and Mickey's is more like a whole page. This minor difference would appear to indicate similarity in the complexity of the language under consideration. However, the proof in the earlier paper is approximately 2.5 pages in length, whereas in Painter's thesis, the proof of compiler correctness spans 54 pages. Even allowing for the aforementioned difference in rigour, it is remarkable how much difference the inclusion of what seems like a few extra concepts makes to the complexity of proving properties. Of course, the possibility remains open that the document is simply poorly written; indeed, in a later discussion of work on correctness of compilers, McCarthy (in Walk 1969b, p. 16) described Painter's proofs as "clumsy". Shortly after finishing his PhD, Painter left academia to work for IBM research in San Jose; he was involved in the organisation of the 'Mathematical Theory of Computation' (MTOC) conference held at IBM Yorktown Heights, in New York, on 27th–30th November 1967 (Painter 1967b).

Although the area of compiler correctness would remain a very important one for those working in the area of formal semantics of programming languages, McCarthy was beginning to move away from semantics. Painter, too, appears to have ceased working in the field at this point; certainly there are no more publications by either on the subject. Jones recalled[19] that at MTOC McCarthy was instead becoming interested in Zohar Manna's new ideas on assertions—even to the point of attacking Mike Paterson's presentation on schemata. The assertions work from Manna was published towards the end of the 1960s and included a paper co-authored with McCarthy (Manna 1968a,b; Manna and McCarthy 1969). McCarthy's interest in Manna's work can also be seen from the minutes of a meeting of IFIP WG 2.2, a community focused on semantics of programming languages,[20] held in September 1969. By this time, he had stopped advocating the operational semantics approach from earlier in the decade and was pushing Manna's ideas (McCarthy, in Walk 1969b). A letter written by McCarthy in March 1968 shows him excitedly describing the method, and remarking that the feature which particularly appealed to him was the ability to prove termination under certain criteria. McCarthy (1968, p. 4) wrote "the importance of Manna's result—and this is a surprise to me—is that it

[19] In personal communications, 2017.

[20] For more on this working group, see Section 7.2.

shows that we can prove the termination of programs exactly to the extent that we can axiomatise the base domains".

Part of the reason for McCarthy's move away from formal semantics, and the mathematical theory of computation, was the cool reaction he received at FLDL. In a letter to Ershov, he wrote:

> I was discouraged at Baden by the reception the ideas received, especially from the Dutch and Germans, and I thought I would have to devote much effort to get a good article and going to meetings and arguing. Anyway, I didn't do any more which I now regret.
> (McCarthy 1965, p. 5)

Perhaps due to this, McCarthy's motivations for approaching semantics also appeared to shift towards the end of the 60s, with the notion of compiler correctness coming to the fore and less mention being made of the mathematical shoring up of computing which was so prevalent in his early works. For example, at another WG2.2 meeting, he asked Garwick whether the scheme that man had just presented "advance[d] the goal of proving correctness of compilers and the equivalence of descriptions?" (McCarthy, in Walk 1969b, p. 3).[21] In the following WG 2.2 meeting McCarthy also re-iterated that achieving compiler correctness should be the main goal of working on semantics of programming languages, no longer even mentioning the idea of equivalences (McCarthy, in Walk 1970, p. 46).

3.1.6 Characterising John McCarthy

The current Section concludes with a brief look at McCarthy as a person. His personality was complex: although those who knew him praised his generosity and eagerness to teach, he was also quite capable of being somewhat prickly. In an obituary, Meyer recalled an incident:

> He also had a facetious side. At the end of a talk by McCarthy at SRI, Tony Hoare, who was visiting for a few days, asked a question; McCarthy immediately rejoined that he had expected that question, summoned to the stage a guitar-carrying researcher from the AI Lab, and proceeded with the answer in the form of a prepared song.
> (Meyer 2011)

[21] Garwick, for his part, replied "Probably no, but this goal anyway has not been reached so far."

Jones (2012) also remembered this side of McCarthy, writing:

> At another event, an IBM speaker was incautious enough to hint that there was related research that he couldn't talk about because it was company confidential. John was to give the after-dinner speech and clearly had prepared notes but after a few minutes put these aside and delivered a withering attack on companies that expected to benefit from science without sharing all of their own research. Anyone in the audience from IBM—and that included me—was left feeling profoundly uneasy.[22]

Both writers also mentioned McCarthy's generous qualities, however, with the first remembering McCarthy indulging his lack of knowledge about undecidability and taking time out from a lecture to carefully explain it, and the second McCarthy's help in demonstrating how to remove snow chains from a car at a particularly wintry November 1967 conference. A glimpse into McCarthy's character and beliefs may be gleaned from reading his science fiction short story *The Robot and the Baby*,[23] which comes complete with a lengthy selection of LISP code to justify the behaviour of the robot.

McCarthy was noteworthy for being engaged with board games, calling chess in particular "the *Drosophila* of artificial intelligence", referring to the fruit flies which are of tremendous utility to biology researchers (McCarthy, quoted in Myers 2011). The same source adds that McCarthy was known in 1966 for playing four chess games simultaneously with opponents in Russia, the moves being carried by telegram over a period of months. A series of photographs of McCarthy in 1967 show him playing chess (or at least posing with a chess board) in front of an IBM 7090 machine; one such is shown in Figure 3.3.[24]

Hermann Maurer, who was assigned to McCarthy as a local guide during the *Formal Language Description Languages* conference in Vienna in 1964, added:

> He had learnt to play Go (the Japanese board game) 4 weeks before he came to the meeting, and he beat me (who had played it for two years

[22] McCarthy may have been extra annoyed because this had happened to him before: he was told that the IBM Laboratory in Vienna had begun to adapt his approach to the description of PL/I, but that he wasn't allowed to see before it was published because it was IBM confidential. This he found galling (McCarthy 1965, p. 9).

[23] `http://www-formal.stanford.edu/jmc/robotandbaby/robotandbaby.html`
The reader should be warned of casual misogyny and strong language in this piece.

[24] Image courtesy of Chuck Painter, according to the Stanford obituary of McCarthy (Myers 2011).

Figure 3.3: John McCarthy with a chess board sitting at an IBM 7090, 1967.

> with a good teacher) dramatically. This is interesting, since you either have a Go mind or not, like you are a musician or not. John had a Go mind. I was playing at that point at level student 3 (after you reach student 1 you turn into dan level 1, and dan level 12 is the best living Go player). The point is, if after two year you have only reached student level 3 the best you can achieve, no matter how much you learn, is dan 1 ... Now John was after weeks at level dan 2!
> (from personal communication, 2016)

This desire to play around with games and computing actually played a role in the development of one of the key aspects of LISP: conditional expressions. McCarthy (1981, p. 5) wrote that he first developed the concept while writing a chess-playing program in FORTRAN, noticing that the need to change values based on some condition was extremely important. Using pure FORTRAN to do it, however, was clunky due to the way that language structured its statements, so McCarthy wrote a FORTRAN function. This came up against the inefficient way that FORTRAN processed conditionals (both branches would always be calculated), leading McCarthy to desire something more inbuilt and usable.

In summary, McCarthy was a multi-faceted person, interested in many different areas, and making important contributions to diverse aspects of computing. His work on semantics was particularly relevant to the current story, setting out as it did many of the very fundamental aspects of what came to be known as operational semantics (and even some indirect influence on denotational approaches). The concepts of a simple state and an interpretation function taking a program and an initial state into a final state provide half the core of subsequent approaches to operational semantics. The other half was provided by the work of another famous and influential computer scientist, operating in the same time frame as McCarthy: Peter Landin.

3.2 Peter Landin

A very important figure in programming language semantics and theory in the early part of the story is Peter John Landin (1930–2009). Born in Sheffield and educated at a local grammar school, Landin studied mathematics at Clare College, Cambridge, between 1949 and 1953. He was identified as particularly able, and put on a fast-track shorter course (Landin 2001), although this somewhat backfired, and he emerged with only a 3rd class degree (Gaboury 2013).

At Cambridge, Landin learnt about lambda calculus from an enthusiast or even pupil of Church, potentially named Norman Steam[25] (Landin 2001). This leads to an amusing anecdote: Landin (2001) recalls that the analysis class was split into two streams, and his stream, the lambda calculus stream, shrunk until there was only Landin and one other left. On one occasion they realised they had both missed a lecture, but assumed that "due to the inflexible social manner of Norman Steam that he delivered his lecture anyway." The students did not understand the notes sufficiently well to determine whether there was a gap.

During this period, Landin also learnt about universal algebra, a fact about which he considered himself very lucky, and almost unique among people in computing at that time (Landin 2001). In fact, another very important figure had also studied both universal algebra and lambda calculus: Dana Scott.[26] Scott (2018), too, acknowledges that it was a very useful combination.

[25] This is the name Landin gives for his teacher; however, Church does not appear to have taught anyone with that name. Nor has a search of Clare College records turned up any teacher named Norman. Nevertheless, let us assume that Landin was not making up the entire story.

[26] Scott was crucial to the success of denotational semantics; see Section 6.4.

3.2.1 Introduction to computation

After graduating from Cambridge, Landin moved back to Sheffield and tried to decide what to do next. During that time, he met a most curious individual:

> When I ceased to be an undergraduate, and because of being fast-laned through all these things and leaving with a rather ambiguously low grade degree I was very uncertain of what to do with my life and spent the next six months in a Sheffield reference library trying to avoid making a decision about my life. I used to go out to a café just around the corner from this reference library [...] and one day I was having my coffee in Fields café, and a voice came booming across the crosswise tables; and this voice said 'I say! Didn't I see you reading *Principia Mathematica* in the reference library this morning?' And that's how I got to know the legendary Mervyn Pragnell, who immediately tried to recruit me to his reading group.
> (Landin 2001)

Shortly thereafter, Landin moved to London, and was surprised to run into Pragnell again. At that point, the group was reading Markov's *Theory of Algorithms*,[27] and Landin became a regular attendee. The group's reading material was somewhat eclectic; when Landin first encountered Pragnell in Sheffield they were halfway through the six volumes of Gibbons' *The History of the Decline and Fall of the Roman Empire*. The way the group approached the reading of texts was distinctive: "the structure of these meetings was very theological in style. You would read in clockwise order, aloud, pages, miming the formulas to the best of your ability" (Landin 2001). Landin (2002) also described them as "clock-wise bible-reading meetings", a sentiment echoed by Rod Burstall (2017),[28] who likened the experience to being at "kirk".[29] Burstall also added that they would read aloud a page at a time, and whenever someone new joined, they would start again from the beginning! A number of other influential computer scientists also found themselves passing through Pragnell's reading groups: Robin Milner, George Coulouris, John Iliffe, and Bill Burge (Burstall 2017).

Landin and Burstall became good friends in particular, with Burstall (2000) recalling that he got his real education in computing from Landin in the Duke of

[27] Landin (2002) later noted "somebody said it might be applicable to proteins".
[28] More of Burstall's contributions are discussed in Sections 6.2 and 7.4.
[29] 'Kirk' is the Scots word for 'church'.

Marlborough pub around the corner from Birkbeck College. Burstall had his own interesting story about meeting Pragnell:

> I asked a gentleman standing next to me... I thought he worked in the shop, so I said, "Do you have *Elements of Mathematical Logic*?" and he said, "I'm not a shop assistant," and stalked away. And then he came back after a bit and said, "But it's a very good book." And then he said, "Would you like to come to my seminar in Birkbeck College?" so I thought he must be a professor or a great eminence. So I said, "Of course I'd be honoured to come to it." It turned out that he knew a lab technician who had a key to a building who would let us in, so we had this seminar with this guy called Mervyn.
> (Burstall 1993)

In the same interview as above, Burstall remembers that Pragnell was a strange fellow, certainly not an academic. He had taken logic and philosophy at Bristol and passed logic but failed philosophy. The seminars were somewhat furtive, as the lab technician was not supposed to allow people into the building in the evening.

Landin's first brush with computers came around 1954, when he saw the EDSAC in operation, but was not impressed with its "stumbling" performance (Landin 2001). At this time, Landin had started working for English Electric (EE), spending 1954 to 1960 there. One task he applied himself to was trying to use Hollerith machines for a job determining the conductivity of North Sea herring for a government agency (Landin 2001); equally fruitlessly, Landin found himself toying with trying to get first EE's Pilot ACE and then its DEUCE to do β-reduction. Towards the end of the 1960s, Landin's interest in programming languages began to develop, and he contributed a number of suggestions for improvements to ALGOL 58 during the period leading up to ALGOL 60 (Naur 1981b, p. 121).

Attending a conference at the National Physical Laboratory in 1958, Landin heard about LISP; the use of a garbage collector impressed him especially (Landin 2001). He noticed, though, that McCarthy had the implementation of lambda calculus wrong, later calling LISP "a sort of FORTRAN-like language about lists with lambda in it" (Landin 2001). Landin became interested in "tidying up some loose ends" from McCarthy's work. In particular, he liked Alan Gilmore's attempt to write an abstract LISP machine: it was messy, but Landin could see a way to tidy it up. Specifically, he was interested in creating a symmetry between functions and arguments. This led into Landin's own famous abstract machine, the SECD machine, which is discussed in more detail later in this section.

3.2.2 Consulting with Strachey

Landin's work in theoretical computing began in earnest in 1960, when he started working for Christopher Strachey[30] (Landin 1967). The pair had a fruitful partnership for four years; Strachey was at the time an independent computing consultant and Landin was his only employee (Campbell-Kelly 1985, p. 31). Landin was brought in to cope with the greater demand than Strachey had anticipated, and to help him debug his programs "by the famous method"[31] (Landin 2001). Landin acknowledged the effect of Strachey's support on him, writing "adopting a drastic stand against [LISP's weaknesses] would have been beyond my self-confidence but for Strachey's self-denying and perceptive support" (Landin 2000, p. 75). For his part, Strachey was proud of the fact that he was paying Landin to do research in theoretical computing, writing later that he was financing "the only work of its sort being carried out anywhere (certainly anywhere in England)"(Strachey 1971a, quoted in Campbell-Kelly 1985, p. 31).

The main task for which Landin had been hired was the provision of a Mercury Autocode-compatible compiler for a new Ferranti ORION computer (Landin 2001). Its "rococo endowment of address-modification and indirection" made producing optimal code awkward and suggested using an "interlingua" (Landin 2000, p. 75). Thanks to his mathematical background, Landin thought the best way to do that would be to use lambda calculus.[32] He believed the compiler should emerge from the semantics of the language, something the technicians building the computer's 'supervisor'[33] program were not quick to accept (Bornat 2009a). Landin's previous interest in LISP led him to the idea of creating LISP-like abstractions of the Autocode and generating a compiler from that (Landin 2000).

Writing in a statement of his interests as part of a job application, Landin (1967) described his motivation for the work as being a way to cope with the vast multitude of computing systems abounding in the world at the time. What was required was a

[30] Strachey is most notable in this story for his development of denotational semantics. His part is mostly told within Chapter 6.

[31] Strachey was known for this: he would give a program to an underling and tell them to find the bugs in it. The combination of a different set of eyes and the terror of disappointing the great man would invariably lead to bugs being found—even in programs Strachey had previously thought correct.

[32] Trakhtenbrot, Halpern, and Meyer (1983, p. 21) attribute the first use of lambda calculus to underpin a programming language to Landin.

[33] A program that would run constantly on the machine, running other user programs, performing a similar role to certain aspects of modern operating systems.

"conceptual framework" for considering and comparing such systems. Much activity in computing required performing fairly mundane tasks, the effective mechanising of which requires that the systems themselves also be properly mechanised. One way to achieve that mechanisation, claimed Landin, was through grammars and syntax analysis, but he was more interested in the semantic angles. Landin, like McCarthy, saw this work as a way to bring together computing and mathematical systems.

3.2.3 Mechanically evaluating expressions

The first expression of these ideas, developed while working for Strachey,[34] was written down in the very influential 'The mechanical evaluation of expressions' (Landin 1964). In brief, the core idea was to translate from programming languages into a lambda calculus, and then give meaning to these expressions with an abstract machine.[35] The paper gives us a good idea of Landin's view of the definitions of syntax and semantics:

> The relationship between expressions and their written representation encompasses all that is customarily called the 'syntax' of a language and part of what is customarily called its 'semantics.' The chosen name/value relation, together with the primitives themselves (that is to say, the applicative relationships between them) constitute the rest of what is customarily called the 'semantics' of a language insofar as it is distinct from the semantics of other languages.
> (Landin 1964, p. 319)

The central concept of the paper revolves around the translation from program parts into what Landin calls "applicative expressions", or AEs. This translation is somewhat like abstract syntax: "a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations" (Landin 1964, p. 308); Landin acknowledged this similarity to McCarthy's ideas about abstract syntax. The program parts in question, it is important to realise, are solely applicative, i.e. ultimately resolvable to a value or function. Imperative aspects of computing such as assignment, jumps, and sequencing were not handled in this paper.

[34] All this theoretical work meant the compiler development suffered somewhat, and had to be subject to heavy modification from staff at Ferranti to get it to work properly (Campbell-Kelly 1985, p. 31).

[35] The first aspect ended up at the core of denotational semantics, as described in Chapter 6, and the second in the Vienna group's style of operational semantics, as described in Chapter 5.

$(u - 1)(u + 2)$ $\quad$ $\{\lambda u.(u - 1)(u + 2)\}[7 - 3]$
where $u = 7 - 3$.

Figure 3.4: An example AE pair with equivalent meaning but different form.

AEs were essentially syntactically-modified lambda calculus expressions; Landin called these modifications 'sugaring'. The term 'syntactic sugar' is one still in use to refer to modifications that change a piece of text's appearance, typically for better readability, without changing its meaning. An example pair is shown in Figure 3.4 (the AE on the left has two lines, and is equivalent to the one on the right which has one line). Landin began by formalising the left-hand style by showing its equivalence in lambda calculus, and then, having done so, used the more illustrative left-hand style throughout the paper.

All kinds of structures could be defined in this way, including simple arithmetic expressions, and more complex objects like lists. However, Landin's approach to list manipulation, compared to McCarthy's, reflected Landin's interest in higher-order abstraction: whereas LISP's *cons* function is a function with two parameters, either lists or items, that produces a list, Landin's *prefix* function is a function-producing function which is then *applied* to two lists. Some particular kinds of expression required some additional care: recursive expressions needed a fixed-point operator to prevent a purely circular definition, which is to say a lambda-defined function applied immediately to itself. For this task, Landin used Curry's Y combinator.

The evaluation of AEs was given in terms of some 'background' information, namely the values of the free identifiers used in the AEs. Note this is a similar term to the 'background' referred to by Gorn (1964) in his summary of the *Working Conference on Mechanical Language Structures*, but for a somewhat different concept. Gorn meant the imagined computing machine with registers and storage that programs would operate upon, whereas Landin was imagining a very abstract 'environment'. The environment as Landin used the term was simply a partial function associating identifiers and values. Intuitive evaluation of AEs was straightforward: either an identifier is simple and it is looked up, or it is a combination, in which case the operators are looked up and evaluated before the operator is applied. For lambda expressions, a lambda abstraction is formed to describe the function: the body is evaluated in a new environment formed by the modification of the current environment to include the newly-bound variable. This combination of a function and its environment was called by Landin a 'closure'.

Mechanical evaluation of the AEs required some more formalism. The environment is modelled as a list of name-value pairs, and closures are formalised with a constructor function. In order to mechanise the evaluation, a class of constructed objects called 'states' is defined, and 'transition rules' added to describe the transformation of these states. Ultimately, a state is left that contains in it somewhere a value or closure representing the 'result of evaluation' of the AE.

A state is comprised of four parts: a Stack, which is a list to hold intermediate values of nested expressions; an Environment, as previously described; a Control, which contains a list of AEs yet to be evaluated; and a Dump, which is a stack of entire states, needed if there are function calls. The dump is used to hold a previous state of computation to be resumed after the current function call terminates. The first letters of each state component give rise to the name 'SECD'.

This combination of states and transition rules is called a 'machine' by Landin, and shows some inspiration from the 'abstract computer' of Gilmore (1963) as mentioned earlier. However, Gilmore's approach was not an abstract machine for interpreting LISP, as might be expected, but rather a loosely-specified computer whose machine code is LISP-like. Landin (2000, p. 75) described it as a "valiant try" whose great merit "was to expose LISP's ramshackle semantic features". Gilmore's machine had much more in common with a real computer, such as registers and modelled storage, than Landin's. The only really shared aspect is a stack (called a 'push-down list' by Gilmore) for the evaluation of expressions; Landin's insistence on crediting Gilmore for the basic idea seems like an example of Landin's characteristic self-effacing nature.

The transition rules of the machine were sketched out in natural English, and then formalised in AEs.[36] Having finished the abstract presentation, Landin then explained a way by which this system could be represented on computer. A way to represent each state component by an address was described, although briefly. The Y combinator in particular requires careful handling, and Landin provided a quick sketch of how to handle this in an address-based model. He also noted that many other evaluation techniques could exist as alternatives to the ones mentioned in this paper, including partial evaluation, and compiler re-ordering for efficiency. This was remarkably prescient, as modern models of weak-memory, which utilise such effects, continue to cause headaches for formalists today.

[36] This definition of a system using the system itself is another clear indication of Landin's influences from McCarthy.

The mostly informal nature of the paper, giving explanations with words and lots of examples, illustrates Landin's mathematics background. He credited Church and Curry as his main inspirations, as well as McCarthy and the authors of the ALGOL 60 Report. 'The mechanical evaluation of expressions' is very typical of Landin's nature, with a self-effacing tone throughout, as evidenced by the introduction: "It leaves many gaps, gets rather cursory towards the end and, even so, does not take the development very far. It is hoped that further piecemeal reports, putting right these defects, will appear elsewhere." (Landin 1964, p. 308). The conclusion continued this theme: Landin (1964, pp. 319–20) described the work as a "sideways advance", stating "most of the above ideas are to be found in the literature". He did concede that although (by his estimation) there was little novelty, the introduction of new jargon could nevertheless be useful for its new features. Landin also allowed himself to take credit for the formalisation of the abstract machine, something for which he believes there is no precedent (Gilmore's machine having been described with a flowchart).

Later authors have been considerably kinder to Landin's innovativeness. Danvy and Millikin (2008), for example, cite 'The mechanical evaluation of expressions' as providing all the essential aspects of what would later become functional programming, as well as the seeds of continuations and continuation-passing style. In particular, the notion of a closure is essential in functional programming. Landin saw closures as an alternative to the 'literal substitution' used in Church's normalisation process, which is to say the 'copy rule'[37]. Landin also acknowledged the great power and flexibility of AEs, noting that simply changing the primitives in use allowed their application to many problem domains. However, so far, only applicative languages could be handled this way.

The 'Mechanical evaluation of expressions' approach received a small expansion in a paper called 'A λ-calculus approach' (Landin 1966a), published in the proceedings (Fox 1966) of a 1963 summer school called *Advances in Programming and Non-numerical Computation* held in Oxford at the Computing Laboratory. The background to this summer school is worth mentioning: at the beginning of the 1960s, the Computing Laboratory at Oxford was run by Leslie Fox, a mathematician who viewed computers as tools for performing numerical calculations. He published a piece in the BCS *Computer Journal* in which he scathingly dismissed ideas that there was any point studying or teaching programming techniques (Fox 1961). A

[37] See Section 2.3.

controversy in the letters section followed, with a particular rebuttal from Strachey and Stanley Gill who argued that the mathematical foundations of programs were very important indeed (Campbell-Kelly 1985, p. 32). Fox agreed in response to host a summer school at the Computing Lab in 1963, with Strachey and Gill to act as organisers. One of the speakers was Landin; others included Gill, Strachey, David Barron, and Donald Michie. By the time of the proceedings' publication, Fox was pleased with the success of the school, as indicated by the tone of his introduction to the proceedings (Fox 1966).

Landin's paper in the volume was largely a reprint of his 1964 publication, including a new introduction, a section providing an overview of the notions of syntax and semantics, and two appendices. The changes from the earlier publication are not significant; more interesting is the new name. Compared to 'The mechanical evaluation of expressions', 'A λ-calculus approach' emphasises the foundational aspects a lot more clearly. The new section defining syntax and semantics is worth quoting also:

> A 'complete' description of a computing language is, roughly speaking, one that specifies, for each text, first whether it is admissible, and secondly, if so, what is the outcome of 'running' it. (In the special case of a programming language this means specifying the outcome for every relevant initial state of the system).
> (Landin 1966a, p. 101)

This is a more thorough and well-realised definition than the one in the first printing.

3.2.4 ALGOL application

A more extensive overhaul of Landin's technique came in 1964, at the paper he presented at the *Formal Language Description Languages* conference. Interestingly, the report prepared by Zemanek (1964b) to explain the conference organisation progress to IFIP TC-2, did not include Landin on the list of initial invitees. However, Gill—perhaps knowing of Landin's worth from attending the summer school with him—wrote to Zemanek to ask him to invite Landin (Zemanek 1964a). Gill was a member of IFIP TC-2, and attended the meeting at which Zemanek presented his report, so it is likely he noted Landin's omission and asked for his invitation.

The topic of Landin's talk at, and subsequent paper in the proceedings of, FLDL was the application of his method to ALGOL 60. Called 'A formal description of ALGOL 60', it was, according to the preface, "an introduction, offering discursive

remarks instead of completeness" to a later pair of papers on the topic, published in *Communications of the ACM* (Landin 1966c, p. 266). Trying to date the precise time of writing for these papers is slightly involved: the proceedings were published in November 1966, after the twin papers, but the bibliography number for the 1964 paper is [201], and the papers published in 1965 are given [201a] rather than [202], which suggests their late inclusion. Landin's talk was given in September 1964, and all the papers had to be written in advance for distribution to attendees (Zemanek 1981, p. 13). As Landin was invited in May, he likely wrote the paper over the summer. This means the FLDL paper was probably an earlier working out of the idea, before a fuller treatment in the 1965 *CACM* papers.

'A formal description of ALGOL 60' began with an outline of some inconsistencies and problems in ALGOL 60. For example, declarations were not handled precisely the same in every context: for procedures and switches they initialise the object, but in other places they constrain type. Landin argued these kinds of issues were highlighted by the provision of a formal description for a language. Descriptions would also identify elements that had been accidentally left ambiguous in the source language.

Landin continued with a general introduction to the concept of AEs, which received no major changes since the earlier paper. However, the style of description showed a little more care taken to describe the concepts, and properly explore the consequences of their properties. Let us consider an example. Typically, the evaluation of an expression such as $x + y$ would involve first the evaluation of each operand, before the evaluation of the operator, and the application of the latter to the former. In a case such as the evaluation of a conditional, however, one may wish to avoid evaluating both branches first, as one is never going to be traversed. Landin introduced a trick to delay and possibly avoid evaluation by replacing an expression with a lambda abstraction with an empty binding, as in $\lambda() \cdot x$. This turned every evaluation into a 'zero-adic' function which could then be applied to produce the expression in the body.

Reynolds (1981) commented on this particular trick played by Landin in his paper 'The Essence of Algol'. He noted:

> Since [in Landin's style] operands are evaluated before operators, the basic method of parameter passing is call by value, and call by name is described in terms of call by value using parameterless functions (in

> contrast to the Algol 60 report, where call by value is described in terms of call by name using appropriately initialized local variables).

Reynolds went on to argue that this fundamentally different view of the procedure mechanism led to Landin's approach losing some of the 'essence' of ALGOL.[38] He summarised the essential difference as follows:

> Landin was right in perceiving the lambda calculus underlying Algol, but wrong in embracing call by value rather than call by name.[39] (Reynolds 1981, p. 3)

The major revision to the AE/SECD model in Landin's FLDL paper was the inclusion of facilities able to cope with language features that did not fall into the "descriptive" (or "functional", or "non-procedural") category (Landin 1966c). Landin felt the method coped generally fine with these alterations, although the changes led to inelegance with some features. The modelling of commands required this fundamental shift, as previously AEs had been used only for applicative language constructs. Landin noted there are two possible approaches that could be used here. One was to "find, corresponding to each command, an AE denoting the SECD-transformation it effects" (Landin 1966c, p. 277). This is the approach, Landin explains, followed by Strachey in his paper at FLDL, as well as being similar to McCarthy's approach. This reference to McCarthy is a little strange, as while McCarthy had indeed defined a function which describes the state transformation, he had clearly explained on a number of occasions that, to him, the meaning of a program is the resultant state rather than the transformation. It is furthermore interesting that Landin referred to McCarthy's (1960) LISP paper rather than one of the papers that dealt more directly with semantics, implying that Landin had not

[38] As a result, a number of the languages and language approaches that flowed from Landin's work—including Landin's ISWIM (see below) and Reynolds' own Gedanken language (see Section 6.6) failed to be properly ALGOL-like.

[39] This disagreement is also illustrated in a fun anecdote from Danvy (2009b, p. 127); he was lunching with the two in France in 1997:

> I took the opportunity of a pause in the conversation to venture the question as to whether in their mind, the evaluation order of the meta-language of denotational semantics was call by value or call by name. Peter and John immediately, and simultaneously, answered "call by value of course" (for Peter) and "call by name of course" (for John). For a second of eternity, they looked at each other. Then it was like they were mentally telling each other "let's not have this discussion again" and the universe resumed its course.

followed those. In any case, following this approach would require putting references to the SECD machine (as Landin was by this time calling it) into the AEs which modelled ALGOL 60, something he was unwilling to do.

Landin preferred instead to follow another route: "extend the AE/SECD system in such a way that executing an ALGOL 60 program can be directly by 'evaluating' a generalised AE on a generalised SECD-machine" (Landin 1966c, p. 278). This was an idea that would be central to the operational[40] semantics approach throughout the 1960s. Landin effected this by extending AEs to include new components, *assigners* and *programpoints*, and called this new class of object Imperative Applicative Expressions (IAEs). The *assigners* are designed to model assignments and *programpoints* jumps.

Very little detail is given on the behaviour of *assigners*, except that it requires the semantics of IAEs to involve a concept called 'sharing', which "models the notion of 'pointing to the same cell' in a chain list-structure representation of SECD-states" (Landin 1966c, p. 278). This short sentence is all the explanation given on the subject; the concept of the 'sharing machine' was, however, to be given some more detail in the later *CACM* papers.

The modelling of jumps is described in more depth. The core idea was to treat a jump like a parameterless nontype procedure (which is to say one which takes no arguments and returns no value): in other words, simply a transfer of control. This was somewhat similar to the approach of van Wijngaarden: *programpoint* keywords were written into the ALGOL program before its evaluation.[41]

Landin also treated 'own' variables by rewriting the source program. They were handled by moving the declaration of the variable outwards into the global scope, and writing in extra variable decorations for each new resumption of the variable. Landin accepted the less-than-ideal treatment, but explains the blame should be apportioned to the strange behaviour of the 'own' variable and its somewhat poor description in the ALGOL 60 Reports.

Having outlined the correspondence between ALGOL 60 and IAEs, Landin intro-

[40] Strachey was one who characterised Landin's method this way, as he wrote in a later essay (Milne and Strachey 1974, p. 26):

> The approach is 'operational' as the machine, though abstract, is defined by a set of transition functions and the meaning of a program is given by the sequence of states of the machine.

[41] See Section 3.3 for more on van Wijngaarden.

duced a formalised version. The first step was a translation into "abstract ALGOL", which he noted is somewhat like an abstract syntax: "Abstract ALGOL is a system of COs [constructed objects]. They are, roughly speaking, the result of subjecting ALGOL 60 to syntactic analysis and then omitting noise words and format indicators such as '**do**', '**then**', etc." (Landin 1966c, p. 283). It included all the elements typical of an abstract syntax, such as identifying predicates, selectors for parts of composite objects, and constructors for making program parts. Landin then defined translation functions from abstract ALGOL into both concrete ALGOL and IAEs, thereby establishing a correspondence between the two. He admitted that a translation from concrete into abstract ALGOL was more appealing for the task of machine language processing, "but as a precise presentation to a human audience its definition is more cluttered with distracting details" and it is not included (Landin 1966c, p. 284). The importance of the abstract ALGOL is that it provided "a convenient way of referring to components of the phrase-structure tree. Such a capability is important for the clarity of the semantic function" (Landin 1966c, p. 286). This capability would not be shared by a synthetic syntactic definition method such as Backus' notation.

One trivial point about Landin's syntax style is that the selector functions have a charming style: rather than the full words of McCarthy's approach or the lengthy hyphenated names used by the Vienna group, Landin tended to take the latter part of a word, as in *rator* and *rand* for operator and operand.

Finally, Landin described the semantic function; however, it was deliberately incorrect. Substitutions required, such as those transferring 'own' variables to the global level, were not performed, to make the process easier to follow in this introductory paper. The general style was a recursive IAE, which operated by cases on the various parts of the source program, as governed by selectors and identifying predicates from the abstract ALGOL program. A number of auxiliary IAEs were required as part of the semantics, such as *parallel*, which assembled a list of definitions into a single composite declaration; these were for compactness of the final function. The heavy use of such auxiliary functions, however, does disguise the AE-based nature of the final (partial) semantic function. Abstract ALGOL, the translation functions, and semantic function, were all only sketched in the FLDL paper. Landin explained this was because it was an introduction to the topic only, and not yet a fully-realised piece of work.

As previously mentioned, one very useful aspect of the FLDL proceedings is the inclusion of a transcription of the discussion after each presentation. In reception to

Landin's talk, a key point was made by Samelson, who noted that there is a fundamental difficulty in reconciling ALGOL 60 and lambda calculus because the latter is purely functional and the former is inherently imperative. Landin argued that a programmer actually does not use as many assignments as you would expect, with most variables being assigned once only at the beginning of a block. These one-time only assignments could then be easily handled with **where** clauses, as illustrated by their utility in CPL.[42] Landin admitted, however, that lambda calculus did not model imperative ideas easily, and that another way to address that problem was with Strachey's method.

McCarthy—somewhat sharply—asked Landin what the point of his research was, pushing him to answer whether his work helped in understanding the programming language, or for making a compiler clearer. Landin (1966c, p. 293) explained it was a "side-product of a more fundamental investigation" to try to aid in comparing programming languages. The aim was to provide a framework for the discussion, the better to improve future developments of programming languages. While the method could possibly be used for implementing a language, the real gains were descriptive and discursive. He pointed specifically to the discussion of 'own' variables, which he hoped would clear up some of the confusion about their meaning and application. Using his method, it should be "possible to state various views about 'own' precisely in these terms" (Landin 1966c, p. 293). Despite this desire from Landin, there was no fully formalised description of 'own' variables in this paper, although that would come in the later *CACM* papers.

McCarthy also criticised Landin's approach in his own FLDL paper. There, he wrote that techniques which translate programs into a framework such as lambda calculus "have a certain practical value in resolving ambiguities, but they do not correspond to our intuitive ideas; they will make mathematical results difficult to obtain and leave us with the problem of the semantics of the target language" (McCarthy 1966, p. 6). This is an interesting criticism for McCarthy to make given that his own approach faced questions of target language semantics, and that the lambda calculus—albeit perhaps not quite the extension made by Landin—was a more well-established form of mathematical notation and formalism than McCarthy's functions. The intuitiveness argument, however, seems more justified, given the close tying McCarthy established between his semantic function and state, and the operation of a physical

[42] CPL, a language whose name had many different expansions, was developed throughout the 1960s by Strachey, among others. It is discussed in more detail in Section 6.2.

machine.

Overall, 'A formal description of ALGOL 60' showed a strong development from the earlier 'Mechanical evaluation of expressions'. The extension to cover not just putative fragments of applicative languages, but also the majority of a real language, demonstrated much more convincingly the power of the technique. There was a lot more effort shown in the formalism and completeness—perhaps due to the continuing influence of Christopher Strachey on Landin, or perhaps just the extra time spent on the work. However, many aspects were still very roughly sketched out, with some of the more crucial areas—such as the precise mechanisms of the modelling of assignment—being given very little detail. The technique was, at this point, still a work in progress.

The final chapter in the development of Landin's method for describing programming languages with AEs was published in 1965, the year before the FLDL proceedings came out. The paper, entitled 'A correspondence between ALGOL 60 and Church's Lambda-notation' was written before the conference, however. Footnotes indicate it was first submitted in April 1964, and re-submitted following revisions in November 1964 (after FLDL). Acknowledgements were given to Burge, Strachey, and Mike Woodger for reading and commenting on the draft, and it is likely that Landin also took into account feedback received at FLDL. Landin worked on this material while still working for Strachey, but by the time of its publication in Spring 1965, Landin had moved to America to work for Sperry Rand, specifically UNIVAC, as shown by his address on the paper. 'Correspondence' was actually published in two parts, in consecutive issues of *CACM* (Landin 1965a,b), but it was intended as one piece and will be referred to as such here. The first part of the paper is an introduction to the method and an informal description of the correspondence with ALGOL, and the second provides the formulae.

In the motivation section, Landin gave a very good list of reasons for attempting to formalise the semantics of programming languages, with his method in particular. One is simply that by providing a simpler model of a more complex language, meaning can be attached more easily (Landin 1965a, p. 89). This links to McCarthy's point, made both in his paper at FLDL and in response to Landin's, that part of the value in providing semantics is that it is simplifies the task of understanding the language. He wrote:

> The attempt to fit ALGOL 60 into the AE/SECD framework can be

> considered from two sides. On the one hand, for someone familiar with ALGOL 60 it may clarify some of the features of AEs. [...] On the other hand, AEs illuminate the structure of ALGOL 60.
> (Landin 1965a, p. 90)

As Landin had stated in other reflections, a goal for him in working on language theory was to provide a kind of toolkit for working in a situation with multiple programming languages. Formalising syntax had provided benefit to language designers and implementers, he reasoned, and there might be similar advantages from semantics. He noted that ALGOL 60 itself was frequently used as a standard for comparing languages, thanks to its own clear description and favourable power to complexity ratio. A common interpretation language though, such as AEs, might allow for greater flexibility in comparing and even combining programs written in differing languages. Landin admitted that his method tended to favour heavily more descriptive high-level languages than those which are more machine-oriented and concerned with specificities like fixed formats and absolute addresses.

Finally, Landin made the point again that AEs could help identify areas of weakness in their target language:

> Analysis brings out certain respects in which ALGOL 60 is 'incomplete,' in that it provides differing facilities in situations that are closely analogous. It dictates what extensions (and small changes) are required to remove this incompleteness.
> (Landin 1965a, p. 90)

As he had done many times before, Landin started the paper explaining the concepts of AEs and the SECD machine, most of which were unchanged from the previous paper. He made an important distinction from the work in 'Mechanical evaluation of expressions', however: the applicative language concepts discussed therein could be given meaning and value without recourse to a machine, and should a machine be desired, the SECD would be only one of many possible mechanisations. Indeed, the formalisation of the machine-free evaluation process could be used to check and prove the correctness of a machine-based approached. However, imperative features like assignments and jumps caused problems, and it seemed impossible not to use a machine to specify the semantics of the resultant IAEs. The 'sharing machine' used in this paper was still not quite specified perfectly enough for Landin, and he wrote "it is hoped a more exact account will be set forth for publication elsewhere,

and that enough is said here to define all but the details of the main purpose of this—explaining ALGOL 60 in terms of IAEs" (Landin 1965a, p. 92).

The SECD machine did get an update, in the form of a new operator, called J. If this appeared at any point within an expression, the value of the expression would immediately become the value of the body of J. If J was applied to a closure—Landin's term for a function plus its evaluating environment—it would become a 'program closure'. A program closure also contains a dump, which gets installed as the new state of the machine upon its activation. In this manner, jumps into different contexts and scopes could be modelled. Landin described the related notion of 'program-point' in the following way:

> We call an expression of the form $\mathsf{J}(\lambda L \cdot S)$ a "program-point". Roughly speaking, ALGOL 60's labels are a special case of program-points. They are parameterless, and the λ-body is typically a functional product whose terms correspond to the statements following the label.
> (Landin 1965a, p. 92)

This concept becomes particularly important later in the consideration of denotational semantics, as it is similar to the 'continuations' used in that style to model jumps.

It is interesting to note that Landin (1997) joked that he had sneaked the specific naming and definition of the J operator into his paper between the second and third printing proofs.[43] Danvy (2009a) claims to have seen copies of these proofs in Landin's papers, and confirms the story. Previous versions, and previous papers, included a similar functionality, but not so clearly packaged; the J operator has gone on to enjoy a long role in functional programming and continuation-passing style which might not have come to pass without this last-minute alteration. Landin (1997, p. 1-7) [44] remarked that the idea of labelling the operation should have occurred to him before:

> The idea of pulling an ordinary lambda-expression out from inside a program-closure, and the possibility of indicating their relationship uniformly by a name ("J"), should have come sooner. It was exactly another instance of the way Curry recommended looking for the lambda in every binding operator (such as differentiation and integration), and hoping for

[43] "So much for refereeing," he continued. "Indeed, scrutiny of that paper seemed concerned only with whether a space should separate 'Algol' and '60'."

[44] This is not a range of pages; this pages in this source are numbered with a dash.

> a higher-order operation to bridge the gap. Had Y emerged from LISP's "LABEL" it would have been a copy-book example of the phenomenon.

Here again we see an example of Landin's self-effacing nature, as well as his desire to situate his own work within a line of mathematicians and logicians.

Another extension to the SECD machine in 'Correspondence' was required to properly model assignments. This 'sharing machine' concept (already mentioned in previous work by Landin but given more in detail here) was a way to model the retention of values outside of the current environment. The machine had a new 'sharing relation' which modelled the addresses of a computer, and had to be updated whenever a state change occurs. It described which environment elements match across different environments held within stacks or dumps in the state. The final value of an IAE, once evaluation is completed, no longer related to simply the result at the top of the stack, but also the sharing relation:

> For present purposes we can roughly say that the meaning of an IAE has two aspects, a descriptive and an imperative aspect, of which one or other may be unimportant. The descriptive aspect corresponds to the value, or denotation, of an AE. The imperative aspect corresponds to the change of machine state caused by executing the IAE.
> (Landin 1965a, p. 93)

These two aspects correspond rough to the core conceits of denotational and operational semantics respectively, and show how Landin's work would go on to influence both schools.

Continuing to expand on the application of his method and its corollaries, Landin considered the correctness of the translation from ALGOL into IAEs. He formulated this correctness in the same terms that McCarthy had: the ALGOL program and its IAE "should have corresponding effects when executed in corresponding environments" (Landin 1965a, p. 93). However, Landin did not formalise the notion of 'corresponding', as McCarthy had done with his concept of 'partial equality', and Landin also did not provide any proof, seemingly being satisfied simply to state that it should be possible.

The actual correspondence between ALGOL 60 and IAEs is given both informally (in the first part of the paper) and formally (in the second). The informal description is actually slightly more complex as it includes a discussion of the primitives present

in both source and target. Landin also did not give exactly the same correspondence in the two parts, because the IAE produced by a rule, as in the formalism, is not always as intelligible. The informal correspondence covered mostly the same content as Landin's FLDL paper, but with more examples, and more confidently-written accompanying text.

The formal presentation of the correspondence took up the second, shorter half of the paper (Landin 1965b), and again used a combination of 'ALGOL 60 Constructed Objects', or ACOs, for the abstract syntax, and one large IAE written in terms of smaller IAEs to define the semantic function. The class of programs expressible as ACOs is larger than that of legal ALGOL 60 programs, as there is no attempt to prevent errors such as **if** $a+b$ **then**. However, given that an attempt to pass such an erroneous program into *nfunction* would result in an undefined outcome, Landin is happy to skip over such context-dependent error checking. Some tricks were needed to cope with certain problematic features of ALGOL, such as the sequencing of imperative programs. Another parameter was added to the semantic function to indicate the next imperative following the one currently being treated. This would be the next statement in a compound statement, or in the case of a jump, a program closure. Landin took pains to point out the precise language features causing these complications, which aligned with his goal of using a semantic description technique to improve languages.

The semantic function was the last thing presented in the paper; there was no formalism of the mechanism to extract the final value or meaning out of the last resulting IAE. It could be argued that this means the formalism is not quite fully complete, but that depends on the reader's desire for the use of the semantics. For someone primarily interested in a theoretical framework for considering languages, as Landin stated was his main aim, this would be plenty of information. However, a compiler writer would find there to be a few places in which more work yet remained.

Altogether, the 'Correspondence' paper showed a much more solid realisation of the ideas first present in 'Mechanical evaluation'. The (almost) complete formalism showed the feasibility of the ideas previously described in much vaguer language, and leads to an overall more believable method. This is reflected in the presentation of the paper, which was written with considerably more optimism and assuredness. The paper was, however, reasonably lengthy, with about six pages of solid formulae and a lot of technical depth, even despite the compact notation. The choice to split out the informal and formal descriptions was made very deliberately as it

left the reader the choice of precisely how much detail to consume. Despite this, as mentioned earlier, Landin did still hope for a more exact account to appear, but that was not to come: Landin's next piece was on the application of his method to language design.

3.2.5 Tackling language design

Shortly after attending the *Formal Language Description Languages* conference, Landin left the employ of Christopher Strachey, himself already taking steps at this point to wind up his consulting business (Campbell-Kelly 1985, p. 34). Landin moved, with his wife and two young children, to New York, working for the Sperry Rand corporation, in the Systems Programming Research Department of UNIVAC (Landin 1967). Here, Landin was involved in designing experimental programming languages and their processors.

One output of this time was a paper called 'The Next 700 Programming Languages' in which Landin took his experience in specifying aspects of programming languages and applied them to questions of language design (Landin 1966b). Confusingly, this was printed in 1966, the same year as two of his earlier papers (one given at the 1963 summer school and the other at FLDL), although the three represent quite different stages in Landin's work. Bornat (2009a, p. 394) described the paper as a "witty account of how *all* programming languages of the time were just sugared versions of the lambda calculus", going on to praise it as a landmark in the development of programming languages.

Landin delivered the paper at an ACM conference on *Programming Languages and Pragmatics*, held in California on 8–12 August 1965. The audience, as listed in the welcoming remarks, included such luminaries as Alfonso Caracciolo di Forino, John Carr, Bob Floyd, John McCarthy, Peter Naur, Christopher Strachey (now himself having moved on to MIT), Tom Steel, and Heinz Zemanek (Forsythe 1966). The author of the introduction placed the conference in a line of succession from the 1963 *Working Conference on Mechanical Language Structures*,[45] described as being about syntax, and the 1964 *Formal Language Description Languages* conference[46] on semantics. The latest entry, then, was to concern *pragmatics*; however, most of the authors presenting seemed to interpret that as meaning 'practicalities', and most of the papers were focused on practical aspects of programming, compilers, and op-

[45] Discussed primarily on page 50 of the present work.
[46] Discussed in Chapter 4.

erating systems. This upset one member of the audience in particular, Julien Green, who complained that the only paper given which actually concerned pragmatics was Zemanek's (Woodger and Green 1966).

Landin's paper, however, was firmly in the 'programming languages' part of the conference name, and opened with a quotation:

> "... today... 1,700 special programming languages used[47] to 'communicate' in over 700 application areas."—*Computer Software Issues*, an American Mathematical Association Prospectus, July 1965.
> (Landin 1966b, p. 157).

The implication is, of course, that this is far too many! Landin's answer to that was to introduce another language, which he called, typically irreverently, ISWIM, standing for 'If You See What I Mean'. In tacit reference to the large number of problem areas, one aspect of ISWIM that Landin mentions as a positive is its domain-agnosticism.

ISWIM was a purely functional language, which is to say there are no procedures. The language was not defined in terms of any machine or problem domain, which Bornat (2009a) notes was particularly unusual for the time. Indeed, Landin was explicitly clear that ISWIM was not a language intended for implementation. Rather, ISWIM was presented in terms of Landin's previous work: AEs and abstract machines. It is likely to have been the difficulty in trying to fit imperative features in this framework which prompted Landin to speculate about languages without such features.

Landin explained that programming languages comprise essentially of a set of basic objects and a set of ways to combine them. The former are controlled by the language's problem domain (or constrain the language's applicability to certain domains); and the latter describe the language's features. The presentation of ISWIM is focused on the latter, making it a language family: modifications could be made to the basic objects to allow variations of ISWIM to be used in a number of contexts. Landin pointed out, however, that essentially all languages of the time were language families due to local differences across implementations. The incompleteness of the ISWIM system, then, rather than being a disadvantage, instead suggests that it could form the basis of a programme for research. "A possible first step in the

[47] Sic.

research program," noted Landin (1966b, p. 157), "is 1700 doctoral theses called 'A Correspondence between x and Church's λ notation' ", going on to add that there would be no further λ symbol in the paper, although a number of lambda-esque ideas would permeate it. A possible subtitle for the paper, he joked, is "Church without lambda".

Landin's preference not to produce a full language but instead discuss *concepts* of languages was a sentiment shared by Hans Bekič of the IBM Laboratory in Vienna. Writing his thoughts on the best focus for IFIP Working Group 2.1 at its 10th meeting, he said:

> I think the most important thing is the analysis of concepts which are expressed by programming languages. One extremely important example of how this can be done has been given, for Algol 60, by P. Landin. [...] Which are the concepts present in such languages, how are they interrelated, which are the primitives in terms of which the other ones can be expressed? I think this analysis is even more important than formal description and has to precede it.
> (Bekič 1968b)

An interesting element of 'Next 700' is the lengthy comparison to LISP. Landin explained there were many similarities between the languages, but ISWIM aimed for more abstraction, and clearer notation.[48] This detailed discussion of LISP clearly showed Landin's respect for McCarthy, as he explicitly praised the logical properties underlying LISP's notation, specifically that the denotable objects of LISP are very close to those of mathematics and logic. This made the creation of equivalence rules between LISP programs straightforward.

This was a very important aim for Landin with his language. In ALGOL 60, there are a number of aspects which are almost but not quite equivalent, such as the scoping of blocks and procedures; these differences seemed to Landin to be most likely accidental, something which seemed to irritate him. ISWIM, on the other hand, has very deliberate equivalence classes, and easy-to-use equivalence rules. He went so far as to present a formalisation of a large quantity of these rules, giving them names which are simple single letters with decorations. The style was deliberately based on Curry's *Combinatory Logic*, showing Landin attempting to couch his work

[48] An aspect of ISWIM's notation should be mentioned, as well: Landin liked the idea of using indentation to indicate phrase structure, something he called "the offside rule" (Landin 1966b, p. 160).

as extending from that revered figure, and demonstrating again Landin's pedigree in mathematics. The point of these rules was to allow an easy rearranging towards a standard form that has a simple left-hand side[49] which is precisely the definee in question, and a complex right-hand side.

One such rule is:

$$L \textbf{ where } x = M \equiv \mathrm{Subst}_x^M L \qquad (\beta)$$

This is essentially a formalisation of the copy rule, as discussed in Section 2.3. Landin points out the importance of having such a rule: it allows use of the Church-Rosser theorem, which is to say that all instances of **where** can be removed from a text. This would be of particular use, argued Landin, if the functional programming style became popular. Another rule which is presented is the Y combinator; again an important throwback to Curry.

The ISWIM style, especially repeated application of the β rule, tended towards expressions with lengthy right-hand sides, as mentioned. Landin had expressed a preference for complex right-hand side expressions in discussion of his paper at FLDL: "I am sort of partial to long right-hand sides, and I would like to have a program written as a single print statement" (Landin 1966c, p. 291). This was somewhat in vogue at the time: Barron et al. (1963, p. 134), in their report on CPL, noted that their language "exemplifies the trend towards more complicated expressions embodied in a very few basic command forms".

Landin concluded his paper with a few remarks. Languages, he said, have many variations, but do these always serve to help their users, or are they relics of tradition in programming? Landin argued that we should aim to study language *families* and their properties, so that when a new language is needed it can be chosen from a familiar area and not simply constructed from scratch. He identified three techniques in particular from formal description which could be levied to aid in language design: "abstract syntax, axiomatisation, and an underlying abstract machine" (Landin 1966b, p. 164). The success of 'Next 700'[50] is testament to Landin's assertions that language design can benefit immensely from formalism, and perhaps went some way towards assuaging McCarthy's challenge to Landin at FLDL about

[49] Left-hand of the assignment symbol.

[50] At time of writing, Google Scholar records nearly 800 citations; of Landin's papers, only 'Mechanical evaluation' has more (with 1200).

the utility of his approach.

Following each talk at the *Programming Languages and Pragmatics* meeting, and the conference generally, there was some discussion which is recorded in the *CACM* article containing the proceedings. The general summaries, like the majority of papers, mainly focused on practicalities (Woodger and Green 1966). The summary worth mentioning is that provided by Carr (in Woodger and Green 1966, p. 222), who wrote about the difficulty of marrying declarative and imperative languages—clearly he was not convinced by Landin's attempts to do so. He argued that what was necessary was, instead, a way to carefully specify the current universe and the desired end universe after execution of the program. Abrahams (in Woodger and Green 1966, p. 223) thought it might be a good idea to distinguish between declarative and imperative language aspects and define them through separate means; but Strachey dismissed the idea, explaining it would be too difficult to separate the ideas cleanly.

In the discussion after Landin's paper specifically, a large part of the discussion was taken up by Strachey (in Landin 1966b, p. 165) espousing the virtues of purely declarative languages. He echoed Carr's general thoughts about their utility in specifying programs in terms of their desired outputs, and the much easier formulation and discharge of proofs. Strachey acknowledged, however, that the universal applicability of such languages was as yet undetermined at that point.

What remained of the discussion of Landin's work was dominated by arguments about the practicality of using indentation to indicate structuring when printing programs, as named the 'offside rule' by Landin. This, combined with the intensely practical nature of the conference, must have irked Landin, who was always considerably more interested in the theoretical aspects of computing.

3.2.6 Years of wandering

In 1966, Landin left UNIVAC, having found the corporate lifestyle not to be to his taste (Bornat 2009a). He went to work at MIT for a year, at their Project MAC (essentially the computing laboratory), courtesy of an introduction from Strachey (Landin 2001). There Landin began to focus more on what would become the greater passion for the rest of his working life: teaching. He was involved in both "design and implementation of a pedagogic computing system" and "teaching and planning an experimental course in 'Computer Sciences' " (Landin 1967).

However, he had not totally left research behind: later Landin (2001) reminisced

about the atmosphere at MIT regarding research. He was playing around with Tarski-style semantics linking identifiers and meanings, but came up against some hostility from Marvin Minsky, who was annoyed that researchers were being awarded grants in 'ALGOL-like languages' but really were studying language theory and automata. Minsky thought Landin was like the others, and simply putting the word 'computer' into his bids to get extra money, so he invited a friend of his to "defeat" Landin: the logician Dana Scott. This was the first meeting between the two, and Landin attempted to convince Scott of his desire to ease proving equivalence of programs. At first, he said, Scott didn't know very much about programming practically, but he was able to very quickly get a good grasp of the subject to put together with his immense theoretical knowledge (Landin 2001). This could have been Scott's first proper introduction to programming, although in 1965 McCarthy (1965) wrote that he had managed to get Scott interested in the mathematical theory of computation; nevertheless, it continued to be an interest to the logician throughout his career.

Sadly, Landin ended up not enjoying MIT too much either, being used to a culture of collaboration and "loud pub discussions"; At MIT, people were much too secretive about their work (Bornat 2009a). So Landin began looking around for an academic place in the UK. Another reason for Landin's desire to move away from the USA could be the difference in the approach to computing. McCarthy aside, most of the computing researchers in America were more interested in practicalities—witness the nature of the papers and discussion at the *Programming Languages and Pragmatics* conference held in the USA, so different in tone to FLDL, held in Vienna—which did not fit with Landin's style at all.

At the end of 1967, Landin was awarded a readership at Queen Mary College, University of London (Landin 1967). There, he had a primarily lecturing role; his teaching topics included determining the equivalence of flowcharts, scoping of variables, and lambda calculus methods (Landin, in Walk 1970, p. 3). In particular, he was interested in developing a course based on his work on distinguishing parts of languages and deciding what to do with them: theory-focused teaching was Landin's aim (Landin 1967). Burstall (2017) remembered that his friend cared about teaching a lot, and was very good at it. Landin himself recalled that he ceased to publish papers after 1970, and so fell foul of the new 'managerial' system of university departments which required frequent research assessments (Landin 2001).

Towards the end of the 1960s, Landin's academic work slowed right down. He hosted a number of important guests at Queen Mary, including Hans Bekič and John

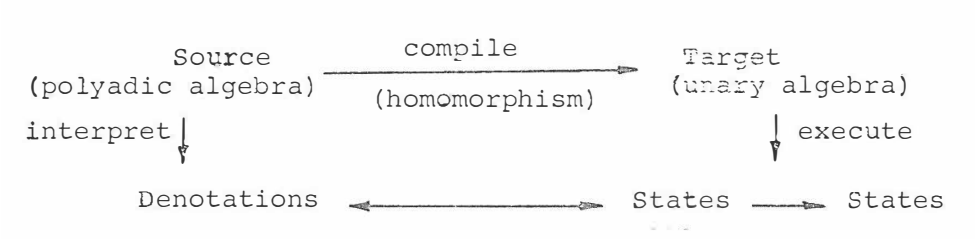


Figure 3.5: Commutative diagram showing the algebraic transformation approach to correctness.

Reynolds. Bekič visited for a period towards the end of 1968, giving fortnightly seminars on the subject of formal description of programming languages (Landin 1968b); Reynolds was a Senior Research Associate for one year 1970–71 (Reynolds 2012).

The last research on which Landin worked was in collaboration with Rod Burstall on the topic of program verification; they published a piece in Edinburgh University's *Machine Intelligence* report series entitled 'Programs and their Proofs: an Algebraic Approach' (Burstall and Landin 1969). This drew on Landin's background in universal algebra as well as Burstall's 'structural induction' technique,[51] in a response to McCarthy's use of 'recursion induction' to prove the correctness of compilers (McCarthy and Painter 1967). A very intensely technical paper, Landin (2001) described the approach later as "commutative diagrams of machines simulating each other and proving that they were". The core idea was to create a series of interpreting machines for expressions, starting very abstract, express those as algebras, and link them to each other with proved homomorphisms. Then the final machine, a store/program machine, would be provably correct because of the properties of the homomorphisms proved at each point along the series. A diagram of this central idea was presented by Landin in an IFIP WG2.2 presentation, and is shown in Figure 3.5 (Walk 1970, p. 41).

In that talk, Landin began to go on to extend the idea to look at polyadic systems with loops and branches. The idea was expressed in a wonderful analogy: directed coloured graphs are drawn on a plane, and another graph is drawn on a ball which rolls along the plane, matching the colours; this motion describes the path of computation (Walk 1970, p. 43). The notion was rather complex: Elgot (in Walk 1970, p. 45) commented that he "really can't understand Landin", and even Scott, at the same meeting, seemed a little hazy on the details. No more detail on Landin's al-

[51] See Burstall (1969a) and discussion in Section 7.4.

Figure 3.6: A photograph of Peter Landin taken around 1970.

gebraic work will be given here, as this approach moves away from the core notions of semantics which concern this story, and the algebraic approach was something of a dead end for a long time.

Landin continued to attend the WG 2.2 meetings until 1970. He tended to be less interested in presenting work of his own and instead encouraged comparisons of existing description techniques; one contribution was a short but very telling comment about the best way to approach writing programs:

> Presently, programmers proceed from fast, incorrect programs to correct programs, maintaining speed. A less dirty way would be to proceed from slow, correct (clean) programs to efficient programs, maintaining correctness.
> (Landin, in Walk 1969b, p. 14)

Indeed, at that time he moved away from working in the theory of computing generally (Landin 2001). The story goes that he was attending an examination of a PhD candidate as an external examiner, exclaimed that computer science had become "too theoretical", and left (Danvy 2009a). Landin retired from his position at Queen Mary, becoming emeritus, and for the next 40 years had only limited involvement, although he did continue to teach and supervise (Gaboury 2013). See Figure 3.6 for a photograph of Landin likely taken around 1970.

One exception to this period of self-enforced distance came in the middle of the 1990s. Olivier Danvy noticed that Reynolds' (1993) work on the discoveries of con-

tinuations missed out any contribution from Landin. Danvy thought that Landin's J operator was a crucial part of the story, and invited him to give a keynote speech at the ACM workshop on continuations in 1996, in Paris. Landin attended, and wrote a wonderfully idiosyncratic paper for the proceedings entitled 'Histories of Discoveries of Continuations: Belles-Lettres with Equivocal Tenses' (Landin 1997). This shows Landin was not quite as removed from the computing research community as people may have thought. Danvy also found amongst Landin's papers (after his death) some notes prepared for the talk, which show Landin was indeed keeping up with developments in certain areas of computing, and even anticipating some of the results that Danvy himself later put together about Landin's SECD machine and its relationship to continuations (Danvy 2009a). This notable exception aside, however, Landin did not continue to involve himself with the world of theoretical computing.

3.2.7 Retirement

Peter Landin was always a very modest man, describing himself as "one of those people that has never really understood anything unless I think I invented it myself" (Landin 2001). This was visible in a number of the self-deprecating remarks he made in his own papers. Although he was (quite rightfully) invited as a guest of importance and asked to speak at a seminar called 'Program Verification and Semantics: the Early Work', held at the Science Museum in London in June 2001, he did not think himself worthy of attendance. He even went so far as to say, during the speech he eventually gave,[52] that he thought he must only have been invited through cronyism (Landin 2001). Landin did not even attend the *Mathematical Foundations of Programming Semantics* XIV conference in 1998 at Queen Mary, despite it being held in his honour; this led Danvy (2009b) to jokingly refer to Landin as "the Bourbaki of Computer Science". Danvy (2009a) added that he had been told Landin declined to attend because he felt they had gathered "to bury him".

Having moved away from computing, Landin spent the latter part of his life becoming involved in the gay rights movement: he joined the Gay Liberation Front as it emerged in the early 1970s (Gaboury 2013). A bisexual all his life, Landin split amicably from his wife in 1973, but continued to be involved in the life of her and their two children (Bornat 2009a). Landin's house in Camden became a hub for the

[52] Cliff Jones, who organised the seminar, remarked that he was unsure whether Landin would actually speak right up until the moment he stood up. He then proceeded to only speak for half his allotted time. From personal communication with Jones, 2016.

gay rights scene in London, with *AIDS! The Musical* being composed there, and the resurrection of the Pride marches in the 1980s planned within its walls (Bornat 2009b). Landin had always been concerned with social justice, however: Barron (1975) mentioned an occasion during the time of Strachey's consultancy that Strachey "had to stand bail for his assistant [...] who was given to sitting, in the company of others, on the public road in front of American air bases".

Olivier Danvy (2009a) reported that he attended Peter Landin's memorial ceremony, held on June 20th 2009,[53] at the Hampstead Quaker Meeting House in London. There he saw an uneasy coming together of the two main groups of people in Landin's life: the computer scientists, and the activists. He likened this experience to memorial ceremony for Jean van Heijenoort, who had been a bodyguard for Trotsky before leaving him shortly before his death and going into mathematical logic; this led to two very different groups at Van Heijenoort's remembrance, both wishing to claim him for their own. Danvy explained that the memorial was filled with people describing him: he had "a complete presence" and an "inspiring ability to clarify", according to activist friends of Landin—descriptions with which the computer scientists could certainly agree. Landin's brother-in-law said that despite him being "outrageously loud and enthusiastic", he had an "unusual ability to listen", and "told the truth". One thing which did surprise some of the computing people was that Landin had no laptop and no mobile phone.

But this was reflective of how Landin's views on computing had shifted over time towards concern for the impact it was having on society. At the 2001 seminar, he voiced his worries that government regulators attempted to enforce things about software without the requisite knowledge, and the problems that would bring in a world increasingly governed by software (Landin 2001). He gave an example, which nicely illustrates a little of his character as well: certain ATMs near Landin's house in 2000 had a problem. They did not beep when digits of one's PIN were entered, and they did not show the last asterisk appearing on screen even though it had been entered. This caused all sorts of trouble, of course, and Landin entered into a long correspondence with the company who owned the machines until they eventually conceded that Landin was right, and the fault was simply a minor programming error. He was concerned by growing encroachment of technology in the social sphere, as well: in 2002, Landin (2002) described Burstall as "never appalled by widespread apathy concerning the growing absorption into the surveillable of peoples' lives"

[53] Dating gathered from Gaboury (2013).

(the implication being that Landin himself was very concerned indeed).

In interview, Burstall (2017) echoed this:

> He told me he didn't think computing was a good idea. Socially, it might be deleterious, because it would stop people talking to each other. Instead of talking to the man behind the counter in the grocery shop, you would go through a machine in the turnstiles, and you'd push the money in, and you'd be all done in half a minute or so. He was dead right.

Understandable though all these concerns may be, it was a real loss to theoretical computing when Landin decided to retire. Nevertheless, in his ten years of activity he made a real impact, especially on his erstwhile boss Christopher Strachey. A number of Landin's ideas were very important to Strachey's work, such as the lambda calculus as a tool for modelling computation,[54] the use of dynamic lists, the Y combinator, and many more. Landin's work was also highly influential on the Vienna group, whose abstract machine for interpreting PL/I was heavily based on Landin's SECD machine.

3.3 Adriaan van Wijngaarden

Another important and influential scientist who worked on programming language description in the early 1960s was Adriaan van Wijngaarden (1916–1987). Van Wijngaarden, or Aad as he was known to friends, was a strong force in the computerisation of the Netherlands, and is often referred to as the 'founding father' of Dutch computing (Alberts 2016).

3.3.1 Building computing in the Netherlands

Van Wijngaarden attended the Gymnasium Erasmium in Rotterdam—the same school his future colleague Dijkstra would attend—and then studied mechanical engineering at the Technical University of Delft, graduating in 1939 (Dijkstra 1980; Pérez 2017). In 1946, van Wijngaarden was sent to England by Delft to travel around and learn the state of their mechanical engineering and mathematical developments, information not shared due to the war. As a result of that visit, van Wijngaarden became convinced that computing machines were worth further study (Alberts 2016).

[54] The question of who inspired Strachey to use lambda calculus will be explored more in Section 6.2.

When the Mathematisch Centrum (MC), an institute for research in mathematics and engineering in Amsterdam, was set up in 1946, van Wijngaarden was asked to be the head of the new computing department (Pérez 2017). He accepted, and went on to become head of the whole organisation in 1961 (Belgraver Thissen et al. 2007a),[55] remaining at MC until his retirement in 1981, two years before its name change to Centrum Wiskunde & Informatica.

Upon establishing the department at MC, van Wijngaarden hired a possibly surprising group of employees: twelve young women with strong mathematical backgrounds. They performed complicated calculations, checked results from other sources, programmed computers, and assisted in the development of the ARRA II, the Netherlands' first fully-electronic computer (Belgraver Thissen et al. 2007b). They were known as 'computing girls', 'van Wijngaarden's girls' or both together as in Dutch: '*de rekenmeisjes van Van Wijngaarden*'. Sadly, few of these women stayed in computing as the group grew, and many left to start families and never returned.[56]

In 1949, van Wijngaarden made another journey to the United Kingdom, to Cambridge, where he made an interesting connection (Jones 2017, p. 39). There was a conference demonstrating the new EDSAC machine held on 22–25 June, at which Alan Turing spoke about his 'Checking a large routine' paper (Turing 1949). In this, Turing used assertions to decorate flowcharts of programs to help prove their correctness. Van Wijngaarden did not, however, appear to take any inspiration from this talk, and never referred to it. The only evidence we have of his attendance is that his name appears in the proceedings of the conference. However, Jones (2017, p. 39) reports that Gerard Alberts believes that van Wijngaarden was not interested in logic so much as numerical analysis, and "has evidence that van Wijngaarden 'had little affinity' with Turing and his work".

Van Wijngaarden began an important and lengthy partnership in 1952 when he hired Edsger Dijkstra to program the Centre's ARRA machine[57] (Alberts and Daylight 2014, p. 56). According to Dijkstra (1980), van Wijngaarden had been im-

[55] The sharp-eyed reader will note the less academic nature of this source: a website about the history of computing in the Netherlands. However, its use is justified because it presents information otherwise available only in Dutch-language sources, which furthermore correlates with that described in other, more reliable sources.

[56] This was not an exceptional case: in the early era of computers, many women performed similar work, and were similarly shut out of careers by the career structuring. For a very thorough discussion of the gendering of the computing workforce, with the United Kingdom as an illustrative example, see the work of Mar Hicks (2017) in *Programmed Inequality*.

[57] *Automatische Relais Rekenmachine Amsterdam*, Automatic Relay Calculating Machine Amsterdam. An electro-mechanical computer.

pressed by his "technique for converting from decimal to octal with a mechanical desk calculator". By 1955, Dijkstra had been programming for three years part-time while studying theoretical physics at Leiden, and was uncertain which career path to follow. Dijkstra went to ask van Wijngaarden for advice:

> Full of misgivings I knocked on van Wijngaarden's office door, asking him whether I could speak to him for a moment; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently, he agreed that up till that moment there was not much of a programming discipline, but then he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come? This was a turning point in my life and I completed my study of physics formally as quickly as I could. One moral of the above story is, of course, that we must be very careful when we give advice to younger people: sometimes they follow it!
> (Dijkstra 1972, p. 860)

Together with his protégé, van Wijngaarden worked tirelessly to develop computing in the Netherlands, with a particular focus, in his earlier career, on applications in engineering. Isaac Auerbach (1986b, p. 61), founding president of IFIP, remembers that van Wijngaarden first became involved with the Federation because he wanted to use computers to help with "the development of mathematical models of dikes for his country".

At a joint GAMM–NTG[58] symposium held in Darmstadt in October 1955 on *Electronic Digital Computers and Information Processing*, van Wijngaarden presented the results of a survey on scientific computing in the Netherlands, which suggested that country had the highest number of computers per capita of any European country—although with some scepticism (Zemanek 1981, p. 2). Despite this, computing was not immediately embraced in the Netherlands, with Dijkstra explaining that, upon his move from physics into computing, "the mathematicians, one of whom openly prided himself on 'of course knowing nothing about computers', were just contemptuous" (Dijkstra 2001b, p. 342).

Perhaps as a result, van Wijngaarden was always very keen to drive international computing efforts, and place the Netherlands firmly on the world stage. Alberts attributes this to the efforts of van Wijngaarden's tutors: one, Jan Burgers, had

[58] Two German societies for mathematics and engineering.

tried to set up a Northern European collaborative group in 1948 with little success. The institutions he had asked preferred to instead set up collaborations through an association of national societies: the Dutch society, *Nederlands Rekenmachine Genootschap* (Dutch Computer Society), was established in Spring 1958 with van Wijngaarden right at its centre (Alberts 2016). He was also involved in IFIP since its very beginnings, being the vice president of the first International Congress and taking on a lot of the organisational responsibilities (Zemanek 1981, p. 3). Once IFIP was established as an organisation, van Wijngaarden was one of the two vice presidents under Auerbach (the other was Anatoly A. Dorodnitsyn). In typically flowery language, Zemanek (1981, p. 4) described van Wijngaarden's attitude to international academic relations:

> Professor van Wijngaarden is an admirable example for all of us; in a seafaring country like Holland you might be reminded of a ship's figure-head, a smiling, mythical beauty who is constantly ahead of the crew and the passengers buried in the entrails of the ship.

Van Wijngaarden had spent the 1950s building up computing at MC in Amsterdam, working on making computers and arranging their programming, primarily for applications in mathematics and engineering. An example paper is 'The state of computer circuits containing memory elements', given in Cambridge, MA, in 1957 (van Wijngaarden 1959). But his life was changed dramatically in August 1958, while in Edinburgh to speak at the International Congress of Mathematicians. Van Wijngaarden was involved in a very severe traffic accident which caused him to spend three months in hospital, and his wife, Barbara Robbers, died. Alberts speculated[59] that van Wijngaarden may have blamed himself for the accident, and deliberately tried to break away from his previous life by shifting his attention as much as possible. This included going so far as to cut off friendships previously held, but importantly for the present story, it signalled a shift of focus in van Wijngaarden's work towards programming languages.

3.3.2 Generality in programming languages

After the publication of ALGOL 58, a number of people made suggestions for the refinement of ALGOL via the *ALGOL Bulletin* and other venues. Although neither Dijkstra nor van Wijngaarden had been part of the committee that designed

[59] In a personal communication, May 2017.

Figure 3.7: Alan Perlis (left) deep in conversation with van Wijngaarden. Taken from *History of Programming Languages* (Naur 1981b, p. 155).

ALGOL 58, both contributed extensively to improvement discussions, and attended many of the meetings that led towards the ultimate creation of ALGOL 60 (Naur 1981b, pp. 110–3). These suggestions included the extension of the ALGOL typing system to include more mathematical types such as vectors, and parameter mechanisms, demonstrating two of the aspects that would later become of great importance to van Wijngaarden when he came to design his own version of ALGOL. Alberts and Daylight (2014, p. 56) describe the attitude the two Dutchmen took to programming languages:

> Van Wijngaarden and Dijkstra viewed a programming language such as Algol as a mathematical object. Only after it met certain aesthetic criteria would they consider the language to be relevant. An important aesthetic criterion for the Amsterdam team was generality: in the interest of obtaining a simple language, unnecessary language restrictions were best avoided.

Van Wijngaarden's contributions to the development of ALGOL led to his being included in the group of 13 who met in Paris in early 1960 to decide on the final shape of ALGOL 60. Figure 3.7 shows a photograph of van Wijngaarden taken at this meeting. One particularly important contribution he made came a month later: Dijkstra and van Wijngaarden decided that it was vital that one particular

'unnecessary language restriction' was avoided: recursion must not be kept out of the language.[60]

Alberts and Daylight (2014, p. 51 (footnote 27)) explain precisely how this was achieved:

> In a telephone call in February 1960, van Wijngaarden convinced Naur to explicitly state that the recursive procedure should not be ruled out of the official definition of Algol, much to the dismay of Bauer and Samelson.
> Asked why recursion was such a big issue to the Amsterdam team, van Wijngaarden responded in terms of ethos, "a matter of honour and intellectual decency" ("een kwestie van eer en geestelijk fatsoen").

As mentioned in Section 2.3, there was already a recursive element implicitly present in the language, so this insistence on its explicit inclusion is a reflection of the strength of feeling the two Amsterdammers had for the concept. This is particularly interesting given that recursive procedures did not receive recognition as practically useful until after 1960 (Alberts and Daylight 2014, p. 57), and many researchers, particularly those in the German camp like Bauer and Samelson, strongly objected to its inclusion for reasons of efficiency. The episode stands as an illustration of van Wijngaarden's attitude to languages: power and flexibility was more important than ease of implementation or efficiency (Alberts and Daylight 2014, p. 56).

After the development of the Mathematisch Centrum's new Electrologica X1 computer at the end of the 1950s, Dijkstra and Jaap Zonneveld wrote an ALGOL 60 compiler for the machine (Kruseman Aretz 2003). The compiler was instantly well-regarded due to its quick construction and immediate correctness, and was interesting for its two-stage process. First, the ALGOL 60 program was translated into an 'object language', which was machine independent, and then this was translated into the X1's operating code. Dijkstra and Zonneveld implemented the whole of the language using this, admittedly rather slow, process, which contrasted with the

[60] Dijkstra was such a fan of recursion, he even produced the following story in a reflective piece written many years later:

> A few years later [than the publication of ALGOL 60] a five-year old son would show me how smoothly the idea of recursion comes to the unspoilt mind. Walking with me in the middle of town he suddenly remarked to me, "Daddy, not every boat has a lifeboat, has it?" I said "How come?" "Well, the lifeboat could have a smaller lifeboat, but then that would be without one."
> (Dijkstra 2001a, p. 5)

Bauer-led ALCOR implementation team whose compiler tackled only a subset of ALGOL 60 in pursuit of greatest efficiency (Alberts and Daylight 2014, pp. 57–8).

Samelson, a close colleague of Bauer,[61] commented later on this distinction, explaining that in the ALGOL design circles there were two main groups: the "restrictionists" or "safetishists", and the "liberalists" or "trickologists". Samelson (in comments on Naur 1981b, p. 132) put himself in the former camp and van Wijngaarden firmly into the latter.

Another way to describe the approach followed by van Wijngaarden and those who thought like him is 'generalist'. This is vital to understanding his approach to programming: restrictions should not be made that dampen the equal treatment of concepts throughout a language. Niklaus Wirth (1963, p. 548) agreed with the idea, and thought of it as the core concept of high-level languages. The idea was to save programmers from "the myriad of tedious detail considerations required for machine-language coding"—and ALGOL 60 represented this perfectly. Wirth also pointed out, however, that some efficiency might be lost in the ultimate running of the program, which could lead some to balk at the prospect. Alberts and Daylight (2014, p. 59) explained:

> No one was as radical as van Wijngaarden and Dijkstra in choosing *generality*[62] as a starting point. Perhaps, it was the bliss of being late entrants in the Algol community, and not sharing a long tradition of implementing, that allowed them to make a radical choice. More likely, it was the mathematical style of Amsterdam and its Mathematisch Centrum that had prepared van Wijngaarden and his team to think in terms that were free from a "commercial capture."

Van Wijngaarden indeed had a strong mathematical focus and pedigree, which always influenced his work. This sentiment was echoed by Łukaszewicz (1986, p. 296), a Polish representative to IFIP:

> Officially he was the representative of Holland but for me he stood above all for the international community of mathematicians. At our sessions he spared no efforts to see that our rules and resolutions were as rigorous and complete as possible.

[61] In Samelson's obituary, Zemanek (1986c) commented "He cooperated very closely with Professor Bauer (who lost in him his most important friend) [...] It is a nice symbol for their cordial cooperation that once somebody asked me in deep wonder: 'Bauer–Samelson are two people?' "

[62] Emphasis added

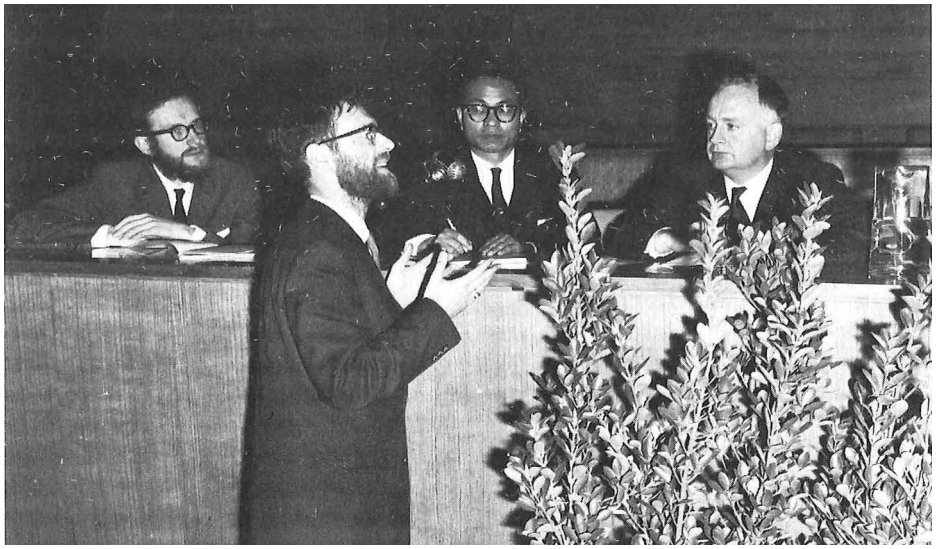


Figure 3.8: Edsger Dijkstra speaks at the ICC *Symposium on Symbolic Languages in Data Processing*, 1962. In the background, Naur (left) and van Wijngaarden (right) watch on. In the centre is an unknown session chair. Photograph from Auerbach (1986a, p. 75).

Zemanek (1981, pp. 1–2) also remarked on the importance van Wijngaarden attached to generality, but added that he also had the ability to be highly specialised:

> From the very beginning I have sensed the dual character of his unique personality: the large mind which has always extended beyond my horizon, and the sharp brain that can suddenly focus on the smallest detail, but will illustrate by it some general aspect; the 'generalizer' who generalized even a general purpose programming language, and the 'specializer' whose production of sentences and questions has often reminded me of a pencil sharpener.

3.3.3 Generalising a general-purpose programming language

The activity to which Zemanek referred was the first important work presented by van Wijngaarden on the subject of describing programming languages: a paper called 'Generalized ALGOL' (van Wijngaarden 1962). This was read at the IFIP organised *Symbolic Languages in Data Processing* conference held in Rome in March 1962, the year Dijkstra left MC to move to Eindhoven. A photograph of van Wijngaarden and Dijkstra at this meeting is shown in Figure 3.8.

The 'generalising' referred to in the title meant taking the concepts of the language

and making them universally applicable across the entire language. This was somewhat similar to Landin's idea of 'fixing inconsistencies', for example the treatment of functions and parameters. The language in question was not quite the same as ALGOL, but had many similarities to that language.

In the introduction, van Wijngaarden (1962, p. 17) explained what he meant by a 'general language':

> The main idea in constructing a general language, I think, is that the language should not be burdened by syntactical rules which define meaningful texts. On the contrary, the definition of the language should be the description of an automatism, a set of axioms, a machine or whatever one likes to call it that reads and interprets a text or program, any text for that matter, i.e. produces during the reading another text, called the value of the text so far read. This value is a text which changes continuously during the process of reading and intermediate stages are just as important to know as the final value. Indeed this final value may be empty.

Similarly to many of the other people interested in semantics in the present story, van Wijngaarden was uninterested in focusing on the syntax of programming languages. Indeed, Dijkstra (1961b) explained that his colleague's approach to fully defining semantics obviated the need for a formal syntax altogether:

> When we have thus defined our language, its semantics are completely fixed and its syntax—I owe this remark to Prof.dr.ir. A.van Wijngaarden—does not have a defining function anymore: we can do without the syntax as it is merely a summary of "admissible constructions", i.e. all constructions to which the machine does not produce the uninteresting reaction "Meaningless".

The core concept of moving away from syntax bears similarities to the ideas of McCarthy and Landin: a shared interest in determining some 'value' of programs and languages. However, the way van Wijngaarden approached the problem was very different. He chose not to use an abstract syntax so the semantics is not based directly on the structure of the language, but rather its textual content.

In 'Generalized ALGOL', van Wijngaarden (1962, p. 18) presented a beautiful metaphor for his approach to giving meaning to languages. Suppose you have a machine, known as the interpreter, which takes in a program and produces a result value. On the lid are presented general instructions for how to use the machine,

which should suffice for the majority of users. But if these instructions are not quite clear enough, you may take off the lid. Inside are two more machines: a preprocessor and a processor. The preprocessor takes the language of the program and converts it into a simpler, more generalised language, so the processor can then process it. Is this still not enough information? You may look inside the processor, and see another two machines... eventually, you will reach a level where you cannot look inside any further, because the language used cannot be stated any more simply. This is in terms of axioms.

This splitting of definition into preprocessing and processing allowed the discussion of languages without too much concern for their precise syntax, as any number of ways to write similar ideas (such as indexing of multi-dimensional arrays—A[1][3] vs. A[1, 3]) can be simply addressed with the preprocessor. It is important to note that all of these were discussed in terms of concrete strings of characters, not as abstract objects with their structure as the primary focus, as in the work of McCarthy and Landin.

The machine interpreter's functionality was described by van Wijngaarden as follows. A sequence **V** represents a 'list of truths' which are consulted in order when determining meaning for the program, and added to as the program progresses. These are essentially the stored results of a **value** function. This function is repeatedly and recursively applied to the program statement until a truth is found with no value function in its right hand side, i.e. some kind of constant, and this value is textually placed back into the name being interpreted. Each successfully matched truth is added to **V** for use later and eventually this method provides a final result for a program. The machine must start with a correct sequence of rules in **V**, and changing this or its order will change the functionality of the language. This also enforces such concerns as declaration of variables before their use: the **value** function would find nothing in **V** if there had been no declaration.

The construct **V** and the function **value** used by van Wijngaarden together represented something of a strange hybrid between concrete syntax rules like BNF, a semantic interpreting function, and a 'state' as in McCarthy or Landin. Some of the job of interpreting is done by the 'scanning' and 'replacing' functionalities of the pre-processor described informally in the paper, and the rest by the **value** function—all through purely symbolic text manipulation. This convergence into a single concept is emblematic of van Wijngaarden's view of everything in computing as strings of symbols: to van Wijngaarden, the program, its interpreter, and the re-

sult are all the same type of object. Scott (2018) commented on van Wijngaarden's attitude to this idea, explaining that he heard him say "If you can't do it by symbol manipulation, then it's not worth doing it". Scott noted that, alongside Curry, van Wijngaarden was the most symbolically-minded person he knew.

McCarthy, however, did not agree with this approach, and in his *Formal Language Description Languages* paper, in the section 'Comparison with other ways of describing semantics', spoke out against a highly concrete approach. Regarding ALGOL data as only "strings of symbols", and treating "the state as a giant string", he said, was insufficiently abstract, and did not correspond effectively with intuitive ideas of meaning (McCarthy 1966, p. 6). The two quoted phrases mark McCarthy's comment as being clearly aimed at van Wijngaarden.

Another interesting aspect to van Wijngaarden's proposal is that in his system, a program would start with **V** containing a list of the rules that it wants to hold within itself, allowing a good degree of flexibility to the programmer. A programmer could then write their own rules at the beginning of their program, which would be interpreted first and added into **V**. This is somewhat analogous to defining procedures, in the sense of extending basic operations, but could also allow programmers to completely change the evaluation system.

Procedures with parameters were handled in an interesting way: quotation marks demarcate strings indicating parameters called by name. The first pass of interpretation removes the quotation marks, and on the next pass, the text is then evaluated as normal. This delays the evaluation appropriately and so the 'by name' effect is achieved; note the similarity in effect (although not in method) to Landin's delaying tactic. A further consequence of this approach is that the calling mechanism can be chosen by the programmer at the call point of the procedure rather than its declaration, which gives greater flexibility and allows the mixing of calling mechanisms.

When the processor has finished, what remains in the value list **V** is the result of the semantic interpretation, which is to say the *meaning* of the program. It is a list of equivalences and deductions that can be made about the program and its components. This suggests that for van Wijngaarden, the meaning of a program is more than just the final values of the variables contained within, also those are certainly present—various parts of meta-interpreting information are also still contained within **V**. This is in contrast with the views of scientists like McCarthy, for whom the final state is what matters, and actually links more closely with Landin's

view of the ultimate meaning being the final state of the SECD machine.

In a later paper van Wijngaarden (1966b, p. 13) described the use of formalism in 'Generalized ALGOL': "The preprocessor was not formalised and may vary from case to case. The processor was formalised to a high degree and does not vary." This estimation did not completely match the content in the paper, however. There was, indeed, some formalism used in the description of the processor, but this was not entirely end-to-end, which is to say there was no precise formulation of every rule that would be required in order to fully process a program.

'Generalized ALGOL' was a radical paper, and remarkable in the power granted by solely string manipulation techniques. The really important notion was that the 'meanings' in **V**, such as that of procedures being blocks with parameter substitutions, were not meta-equivalences as described by other scientists, but direct replacement of texts, with the addition of appropriate high-level characters like the quotation marks that delineate by-name parameters. This showed the beginnings of van Wijngaarden's ideas that ultimately led into ALGOL X, ALGOL 68, and the two-level grammars used to discuss them. More on this story is discussed in Section 7.1.

Reaction to the paper was somewhat mixed. As mentioned earlier, McCarthy was highly critical, and Zemanek (1981, p. 12) acknowledged this, writing "It is a systems theory of programming languages, elegant, general and powerful, but obviously at a certain price. Not everyone is ready to pay this price, as the course of history has shown."

One person who did apply similar ideas was Wirth, who, as already described, also held generality in high regard. He wrote a paper called 'A generalization of ALGOL' (Wirth 1963) which, despite the nearly identical title, does not directly cite and use the ideas of van Wijngaarden (although he does mention the Amsterdammer in his acknowledgements). Wirth's suggestions are a lot less radical than van Wijngaarden's: largely these comprise improved language dynamism through the removal of the typing system and static bounds on arrays. One concept shared between the papers is the use of 'quotations' for procedures and call-by-name.

Van Wijngaarden's next publication was not on the topic of programming language descriptions, but is worth mentioning for a couple of reasons. 'Numerical analysis as independent science' was presented at a NordSAM[63] conference in Stockholm, Swe-

[63] *Nordiskt Symposium över Användningen av Matematikmaskiner*; Swedish: Nordic Symposium on the Use of Mathematical Machines. For more on computing in Scandinavia, see the

den, in August 1964 (van Wijngaarden 1966a). It was published in the proceedings in BIT, and then republished as an MC report. The paper contrasted 'numerical analysis' (i.e. that performed on machine) with 'pure analysis' (as performed by mathematicians). Van Wijngaarden suggested some strategies for representing on machines real numbers and operations thereon, and assessed these with concerns of efficiency. It is a very practically-oriented work and shows van Wijngaarden's depth of understanding of ALGOL, going some way to opposing those who accused van Wijngaarden of being uninterested in efficiency in computing, or of having impractical views. The deeply mathematical nature of the work echoes Łukaszewicz in showing van Wijngaarden as being a mathematician at heart.

3.3.4 A powerful preprocessor

Van Wijngaarden developed his ideas of programming language interpreting further in his paper at *Formal Language Description Languages*, 'Recursive definition of syntax and semantics' (van Wijngaarden 1966b). On the title, Zemanek (1981, p. 14) explained:

> Recursion was a key issue at that time and we teased him by proposing to him the title and official address *His high recursivity Professor van Wijngaarden*. Actually, the paper did not once use the word *recursive* except in the title.

The subject was once more a symbolic manipulation approach and elaborated on the themes of van Wijngaarden's earlier paper. Following the concepts outlined there, this paper looked more at some properties of the working of the interpreter machine, presented as an analogy in the previous paper. This was the first instance of van Wijngaarden using terms like 'metalanguage', 'metaoperator', and so on; the previous paper had focused more on the processor where this later paper devoted more attention to the preprocessor. A number of ALGOL constructs were deconstructed in the way they might be by a preprocessor. This involves the breaking down of pseudoconcepts (concepts which can be expressed in terms of other concepts) into more fundamental parts, generally done by 'deleting' parts of the program:

> The first pseudoconcept we find is the comment, which has no semantic meaning at all. Hence, in any occurrence outside strings one may delete

History of Nordic Computing series, published by Springer.

> certain sequences of basic symbols completely.
> (van Wijngaarden 1966b, p. 13)

For example, a switch statement can be replaced by a procedure which takes an integer parameter and has a series of **if** statements conditional on that integer leading to goto statements. Type procedures (i.e. those which return a value) are replaced by non-type procedures that assign to an additional variable created in the calling context. By this method—lexicographic only, and described informally—the language is broken down into a much simpler form consisting only of arithmetical and Boolean operations and assignment, with addition of procedures and locality. Van Wijngaarden actually saw this process, defining more complex language elements in terms of simpler ones, *as* semantics: "By such an intricate but still lexicographical process, one not only eliminates the function designator, but actually defines what it means" (van Wijngaarden 1966b, p. 14). The quotation also emphasises van Wijngaarden's continued viewing of meaning as linked only to strings of symbols.

A quotation from the discussion serves to explain van Wijngaarden's motivation:

> If I look at the [ALGOL] Report, I say to myself, must I define all this by the processor—all these rules? This is far too much for me! So I say, let's first take all the nonessential things out of ALGOL. Now, this is a task for the preprocessor—to look at this text and say, "I'll translate this text into reduced ALGOL and then define only reduced ALGOL".
> (van Wijngaarden 1966b, pp. 18–19)

A critical point is that the preprocessor must not—and does not—change the meaning of the program, a property that van Wijngaarden (1966b, p. 19) justified by saying "I do not change any identifier".

The paper ended with an example of decimal addition and subtraction given in these purely string manipulation terms; it is rather complicated, involving a full page of text substitution instructions. Later, in a letter to Bekič, Dijkstra (1974b) described his reaction to this approach of van Wijngaarden:

> It is practically impossible to give such a mechanistic definition without being over-specific. The first time that I can remember having voiced these doubts in public was at the W.G.2.1 meeting in 1965 in Princeton, where van Wijngaarden was at that time advocating to define the sum of two numbers as the result of manipulating the two strings of decimal (!) digits. (I remember asking him whether he also cared to define the

> result of adding INSULT to INJURY; that is why I remember the whole episode.)

Dijkstra explained that his problem with van Wijngaarden's approach was the difficulty of determining which aspects of the textual representation are essential to the meaning of the program, and which are supplementary. Dijkstra was writing in 1974, by which time he was most interested in the concerns of determining equivalence of programs, and proving program correctness. This was representative of a general move away from the semantics of programming languages as wholes, and towards the meaning of individual programs, which took place throughout the 1970s.[64] Dijkstra's own style at the time was built around proof by construction, as "programs are not 'given', they have to be 'constructed' " (Dijkstra 1974b).

However, in the early 1960s, Dijkstra's view had been rather different:

> A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it.
> (Dijkstra 1962, p. 1)

It is interesting that, in his letter to Bekič, Dijkstra referred to a later presentation of this decimal addition rather than the one given as part of van Wijngaarden's FLDL talk. Dijkstra was certainly present at the talk, and he commented on certain aspects of it, as part of a very lively discussion section: almost as much paper in the published proceedings is taken up by the transcription of the discussion as by van Wijngaarden's paper itself.

A running theme among discussants is puzzlement. Dijkstra (in van Wijngaarden 1966b, p. 20) began by saying "I am somewhat baffled, I might say, in many ways.", and he was not alone. Various participants seemed to find fundamental problems with the method, to which van Wijngaarden invariably replied that they simply were not understanding, and in some cases even getting angry. Sample comments made by van Wijngaarden include "But this has nothing to do with anything" and "Listen to my words!" (van Wijngaarden 1966b, pp. 19, 21). It is possible that the view of computation presented by van Wijngaarden was so different to that held by the other attendees that they could not see how it could be correct.

[64] This particular topic is discussed more in Section 8.1.

Many were concerned by the power of the preprocessor to change so much of the program text, including Gorn. What if, he asked, there are important efficiency considerations within the writing of the program that the preprocessor eliminates? Van Wijngaarden indicated that he was not interested in such concerns, rather in the meaning of the program. Gorn's questions were about correspondence with a real machine, and the various mechanisms on that machine that allow interpretation; this fit with his concept of a 'background machine' everyone has in their head when reading a program and imagining its execution (Gorn 1964). For example:

> VAN WlJNGAARDEN
> No—no! If you see what a goto statement means, then there is nothing between the initialization of the goto statement and where it leads. It just says that its successor has been designated to be that point.
>
> GORN
> This has to be done by a modular counter. You've lost that.
>
> VAN WIJNGAARDEN
> What is that? A modular counter, what is that?
>
> GORN
> It counts; 1,2,3; 1,2,3; 1,2,3.
>
> VAN WlJNGAARDEN
> That's outside ALGOL. It's a metaconcept I don't understand.
> (van Wijngaarden 1966b, p. 20)

Thus van Wijngaarden showed he did not share Gorn's notions of a background machine.

Dijkstra attempted to clarify the nature of van Wijngaarden's proposals about the treatment of procedures and gotos, which was arguably somewhat contradictory in places. Looking at the work as published, it is not easy to determine whether, after preprocessing, van Wijngaarden's text contains procedures or gotos. In the paper itself, he wrote that there would be procedures left as the only sequencing mechanism. But in response to Dijkstra's question in the discussion, van Wijngaarden (1966b, p. 21) said "If you have performed the procedure call, this is equivalent to insertion of a sequence of statements." and this implies that the preprocessor has some knowledge of the values of variables. This can be illustrated with an example. This mixes up static (i.e. textual) and dynamic (value) information—something McCarthy had been very careful to avoid. It seems van Wijngaarden could have been more clear about this point.

Instead, his reply to Dijkstra became somewhat heated. By this time in 1964, Dijkstra had left Mathematisch Centrum and perhaps there was some remaining bitterness around their departure from each other. This discussion proceeded as follows:

> DIJKSTRA
> I just don't understand. You have your text. You make another text that remains without procedure calls. Now, you say that somewhere or another you make the insertions; so that we do without procedure calls. Now we have only goto's.
>
> VAN WIJNGAARDEN
> What! What? What? You have only *statements*—sequences of *statements*. You have no goto's whatsoever! You have only sequences of statements, and these sequences of statements are *exactly* the same sequences of statements that would have been there in the other case.
> (van Wijngaarden 1966b, p. 21)

Another commenter was Tony Hoare, who complimented the approach, saying the completeness, precision, and even elegance of the processor is sufficiently powerful to define the preprocessor as well. Why not place these rules at the bottom of the processor? Van Wijngaarden agreed that this could be done, but liked the split between the two for psychological reasons.

Garwick asked two interesting questions. The first was about the order of evaluation of expressions and how that is affected by van Wijngaarden's removal of type procedures, given that these may have side effects. Van Wijngaarden argued that the problem is not his but a "question of undefinedness in the language" (van Wijngaarden 1966b, p. 21). This interaction highlights a role of description methods championed by Landin: the easier identification of areas of weakness or lack of clarity in a natural language description. Secondly, Garwick admired van Wijngaarden's treatment of gotos, and asked:

> Do you think that similar simplifications are possible in all reasonable programming languages? This is a very vague question and it really only requires a vague answer.
> (Garwick, in van Wijngaarden 1966b, p. 22)

Van Wijngaarden replied that it should be possible, and his reply shows his humorous side:

> Mr. Garwick, if I answered "Yes" to that I would be admitting that there are reasonable programming languages other than ALGOL. [Laughter] This was not made as a political statement. [Laughter]
> (van Wijngaarden 1966b, p. 22)

As an aside, van Wijngaarden was known to show this side at meetings on occasions. Edwin Harder, IFIP treasurer 1969–1972, described this aspect of him:

> IFIP finance has had its "interesting" moments. In the early years the Treasurer's accounts were audited annually by a committee appointed in the Council. On one occasion the Chairman of the Auditing Committee was a certain Dutchman by the name of Professor van Wijngaarden. The Secretary-Treasurer was a certain Swiss by the name of Professor Ambrose Speiser.
>
> After due deliberation the Chairman of the Auditing Committee announced that the accounts were wrong—with no explanation. There was dead silence in the room. For anyone to challenge the integrity and honor of a Swiss Treasurer, and particularly such a meticulous treasurer as Dr. Speiser, was surely an international incident of the first water. I expected a duel at the very least.
>
> Very stern glances were interchanged, but very few words. Some fifteen minutes later we were able to dig out of a certain stubborn Dutchman the fact that the discrepancy was a matter of three cents in calculating foreign exchange. It was very clear that a certain Swiss did not greatly appreciate this form of humor. Nonetheless when I reminded Professor van Wijngaarden of this incident many years later, he wrote back claiming to be still "persona grata" in Switzerland, in spite of the "international incident".
> (Harder 1986, p. 336)

Back at FLDL, Samelson asked why van Wijngaarden chose to have the procedure as a basic concept and not the goto, given that the latter seems simpler. Van Wijngaarden's reply illuminates a common problem faced by writers of programming language descriptions: there are some language features that are very difficult to handle.

> Well, because I cannot miss the procedure as a basic concept, and I didn't know how to explain the goto in my language. So I asked, "What's wrong with the goto?" I didn't know how to deal with it, so the only thing to do was simply declare it not there. OK?
> (van Wijngaarden 1966b, p. 23)

Of course, van Wijngaarden *did* handle gotos, by a careful description of text manipulation rules, but this theme—glossing over the parts of the language that were

tricky to describe—would emerge time and time again. Jumping, in particular, caused many challenges to formal description, frequently either being overlooked, or requiring very complex mechanisms in their description.

McCarthy challenged van Wijngaarden's treatment of the fundamental objects, arguing that in ALGOL 60 there are some things which are strings of symbols and others which are abstract entities like real numbers. McCarthy stated that he didn't like strings of symbols, and so his approach was to abstract away from these into an abstract syntax; but van Wijngaarden had made these strings of symbols the basis of his whole semantic system. Van Wijngaarden countered that, to him, these were the fundamental part of ALGOL:

> Now, you say that numbers are not strings in that sense. Now, I know exactly what a number is; it is a string of digits. It may be preceded by a plus or minus sign, and it may be preceded by a decimal point. There is no other thing in ALGOL that is a metaconcept called "number" of which this is the number. To me the number 13 is just the sequence of symbols 1, 3. I have never seen a "number".
> (van Wijngaarden 1966b, p. 23)

This last sentence, in particular, seems a very unusual position for a mathematician to take, as it would imply that '13' is a different entity to the base-2 '1101'. However, van Wijngaarden was more likely speaking from the perspective of his ALGOL processor than from his own as a mathematician. Van Wijngaarden discussed his constructivist views about mathematics and computing at length after Landin's talk at FLDL; this is addressed in Section 4.3.

A question from Strachey served nicely to illustrate the difference in views between him and van Wijngaarden:

> Now, the other point is, your technique has been to take all the things that people think are important in languages and replace them by all the features that everybody left out. [Laughter] That is to say, you remove things and replace them by procedures that are not function designators. I would much prefer to move in exactly the opposite direction.
> (Strachey, in van Wijngaarden 1966b, p. 24)

Van Wijngaarden replied that it was simply a matter of taste: he took the procedure as a basic concept because it allowed easy expression of a lot of different concepts.

McIlroy liked the idea of reducing a program into something simpler, especially from the perspective of a compiler writer, but he was worried by the removal of goto and morphing into procedures because it meant "the entire history of the computation must be maintained" (McIlroy, in van Wijngaarden 1966b, p. 24). This was an approach used by a number of different semantic techniques, including McCarthy as mentioned in Section 3.1, and also especially the Vienna group, as covered in Section 5. It tended to receive differing amounts of criticism based on how large the language under definition was, with McCarthy's smaller examples rather hiding the complexity that retaining a program engendered. In fact, the complaint was actually unjustified in van Wijngaarden's case, he claimed, and his reply to McIlroy provides a nice way of understanding the procedure mechanism in place:

> I suppose you have a certain implementation of a procedure call in mind when you say that. But this implementation is only so difficult because you have to take care of the goto statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure to return. That is the whole difficulty with procedure implementation. That's why this is so simple; it's exactly the same as a goto, only called in other words.
> (van Wijngaarden 1966b, p. 24)

This passage is worth quoting in its entirety because it describes the concept which later became known as 'continuation passing form'.[65]

Van Wijngaarden went on to develop these ideas further, and they became a very rich and complex strategy for defining a programming language. The eventual outcome of this technique was the new version of ALGOL, ALGOL 68, the story of which is covered in Section 7.1. Now, it is worth taking the time to consider the *Formal Language Description Languages* conference in more detail.

[65] More on the impact of continuations is discussed in Section 6.5.

CHAPTER 4

A field is born: the *Formal Language Description Languages* conference

The IFIP-sponsored *Formal Language Description Languages* conference (FLDL), held in Baden-bei-Wien, near Vienna, in September 1964, marked a turning point for the field of programming language descriptions, and especially semantics. It gathered together almost all the researchers working the field at the time, giving them an opportunity to air their work in front of others, and partake in some very important discussions. The impact of this conference, although not immediately apparent to all at the time, was immense—and no-one felt that more strongly than the new Vienna Science Group[1] under Heinz Zemanek. This concentration of expert knowledge was exactly what they needed to begin working seriously on the topic of programming language definition themselves.

[1]Note: the group had a number of names across its lifespan, but called itself internally 'VAB', which is how it is indexed in the present work. In-text references to the group, however, will use whichever term was being used at the time.

This chapter is divided into a number of sections. Section 4.1 explores the background and context to the conference and explains the motivations. Section 4.2 looks at the details of organising the event. Section 4.3 walks through the sessions held at the conference. Section 4.4 considers the role of some people who were affected by FLDL but did not present a paper there. Finally, Section 4.5 concludes. A timeline of the major events detailed in this chapter can be found in Figure 4.1.

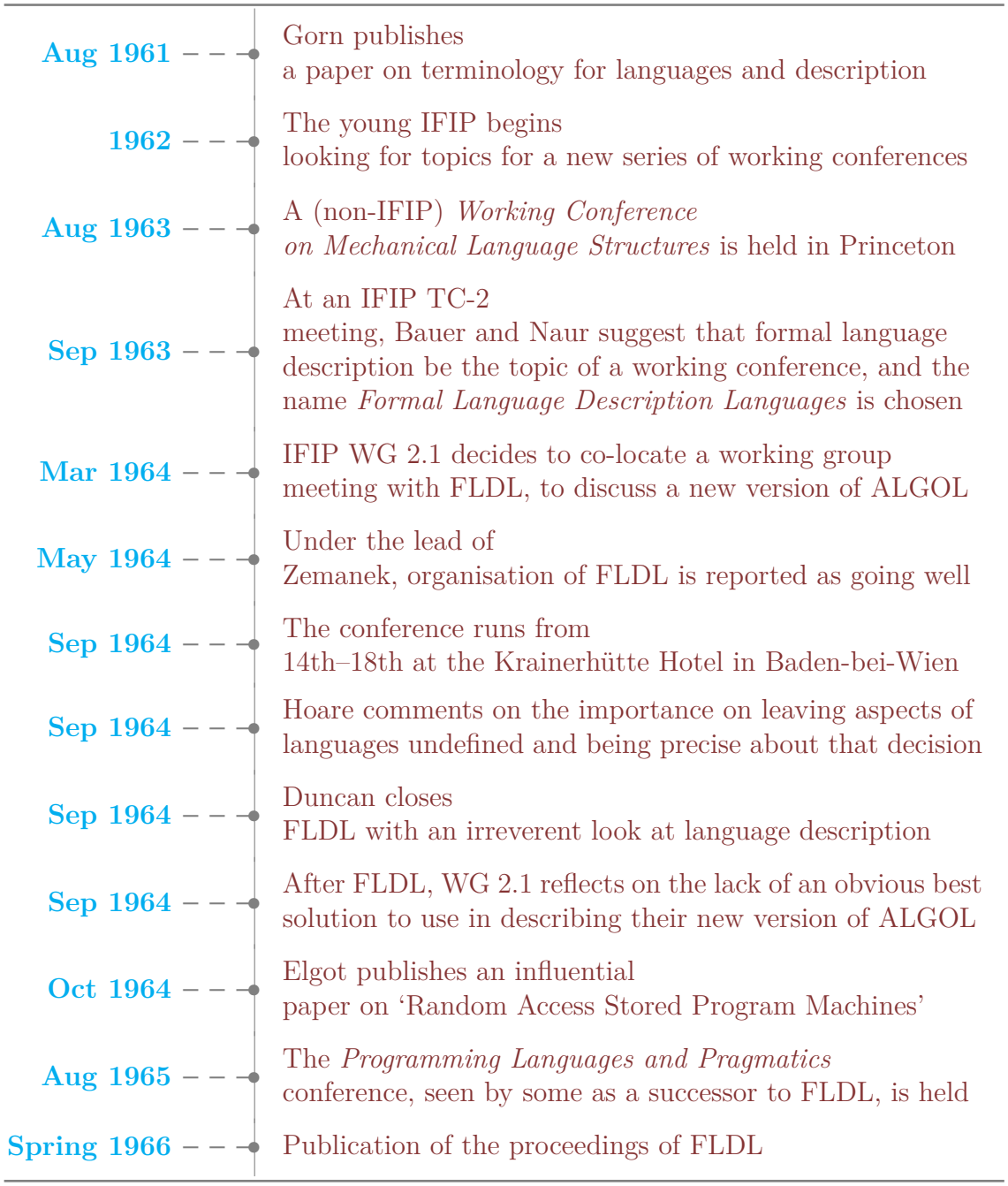
4.1 Motivations

In 1963, the International Federation for Information Processing, IFIP, was a young organisation, just beginning to find its feet on the world stage. With an International Congress and a Symposium on Symbolic Languages both in 1962, IFIP was exploring the possibilities these international gatherings could bring. Zemanek, chair of IFIP's Technical Committee 2 (TC-2) on programming languages, wrote that the organisation was keen to set up useful and productive working conferences: smaller and more focused events than the giant and multi-topic congresses.

> The model envisaged that a TC should work out a list of some 50 to 80 specialists working in a field that was still new and yet developed enough for many people to work in it and to make it possible for discussions and working conferences to bring progress and consensus.
> (Zemanek 1981, p. 13)

TC-2's scope included "general questions on formal languages, such as concepts, description, and classification" (Steel 1966c), and at the committee's third meeting, Zemanek proposed a working conference "on some subject in programming languages". Naur suggested "formal languages", but it was Bauer (in Utman 1963, p. 7) who provided the core motivation for what was to become a very important conference in this burgeoning field: "the problem today is to find or develop suitable formalization to reduce recurrent languages and concepts to the usable level." Naur (in Utman 1963, p. 8) then clarified the topic precisely: "formal and programming language description." The minutes clearly show a group decision on this part; but to listen to Zemanek tell the story, one would think it was solely his influence: in an interview by Aspray, he explains that he was already thinking about the problems of trying to provide a description for programming languages, and hosting a conference on the topic would provide a route towards this goal (Zemanek and Aspray 1987, p. 44).

Figure 4.1: Some key events in the history of the FLDL conference.



TC-2 had been considering creating a working group on 'meta-languages' in 1962, but had decided not to proceed at that time due to the new state of the field and lack of obvious international appeal[2] (Utman 1962b, p. 3). As the early years of the 1960s wore on, however, interest was growing: the 1963 *Working Conference on Mechanical Language Structures* showed a semantics Zeitgeist developing. Despite no paper submitted to this conference concerning semantics directly, the topic cropped up multiple times in the discussions and summaries (Gorn 1964). Bauer (in Utman 1963, p. 8), at the TC-2 meeting where FLDL was proposed, acknowledged this growing interest and development in the field of formal language description, and suggested that a symposium in this area would be a good way to test the water for a possible working group. As a result, the name 'Formal Language Description Languages'[3] was chosen for the conference at the meeting in September 1963.

Also decided at the same meeting was the conference location and time of year: Vienna, in September 1964. The time was chosen to avoid conflict with other IFIP events, and to allow people who had been present at a related conference on 'Algebraic linguistics and automata theory' held at the end of August in Jerusalem to attend more easily (Utman 1963, p. 8). Caracciolo had suggested hosting in Pisa, but Zemanek's proposal of Vienna won out. The minutes state "It was decided to hold this conference within the city of Vienna as a convenience to the organisers and attenders (considering the month, the weather, entertainment and convenience from Jerusalem)" (Utman 1963, p. 8). However it is hard to see why Vienna trumps Pisa in these respects; one may suspect Zemanek's influence as chair of TC-2 may have been the deciding factor instead. Caracciolo was, however, granted the hosting of the following Working Conference, on symbol manipulation languages, held in September 1966.

Tom Steel, also a member of TC-2, provided more explanation of the motivation for the conference in his paper delivered there. He explained that Gorn (1961b) had compiled a list of possible techniques for syntax specification languages, but it would have been impossible to do this for semantics at that time. The only guaranteed recourse available to contemporary practitioners striving for a complete description, argued Steel, was to define a program by its machine code—although that came

[2] The working group was eventually formed in 1966; see Section 7.2.

[3] A quick aside on the name of the conference itself: where does the 'Formal' bind? Does it mean formal languages for describing languages? Or description languages for formal languages? This ambiguity was intentional and present right from the proposal, according to Randell (personal communication, 2018).

with plenty of problems of its own, chief among them portability. Dissatisfaction with this state of affairs was a primary motivating factor in the creation of the conference (Steel 1966b, p. 25).

The preface to the proceedings of FLDL explained the situation:

> TC-2 observed that among the principal difficulties facing the designers and implementers of programming languages and their compilers has been an inability to obtain precise descriptions of the formal languages with which they must deal. [...] TC-2's objective in undertaking the sponsorship of this Working Conference has been to bring this and related questions under the concentrated scrutiny of an appropriately constituted international colloquium.
> (Steel 1966c)

As chair of TC-2 and the local organiser, Zemanek arranged for IBM to pick up some of the cost for the conference. He had been with the company for two and a half years at that point, and already had enough influence to convince IBM to be a sponsor. The additional help was very much required, as IFIP had provided only "token" support of $200. Rank Xerox also contributed some copying services to assist in the task of arranging the event (Utman 1964b, p. 11).

Zemanek had his own motivations for wanting a conference on language description to be hosted in Vienna: his IBM "Vienna Science Group", at that time building in number and confidence, had finished off working on the ALGOL 60 compiler they had developed for the Mailüfterl computer at the Technical University of Vienna at the start of the decade, and was searching for a new research direction. They had been doing some work on hardware and some on vocoders, but this was not terribly satisfying for the group: Zemanek's vision was to "kickstart a theoretical lab", as Group member Hermann Maurer put it.[4] Much more will be said about the work of the Vienna group in Section 5, but for now it is sufficient to understand that they were a young group of specially selected researchers keen to get deep into important work, and to move from being a subsidiary of the IBM Böblingen Research Laboratory to becoming a laboratory in their own right. Hosting an important international conference he could staff with all his own workers gave Zemanek the chance to bring them up to the state of the art in this area.

[4] In a personal communication, 2016.

A further motivation for Zemanek was his pride in his country of Austria. Always keen to link his work and the work of his group to a glorious Austrian tradition, Zemanek (in Steel 1966c) said in his opening address:

> This country should be a good ground for our work. Wittgenstein was born in Vienna and between 1920 and 1926 he worked as a teacher a few miles from here. Before World War II many basic ideas for our subject were created at the Wiener Kreis. Names like Schlick, Carnap [...] and Menger are connected with it. May I finally remind you that Kurt Gödel was an assistant at the University of Vienna when he published his famous paper in 1938.

So this conference had many motivating factors. IFIP wanted to prove itself as a new organisation and establish a new kind of symposium. Zemanek wanted to establish his group, provide training for his workers, and demonstrate its place in the international scene. Both of these suggested the importance of *a* conference; the choice of topic was somewhat secondary to these goals—although IFIP did wish to test whether the field of formal language description was sufficiently mature to warrant a working group.

4.2 Organisation

Zemanek was the main organiser and handled (through an army of underlings[5]) most of the arrangements of the conference. Tom Steel was the corresponding organiser for US-based enquiries. Attendance at FLDL was by invitation only, as Steel (1966c) noted in the preface to the proceedings:

> Attendance was limited by invitation to recognized experts in one or more of the various disciplines of linguistics, logic, mathematics, philosophy, and programming whose frontiers converge around the subject of the meeting. The resulting group - 51 individuals from 12 nations - was ideal in size, breadth of experience, and commitment to the enterprise.

Another attendee, Doug McIlroy, agreed the guests were of an august nature, noting also the stark difference he experienced to the kind of computing being done in the US at the time:

[5]Two of his former workers, Cliff Jones and Kurt Walk remembered that Zemanek employed almost as many support staff as he did scientists, and himself had two secretaries. From personal communications, 2016.

> That meeting, for me, for most people at the meeting—it was the real computer scientists in Europe. There were no real computer scientists in the US at the time.
> (McIlroy 2018)

McIlroy went on to speculate that this could have been because there was a relative paucity of machinery available to European researchers: "you didn't have that many computers, so you figured out how to use them well" (McIlroy 2018).[6]

In March 1964, IFIP's Working Group 2.1, in charge of ALGOL, was hard at work thrashing out strategies for developing new versions of that language: a short term incremental improvement, termed ALGOL X, and, longer term, an entirely new language, called ALGOL Y. With so much work to do, and their next planned meeting not until 1965, the group decided at their third meeting to have an interim meeting at the same time as the FLDL conference. This was particularly advantageous given the overlap in membership of the working group with invitees to FLDL, as well as the related topics, although not all members were convinced of the utility of the idea, which barely scraped past a vote: six members were in favour, six against, and six abstained (Bowlden 1973, p. 22).

The plan for this short meeting was to spend the first day, before FLDL, deciding features for 'Future ALGOL' and the one after the conference on 'meta-ALGOL' (i.e. the method of describing the new languages) (Utman 1964a, p. 21). These conversations show a strong, unanimous commitment to providing a full formal description of syntax and semantics of the new languages, although little in the way of agreement about the best strategy (Utman 1964a, p. 14–17). Indeed, there was some hope amongst the group that a description technique would be put forward at FLDL that would be the obvious candidate for describing their new language.

[6] This is an attitude with which J Moore, author of theorem provers NQTHM and ACL2, agrees. He said:

> I came over here [to Europe, from America, in 1970] and suddenly the computer was this precious thing that only eight people could use at a time, and even they couldn't use it too much because the memory was so small. The idea was: don't sit down and hog the terminal until you know exactly what you're going to do and you make the best use of every key stroke. Plan it out on paper, figure out what it's supposed to do, figure out how you're going to find out if that's what it does, and then when you're ready, get on the machine and try it. It was the difference between a resource-rich engineering environment, very experimental, in America—or at least at MIT—and very theoretical, treating the computer as this delicate, holy object.
> (Moore 2017)

The working group also wrote to Zemanek to ask him to extend invitations to all members of WG 2.1 who were not already invited (Utman 1964a, p. 22).

This resulted in almost half the attendees at the *Formal Language Description Languages* meeting being members of WG 2.1; some FLDL attendees who were not members also observed at the 2.1 meetings. These included Caracciolo and Steel (Utman 1964c). Conversely, some WG 2.1 members attended FLDL despite having done no previous work in the area of language description, such as Randell, Hoare and McIlroy; Hoare even went on to start work on semantics not long after this conference. McIlroy recalled there being no distinction made at FLDL between 2.1 members and the original invitees: "At the conference it all coalesced into one" (McIlroy 2018). This was in sharp distinction to his experience at the WG 2.1 meeting, at which it was "made very sure" he was an observer; he recalled "I sat at my own private table in the corner of the room and listened. [...] As an observer I was definitely a second-class citizen. [...] I did not speak unless spoken to!" (McIlroy 2018).

The organisers hoped that a "meeting of minds" would take place between the different groups present at the conference. These could be largely defined as those who were more language-focused, and those who worked on implementation: the theorists and the practitioners. Whether that succeeded is less certain: Brian Randell recalls seeing Saul Gorn, a prominent theorist, walking through the gardens arm-in-arm with another theorist and lamenting "Well, there's them, and there's us" (Randell 2016).

Another way in which the conference participants could be divided is between those who were interested in the semantics of programming languages, and workers in the field of formal language theory. This could be seen as a reflection on the ambiguity of the name: some participants were interested in the formal description of languages, and others in the description of formal languages. The former group are covered more heavily in the present account due to their higher relevance to the main story, but there was certainly a sizeable contingent of the latter. Many very important names from the field were present, such as Marcel-Paul Schützenberger and Seymour Ginsburg. Even Noam Chomsky was invited, although he declined to attend (Zemanek 1964b). A likely factor in the number of more linguistically-minded attendees is the proximity in time to the Jerusalem 'Algebraic Languages' conference (Utman 1963, p. 8).

There was not always a great deal of overlap between these two groups: those from

each group tended to comment on their own group's papers. Where there were interactions, these tended to be somewhat confused. For example, Caracciolo, a language theorist, expressed difficulty in handling the block structure of ALGOL in formal language terms, and Naur (firmly in the language description group), asked whether he couldn't simply use a pushdown list. Caracciolo's response was that he could—but he was trying to avoid using an algorithmic approach (Caracciolo di Forino 1966, p. 46). Later, Strachey would remark somewhat scathingly that the language theory focus on the study of context-free grammars—prevalent at FLDL—was of little interest:

> The theory of context-free grammars is now a rather over-worked field; it has never been one of great depth and its connections with computing are at best tenuous.
> (Milne and Strachey 1974, p. 10)

This proliferation of differing views and backgrounds in attendees was reflected in the cover of the program for the conference, which showed a reproduction of Pieter Bruegel the Elder's *Tower of Babel*. A photograph of the program can be seen in Figure 4.2. This had already seen use as a visual metaphor for the fragmentation of computer systems, on a 1961 cover of *Communications of the ACM* (Priestley 2011, p. 204). Zemanek particularly liked the picture, both for its connotations of disunity and language proliferation, and its Viennese pedigree—the version placed on the cover of the conference program was, of course, the one from the Kunsthistorisches Museum in Vienna. Again, this demonstrates the characteristic touch of Zemanek overseeing conference organisation; indeed, his presence was felt so clearly that the impression amongst his staff at the time was that he had been the sole originator of the idea (Neuhold 2016).

The conference was held over multiple sessions across the week of the 14th to 18th September 1964, with breaks for excursions, and dinners. Although the dates indicate the whole week, the Monday event was only an evening 'informal talk' at the hotel (IFIP 1964b). The venue was the Krainerhütte Hotel in Baden bei Wien, a spa town in the woods south of Vienna, a short and scenic tram ride from the city. This location was chosen, over the initial decision of the city of Vienna, because September was still in the tourist season, and a hotel in the city would have been prohibitively expense: over twice the cost of similar accommodation in Baden (Zemanek 1964b). Incidentally, Baden was also the birthplace of Kurt Walk, one of the

Figure 4.2: The front cover of the FLDL program (IFIP 1964b).

members of the Vienna group.

All speakers were required to write their papers in full for the conference in advance, before the end of July, so that they could be circulated to attendees in enough time for reading (Zemanek 1981, p. 13). Each speaker would give a presentation at the conference about their work to illustrate the main points, and then a discussion would ensue (Steel 1966c). TC-2 took some time and care over this decision, as the minutes of their planning meeting show (Utman 1963, p. 9). The work could not already have been published, as that would prevent re-publication by IFIP; requiring unpublished would also serve to encourage influence and co-operation between the practitioners of this young field, who might wish to modify their papers following discussions. On the other hand, *something* had to be distributed in advance, as many of the ideas were likely to be too complex to be explained easily just by talking: the extra time and opportunity for careful reading of formulae afforded by advance paper copies would aid proper understanding. Speakers were required to have previous publications, however, to indicate their knowledge and expertise. Quite how many attendees studied every paper in depth beforehand is unknown, of course, but the depth of some questions asked in discussion sessions does seem to indicate familiarity with the work in certain cases.

The proceedings for the conference (Steel 1966a) were edited by Steel, and, at van Wijngaarden's suggestion, published by North-Holland, supported by IFIP (Utman 1963, p. 9). Early estimates after the conference, at the May 1965 meeting of TC-2, hoped for a July 1965 publication date, with a cost of approximately \$6–7 (Utman 1965).[7] However, by November 1965, the cost was confirmed at the higher price of \$11.20, with publication to come "soon" (Teufelhart 1965, p. 7). The main reason given for the delays was the time taken by the authors to review and edit their papers prior to publication, which came eventually in late 1966 (Teufelhart 1966, p. 8). Fraser Duncan (1966), who gave the after-dinner speech on the Friday evening at the end of the conference, even claimed that by the close of proceedings, some 16,000 sheets of paper had been used.

Another potential reason for the delays was the involved process of transcribing and editing the discussions which had taken place after each talk, providing a verbatim account of the reception each paper received. Landin (2001) acknowledged the special and useful nature of this as a resource in later reminiscences. Zemanek (1981,

[7] Minutes of this meeting do not have numbered pages, so page references are not provided in citations.

p. 14) explained this process:

> For this purpose there were a number of portable tape recorders in addition to the master tape on which the speakers were recorded; whoever wanted to contribute to the discussion had to wait for one of the conference assistants to come up with the recorder. That assistant pronounced the name of the speaker so that all names were recorded without exception.

These "conference assistants" were all members of Zemanek's IBM science group, a deliberate decision on his part, as it gave his team the opportunity to get to know everybody at the conference, which was essentially everyone working in the field at the time.

> My people were running around with the portable tape recorder in the discussions saying the name of the discussant before he could start his sentence into the microphone. So there was never a question of whether somebody had mentioned his name or not, which forced my people to know everyone.
> (Zemanek and Aspray 1987, p. 45)

This process can be seen in Figure 4.3, as well as some members of the Vienna group. They were referred to as "scientific secretaries", and their experience served to immerse them in the ideas of formal semantics, a subject in which they had no previous knowledge. Erich Neuhold (2016) recalls being somewhat bowled over by the experience: he was surrounded by very senior scientists, something he was not used to as he was very new to IBM—he had only finished his *Diplom* one year previously. Compounding matters for Neuhold, his English was not quite good enough to follow all the complex discussions; but he remembers all the attendees were very patient and respectful. As well as recording sessions, the scientific secretaries were in charge of looking after the guests in their sessions, and managing the assistants.

Steel (1966c) noted that he was in sole control of editing the transcripts of the discussions, and he kept them as faithful as possible to the recordings: "The participants have not had an opportunity to amend their remarks. Thus, what is reported is what was actually said, not what the speaker might have wished he had said."

Actual transcription was performed by Judi Shirack; typing for the proceedings was the work of Mary Kay Gerth, Eleanor Faulkner, Betty McQuay, and Evangeline Villaros. Steel also credits Virginia Parry Herold as an editor (Steel 1966c). It is

Figure 4.3: A discussion session at FLDL, with Paul Oliva of IBM holding the microphone on the left, and Tom Steel speaking (right). Seated facing in the foreground is Corrado Böhm and in the background, just to the right of Oliva, is Peter Lucas of IBM. Kurt Walk of IBM is visible on the far right. Photograph is from Auerbach (1986a, p. 80).

interesting to consider what the breakdown of work between them entailed—one suspects Steel did little of the manual work.[8]

4.3 Sessions

The program for *Formal Language Description Languages* shows that the talks were grouped into somewhat themed sessions: the two main topics of language description and formal language theory were not really mixed (IFIP 1964b). The order of talks is slightly different to how they appeared in the published proceedings; this section treats each talk in the order in which it took place at the conference.

4.3.1 Session 1

The first section took place on the morning of Tuesday 15th September 1964 and featured talks by John McCarthy, Adriaan van Wijngaarden, and Tom Steel.

[8] This is typical of the situation in the 1960s, where a lot of the more secretarial and day-to-day work was done by women, whose names are then often lost to history. For more on this, see Hicks (2017).

McCarthy's talk, 'A formal description of a subset of ALGOL' (McCarthy 1966), was listed in the programme as 'Syntax and the Formal Description of the Semantics of Programming Languages' (IFIP 1964b); the change in title between the publication of the programme and the published proceedings is interesting and could imply a less obvious focus on Micro-ALGOL. Another note is that as of May 1964, according to a report on the preparations for the conference, McCarthy was not one of the invitees. This would imply he was a later addition; it is possible that the more European organisers were not aware of McCarthy's work on formal language description.

Hermann Maurer, who was the secretary for the first session, was given particular charge of taking care of McCarthy.[9] He recalled an amusing story about the presentation McCarthy gave:

> When John gave a talk he was a leftist: he kept turning around anti clockwise, so the mike-cable (no wireless at that point) got snarled in loops around his body and I had to interrupt his lecture twice to entangle the mess. The discussion with him and his Go level[10] convinced me: here is an unusual man, and as he rose to be a "star" I had the satisfaction of having guessed right.

The reaction to McCarthy's paper is discussed at length in Section 3.1, but some more comments here reflect the interactions with other participants at the conference. Many people were concerned about the small size of the language McCarthy chose to model, and worried whether the method would extend to tougher constructs. McCarthy (1966, p. 8) willingly acknowledged this criticism, without explicitly negating the fears:

> MCCARTHY
> That's right; Micro-ALGOL is very simple.
>
> STRACHEY
> All right. *Minute* Micro-ALGOL! [Laughter]
>
> MCCARTHY
> If anyone should presume to make a Micro-ALGOL compiler, I will be grossly insulted. [Laughter]

Outside of his own presentation, McCarthy was keen to voice his ideas for the meaning of semantics. Speaking after Steel's presentation, McCarthy (in Steel 1966b,

[9] From personal communications, 2016.

[10] This is mentioned in Section 3.1.

p. 36) said "To describe the semantics of a language you say what happens to the state when you go from one end of the program to the other". Note that this focus on the resultant state is slightly different to some other situations in which McCarthy emphasised the *action* as the meaning instead. In the same discussion session, he also described van Wijngaarden's approach: "You have a certain expression and you give the value of this expression", a summation with which van Wijngaarden agreed.

Perhaps the best expression of McCarthy's thoughts was given in response to Strachey's talk:

> I think that the essence of semantics is to describe the relationship between a symbolic expression and what it represents. Now if it only represents another symbolic expression, then that is one thing; but what we are really describing when we describe computation are processes, and I think that if you listen to the informal discussion that goes on at the ALGOL Meeting then you will find that people are really discussing (when they describe what a program really means) the question as to what transformation the values of the variables really undergo, and you simply can't evade this. Eventually your desire is to make a formal notion which will correspond to people's intuitive notion as to what a program does. (McCarthy, in Strachey 1966a, p. 219)

Van Wijngaarden's paper 'Recursive definition of syntax and semantics', as discussed in Section 3.3, was a follow-up to his 1962 paper 'Generalized ALGOL'. More emphasis was given in the later paper on the operation of the preprocessor and its manipulation of program texts. The discussion comments showed a lot of confusion over what precisely van Wijngaarden was trying to do, which was to avoid a syntax/semantics distinction and treat the whole process as symbol manipulation.

After Landin's talk, van Wijngaarden gave a long speech—not directly related to that particular paper but rather to the conference as a whole—on his view on the definition of computation. He explained that there had been a lot of discussion over which approaches were equivalent or even better from a mathematician's point of view. However, van Wijngaarden continued, this presupposed a general consensus amongst mathematicians, when in fact there was a great deal of difference in opinion. Van Wijngaarden himself took a very constructivist view: a number could only be said to exist if it can be constructed, and that construction is a denotation of the number. This would not necessarily lead to different computation results when compared to those obtained by a mathematician with a different viewpoint, but it did mean that one might have a result where another did not. Consequently, van

Wijngaarden's approach was that if a constructable algorithm could not be crafted for a particular computation, it did not exist. In short, he said (in Landin 1966c, p. 294), "I can't do any kind of computation apart from acting on symbols". This rather nicely summed up his attitude to the description of programming languages, and is worth bearing in mind when considering his strong statement "I have never seen a 'number' " (van Wijngaarden 1966b, p. 23).

The third speaker in the first session was Thomas B. Steel, Jr., member of TC-2, secondary organiser of the conference, and editor of the proceedings. Steel had long been working for the System Development Corporation (SDC), at the time of the conference a non-profit organisation providing computing expertise for the US military. Steel was the face of SDC in a famous advertisement from the mid-1950s trying to recruit more programmers, seen in Figure 4.4. He was a member of many organisation such as IBM's usergroup SHARE and various others on standards; he won the ACM Distinguished Service award in 1977 for this work on standardisation (ACM News 1977, quoted in Lee 1995). Prior to the FLDL conference, Steel had been working on programming languages (Dobrusky and Steel 1961) and formal language theory (Steel 1964).

Steel's FLDL paper, 'A formalization of semantics for programming language description', was less a technique for describing languages and more a clarion call for the concept, with a few starting ideas sketched out (Steel 1966b). He decried the idea of trying to use natural language for semantics, terming it, with a phrase borrowed from poet Alexander Pope, "prose run mad" (Steel 1966b, p. 26).

The approach described in Steel's paper begins by assuming that all programming languages are equivalent to Church's family of recursive, computable, functions; with this assumption, one need only define recursive languages and from this will flow the definition of programming languages. This is something of a classic mathematician's approach: reducing the problem to a known problem. However, the undecidability of truth, argued Steel, prevents the full formalisation of semantics. Instead, one can axiomatise most parts of languages: as long as agreement on meaning can be reached, the main goal of language description has been achieved.

Steel then took a formal language approach to the problem: a meta-language $\hat{L}$ was introduced and described in terms of itself. Steel illustrated that it was sufficient to describe any recursive language, and that it had combinators which allowed formalisation of lower predicate calculus, identity, and membership, along with an if–

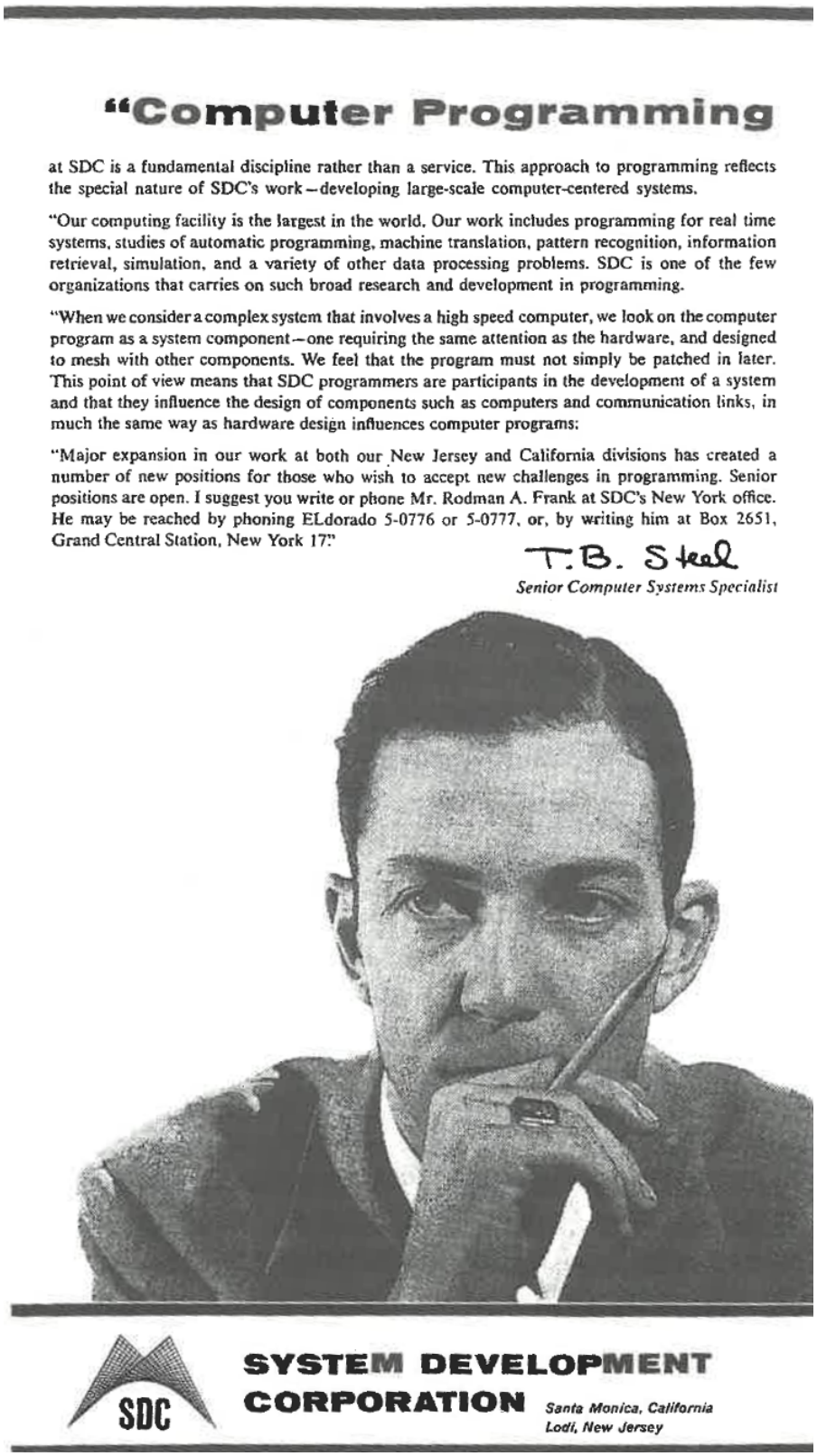


Figure 4.4: Tom Steel poses in an advertisement for his company SDC in 1956 (taken from Campbell-Kelly (2003, p. 28)).

then–else notion. The description of $\hat{L}$ was somewhat complicated and formal, quite possibly too much so for some of the attendees; indeed, van der Poel (in Steel 1966b, p. 35) explicitly stated that the paper was too technical for him to understand.

Assuming it would be possible to define the meaning of $\hat{L}$ sufficiently that everyone could understand it, all recursive functions could be described, and therefore all programming languages. Steel explained:

> Essentially, I am looking on a program as a complex transfer function from initial state to final state and trying to use this kind of mechanism to explain exactly what this transfer function is.
> (Steel 1966b, p. 36)

Despite the complexity of the language-theoretic presentation, Steel here reveals his central thoughts about the meaning of programs to be rather similar to those of McCarthy and Strachey. This demonstrates that although the methods of presentation could vary wildly, there was a reasonable amount of agreement amongst those present at this conference about the core notions of computation.

In the discussion of his paper, Steel explained that a danger to language description techniques was a confusion between language and metalanguage, especially where names and operators are used in both contexts (Steel 1966b, p. 34). This is a crucial point, and one that caused problems in a lot of semantic descriptions. However, van Wijngaarden's responses here indicated that he did not see a problem with this; as we have already seen, van Wijngaarden's approach treated all aspects of the language description in the same way: strings of characters.

Stephen Warshall, as a programmer, was not so interested in the mathematical aspects of Steel's talk.[11] He joked "I'm not very interested in existence theorems, although I'm sure they are there" (Warshall, commenting on Steel 1966b, p. 34). Warshall explained he thought of semantics in terms of basic operations of a real machine, and was not interested in meaning being built from lots of tiny primitives. This is interestingly similar to Gorn's (1964) concept of a 'background machine', and reveals something of the working programmer's notions of language meaning. Steel's response to Warshall's (somewhat lengthy) comment was delivered in a tone of irritation: "If you will quit asking the question, I think I can answer it" (Steel

[11] Incidentally, Warshall was intending—as of May 1964—to speak at the conference, with a talk titled 'On the semantic description of programming languages' listed in the draft programme, but by the publication of the full programme later in the year, he had withdrawn his paper (IFIP 1964a,b).

1966b, p. 35). He stated that he believed very firmly that the only way to construct meaning was from a collection of small primitives, and his approach intended to explicate those primitives.

4.3.2 Session 2

The second session, held on Tuesday afternoon, had a formal language theory theme, and featured presentations by Alfonso Caracciolo di Forino, Jürgen Eickel and Manfred Paul, and Karel Čulík.

Caracciolo di Forino (1966), speaking 'On the concept of formal linguistic systems', presented a series of systems which comprised a formal language and its interpretation rules. These rules mapped languages into subsets of their alphabets, defining grammars in a symbol manipulation strategy. This was an attempt to place the task of formalising the syntax of programming languages, especially ALGOL 60, within the bounds of formal language theory.

Commentary on this paper included a debate about whether a set of words is sufficient to establish a language, and to what extent the abstraction problems of syntax and grammar can be addressed purely symbolically, with Caracciolo acknowledging that the semantics, "the most important part of language definition", would be very difficult to cover in this way (Caracciolo di Forino 1966, p. 48).

Another interesting aspect to the discussion of this paper revolved around the division between the formal language theorists and the programmers, as mentioned earlier in Section 4.2. Naur expressed his surprise at Caracciolo's difficulty in handling ALGOL phrase structure, given how relatively straightforward it was to address this in a compiler. The following exchange demonstrates this nicely:

> CARACCIOLO
> By a mechanical point of view, you speak of an algorithm, and I already said there is nothing difficult at all to define by an algorithm.
>
> NAUR
> What is then your difficulty?
>
> CARACCIOLO
> Just to find basic concepts, for defining certain classes of strings taking into account all syntactical rules, such as for instance that an identifier must not be declared twice in the same block.
>
> NAUR
> Yes, how simple.

CARACCIOLO
How simple to realize if you write a procedure. The problem is, however, to find a metalanguage for doing that in a declarative way, not in an operational way.

NAUR
Then your problem is purely self-inflicted. You do not want to use an algorithm.

CARACCIOLO
What I want to do is to define grammars, where possible, in such a way that a recognizing algorithm can be automatically derived.

An exchange like this illuminates the reasons that Gorn may have had to remark to his theorist friend "There's them, and there's us", as Randell (2016) had observed. It is also interesting to compare Caracciolo's motivations here with those of Landin with his AE/SECD approach: while the two scientists had very different approaches to the problem, the desire to find a declarative solution is emblematic of the struggles of many semantics writers to derive a more mathematical approach to programming languages—to find a way to fit computation within a more comfortable and familiar framework.

The second talk of the afternoon was given by Eickel and Paul (1966) and was entitled 'The Parsing and Ambiguity problem for Chomsky Languages'. Their paper attempted to present and answer to this problem, using mathematical language theory and string manipulation. A very involved paper full of complex equations and diagrams, it is unique at the conference in having no recorded discussions. The proceedings notes "The comments following this presentation were very brief and did not touch on the technical content of the paper. They are therefore omitted here" (after Eickel and Paul 1966, p. 75). One may speculate that this was due to the deep technicality of the paper and the lateness of the hour—the talk likely ended around five o'clock in the evening of the first day of the conference. Perhaps the sentence in the proceedings is simply a polite way of describing stunned silence.

Finally, Čulík's talk 'Well-translatable grammars and ALGOL-like languages' (Čulík 1966), concerned the application of ideas from language theory in performing string manipulation translation from one language to another. Čulík's other work at the time shows an interest in this area of translation using formal language theory, but as the language into which he translated did not have the characteristics of a metalanguage used for semantics by most practitioners in the field, it does fall into the remit of semantics as covered in the present work.

There was at least some discussion of this paper, unlike the previous talk, but it was brief and referred to some relatively trivial aspects of the presentation.

4.3.3 Session 3

The third session took place on Wednesday morning and was another which gathered papers on formal language theory. Presenters were Seymour Ginsburg, Marcel-Paul Schützenberger, and Kurt Walk.

Ginsburg delivered a paper on 'A survey of ALGOL-like and context-free language theory' (Ginsburg 1966). This talk was an overview of the area, looking at operations, translation, ambiguity problems, and recognisers, and so is not relevant to the main story. The discussion here revolved mostly around the precise meanings of certain terms, particularly 'non-deterministic'.

Schützenberger's paper 'Classification of Chomsky Languages' was a brief work discussing methods for classifying formal languages (Schützenberger 1966). An interesting note here is that it was translated from the original French by Zemanek himself.

Kurt Walk was the only member of Zemanek's Vienna group to speak at the conference. Maurer provided an explanation for this: the occasion was too important for Zemanek to allow just anyone from the group to speak: it had to be one of the managers with good PR skills. Walk was particularly good at recognising and nurturing the talents of his employees as well as contributing himself, so he was chosen.[12] His paper was on some properties of formal grammars and was entitled 'Entropy and Testability of Context-Free Languages' (Walk 1966). This was not at all representative of the work he and the majority of others in the group performed after the conference, showing how influential that meeting was on their direction. The paper was quite long and rather thorough, with plenty of diagrams and examples as well as proofs given for a number of the concepts under discussion. Clearly Walk was an accomplished mathematician, and this paper illustrates how keen Zemanek was to show that his group had this kind of ability and rigour.

However, there is another way to interpret this paper. At the end, thanks are given to Zemanek and Werner Kuich, another member of the Vienna Science Group. While Walk had not written and did not write any other work on formal languages, Kuich was much more interested in that topic, later writing papers such as 'On the entropy of context-free languages' (Kuich 1969a,b) and 'Systems of pushdown

[12] Personal communication with Hermann Maurer, 2016.

acceptors and context-free grammars' (Kuich 1970). This may imply that work primarily performed by Kuich was delivered by Walk instead—with Zemanek choosing work that was relevant to the topic of the conference, and a presenter that had the requisite skills and experience for such an important occasion.

4.3.4 Session 4

On the morning of the third day of the conference, the fourth session returned to the topic of language description. Papers were delivered by Nathaniel Rochester, Jan Garwick, and Maurice Nivat and Louis Nolin.

Rochester, an IBM researcher who was the 'chief architect' of the IBM 701 machine that kickstarted IBM's dominance in the hardware field (Rochester 1983), presented a rather practical paper on a strategy for presenting programming language syntax, 'A formalization of two-dimensional syntax description' (Rochester 1966). Rochester explained that at IBM Poughkeepsie, New York state, where there was a development laboratory as well manufacturing plants and a distribution centre (IBM 2001), the syntax description needs of COBOL users clashed with those who used FORTRAN and ALGOL:

> One group [the COBOL users] absolutely demanded alternatives vertically, and the other group [FORTRAN and ALGOL users] rejected that. The FORTRAN/ALGOL people said that they must have syntactic brackets and the COBOL people said they absolutely must not be used. (Rochester 1966, p. 136)

This serves to demonstrate the immense difficulties faced by anyone who attempted program descriptions: the audience was very likely to be split and have differing requirements, even within the very same facility of the same company. Further, it illustrates that IBM was not a monolithic entity with one central set of beliefs.

Rochester's proposed solution to this problem was to find a way to formalise the two dimensional syntax presentation that the COBOL users desired—employing some syntactic brackets. His notation is claimed as being complete and unambiguous, and a correspondence is established with the more standard BNF. Rochester argued this could help with language standardisation, especially as the standard COBOL approach was informal and ambiguous.

The paper presented by Garwick, entitled 'The definition of programming languages by their compilers' (Garwick 1966), was very short, and outlined a sketch of a way

to use some of the desirable qualities of compilers to aid in the description of programming languages. Garwick explained that a compiler is very good at determining whether a program is syntactically valid, and at giving a meaning for it: the object program. Garwick's suggestion was to propose a machine-independent compiler alongside each new language. This compiler should be written in a generally-readable language and accompanied by a clear natural language description. Rather than being written for any particular computer, it should be described in terms of the operations of a 'machine independent computer', which would allow the description to be very general. Whether that computer was abstract or real, argued Garwick, was irrelevant, as long as its machine code was easily translatable to that of other computers. While this compiler would not be useful in the sense of producing efficient code, it could be used a basis for the development of other, more practical compilers.

The discussion session was rather controversial: for a paper lasting barely two pages in the proceedings, there are six and half for the discussion. Many of the discussants asked about exactly how one would go about constructing a compiler for a machine-independent computer without knowing particular machine properties such as word length. Garwick mostly waved off the criticisms by saying things along the lines of "We try our best" and "we do some tricks and assume things for certain algorithms". This, coupled with the short nature of the paper, and the lack of previous publications from Garwick on the subject, imply a certain speculative nature to his talk. Nevertheless, there was not total dismissal from the audience: Bauer thought the idea had merit and was at least worth further investigation.

Landin was one of those who was sceptical, and questioned exactly how the machine-independent computer could be defined by its language. He argued that machine code was not as easy to define as Garwick implied, and anticipated plenty of disagreement when trying to determine what should be included. Strachey agreed with this perspective and added that the process of writing a compiler is very difficult, something he knew well given his ongoing struggles with the CPL language project, at that stage two years deep.[13] He explained (in Garwick 1966, pp. 145–6): "The only way to define what the translation should be will be in some sort of meta-language or semantic description of what you want the language to do - and then you're back at the problem of describing what you want it to do." This fits in with Strachey's view that the really important task is first determining content, before

[13] More of the story of CPL is described in Section 6.2.

trying to work out how exactly it should be specified.

Hoare gave a very nice and considered response to this paper, which is worth quoting in its entirety:

> I have a comment on your paper which I also think applies to other papers. Every implementation of a programming language is, I think, a precise definition of the semantics - not necessarily an accurate one incidentally. [Laughter] Any absolutely precise definition of a language - semantic definition - is (or can be) regarded as an implementation. If we want to define a language which is capable of more than one implementation (which for many reasons we do) we must avoid, or be very suspicious of, any attempt to give an absolutely precise semantic definition of the language.
>
> What is required is a method of describing a class of implementation perhaps by giving a criterion for testing whether the implementation is satisfactory. [...] There seem to be two intensely practical problems which have not excited very much interest so far at this conference. First, we must give a great deal of thought to deciding which things we want to leave imprecisely defined. Second, in any formal or informal description of a language, we must have a mechanism for failing to define things. (Hoare, in Garwick 1966, pp. 142–3)

This desire to leave certain aspects of a language undefined, which can be seen as the germ of ideas that eventually grew into axiomatic semantics, was not unique to Hoare: McCarthy and Landin also saw the value in not fully specifying some of the basic components of their language descriptions. Perlis (1981, p. 89), too, had praised the ALGOL authors for having "wisely under-described the semantics". However, Hoare's comment went deeper, and most importantly, introduced the idea of a mechanism to delineate clearly these areas of implementation specificity.

Garwick was quick to agree with Hoare's suggestion, and Ingerman (in Garwick 1966, p. 143) was also in agreement, adding "I find it astonishing to have a speaker at this conference with whom I can agree". This emphasises again the split between more practically-minded authors like Ingerman and the theoreticians present at FLDL.

Hoare was also concerned initially that Garwick was suggesting that a language be specified by a particular machine's compiler, which he was not—but this was not a totally unknown viewpoint. At the 1963 *Working Conference on Mechanical Language Structures* Irons had advocated that very notion, and it was McCarthy (in Gorn 1964, p. 134) who provided the rebuttal: "how can you ever say there is a bug in this compiler?"

The next paper was written by Nivat and Nolin and was another that had a name change from the draft programme: the earlier version was called 'A canonical way for the definition of ALGOL's semantics' (IFIP 1964a) but changed to the much less definitive 'Contribution to the definition of ALGOL semantics' (Nivat and Nolin 1966). Their idea was not to present a description technique, however, but rather a system which reduced ALGOL into a simpler language with more basic constructs: for example, removing **for** statements. A quotation from their paper explains the central notion:

> What has been done in this paper, however, seems sufficient to give an idea of a method for reducing a program with complex structure to a simple one - to one which is, in fact, quite similar to a program written in machine language. *And this is really semantics.*[14]
> (Nivat and Nolin 1966, p. 157)

That is, meaning becomes clearer as correspondence grows with machine features—such as addressing. Nivat and Nolin's approach to reducing programs was performed symbolically, and bore some resemblance to van Wijngaarden's paper: indeed, van Wijngaarden's comments on the paper indicate his general approval. However, the scope of the alterations made by Nivat and Nolin was not as ambitious as van Wijngaarden's, and this may be why there was less disagreement from the audience. Naur was one of those who appreciated the idea: to him, it had a parallel with the way in which one tended to think informally about the meaning of a language.

4.3.5 Session 5

The penultimate conference session was held on the morning of the final Friday, and hosted four talks—although it was not longer than the previous day's session. Here, Calvin Elgot, Corrado Böhm, Peter Landin, and Christopher Strachey all gave very technical talks about language description methods.

Elgot gave a talk entitled 'Machine species and their computational languages' (Elgot 1966b), but before getting into that, it is worth reviewing his background. At the time of the conference, Elgot was fairly new to working in mathematics and research. He had completed his PhD in 1960 and joined IBM's Mathematical Sciences Department at the Watson Research Centre at IBM's Yorktown Heights office in New York. Elgot's interest in computation was framed in mathematics, and was

[14] Original emphasis.

about "not 'doing' computer science but 'redoing' it" (Bloom 1982). Elgot was very keen on elegance in mathematics and admired the European computing community for this reason, mirroring McIlroy's comment about Europe having the "real computer scientists".

The talk presented by Elgot was an elaboration of his previous work on 'Random-Access Stored Program Machines, an Approach to Programming Languages' (Elgot and Robinson 1964), a brief description of which is useful for understanding Elgot's FLDL paper. The 'RASP' machines were described by Elgot as being similar in behaviour to a computer's central processing unit. The name came in distinction from other models of computation, such as automata, that lacked either the random access or stored program properties, both of which Elgot felt were essential to understanding the workings of computers. Their instructions were defined as mappings from states of the machines into modified machine states, and the machines themselves were described as quite similar to physical machines.

This was an interesting early paper. The notion of a function mapping states into states is a core concept of most model-based semantic approaches, especially classic operational and denotational ones; the Vienna group in particular cited Elgot's RASP paper as influential on their work. Elgot's presentation of his ideas was couched heavily in mathematics and automata theory, with little to no example of concrete programming. Instead, properties of the system were described, such as a proof that if a RASP machine could calculate two simple functions and can test for equality, it could compute all partial recursive functions.

Following on from this work, Elgot's FLDL paper explored the various different kinds, or species, of machine that could be described in this way. These included Turing machines, and others with more complex instruction sets. Elgot also claimed that the method could also be applied to mathematical structures such as semigroups. This shows the mathematical and logical background to Elgot's work, something he clearly stated: "Our point of view is inspired by that of the Theory of Models of Mathematical Logic" (Elgot 1966b, p. 160). Elgot's firm background in this field was also demonstrated in a comment on McCarthy's talk in which he corrected a minor point to do with the semantics of predicate calculus.

In response to Elgot's paper, Gorn asked whether a "language for a machine" was a way to derive a language from a particular machine. This, Gorn (in Elgot 1966b, p. 176) said, "is the sort of thing you want to get out of a formalism". Elgot disap-

pointed him by stating that this had not been possible so far; there was more detailed technical work required for this technique than he had anticipated. This was also something of a running theme in programming language description: it frequently ended up being larger and more complex than the authors had initially expected.

A paper which provided just such a large and complex formalism was 'The CUCH as a formal and description language' from Böhm (Böhm 1966). This very specific title was an improvement from the earlier draft programme, for which Böhm had proposed 'Some experiences in formal and description languages' (IFIP 1964a). This is probably a reflection of the improved state of work achieved by Böhm at that later point. A photo of Böhm can be seen in Figure 4.3, on Page 141.

The 'CUCH' of the title was a combination of Curry's combinatory logic and Church's lambda calculus. Böhm (1966, p. 179) stated some properties required for what he desired to produce, "a language appreciated both by mathematicians (logicians and nonlogicians) and by programmers": it must contain a good portion of its own metalanguage, must have a universal algorithm to process its expression, must be able to describe all kinds of computers, has to be able to describe information processing, and must express its functions, operators, relations, and predicates non-algorithmically. This list of requirements has a good deal of overlap with the requisites of formal semantics, with some more far-reaching aims, akin to McCarthy's desire for a unification of mathematics and computation. There is also a good degree of similarity with the goals of both Landin and Strachey for their description techniques, particularly the hope of bringing mathematical rigour.

The CUCH was fundamentally a formal language, with reduction rules and the ability to self-describe. Böhm gave an example description of a flowchart in the CUCH, as well as the BNF for its notation. He noted that the ability to reduce anything written in this language into a normal form matched McCarthy's APPLY function and Gilmore's abstract LISP machine. Böhm (1966, p. 189) argued that his approach "may be considered as attempts at partially inserting the CUCH into new programming languages. Our point of view is that the CUCH itself is a programming language, or better that it becomes an abstract machine the moment we use the general reduction algorithm." This nicely encapsulates the link between the formal language theory content of the conference and the language description work.

One interesting point made by Böhm highlights a problem with higher-order func-

tionality. Suppose the function[15] $\lambda x(x^2)$ is applied to a polynomial expression. It would seem appropriate to consider this as squaring the polynomial; but it could also be interpreted as treating the polynomial as a function and applying it to itself. Böhm argued this was not a problem with the CUCH but with the notion of higher-order functionality itself. Gorn picked up on this idea of multiple possible interpretations of expressions, noted that the same expression could refer to either a number, a program, or a machine, and asked whether conversion between these was possible within one computation. This would produce a great deal of flexibility in the formalism, but could also be a source of confusion. Böhm said that such conversion was possible, but had to be done very carefully.

Other than this one point from Gorn, discussion of Böhm's paper was rather brief. This is interesting given the rather positive reception of WG 2.1 members to the presentation, as is discussed later.

Peter Landin's paper 'A formal description of ALGOL 60' has already been discussed in detail in Section 3.2, but in brief, it presented a programming language semantic description technique involving translation into imperative applicative expressions, a form of modified lambda calculus (Landin 1966c). Meaning was then given to these expressions using an abstract machine called the SECD machine.

The discussion of Landin's paper started with a presentation of a chart he had made, which attempted to fit the various semantic description techniques into categories based on approach. This is shown in Figure 4.5.

Interestingly, the groupings made by Landin do not fit into the later standard styles of splitting semantic techniques, such as operational versus denotational, and are more concerned with dividing those who like imperative constructs and a simplified right-hand side, such as van Wijngaarden, from those who prefer a very descriptive approach, such as Landin himself. Some researchers aimed for a very low-level strategy, i.e. starting with a very well-defined set of basic primitives whereas others like McCarthy focused on the high-level aspects of the languages and simply assumed the existence and properties of the simple components. Landin argued that his reason for being less interested in the low-level objects was that it could allow these to be defined in different ways for better portability or application to different problem domains. Another division was between those, like van Wijngaarden, who kept their pieces "concrete" (Landin borrowed the term from McCarthy), whereas he himself

[15] Böhm uses brackets rather than the more standard dot to indicate the scope of the bound variable, but this notational difference has no semantic implications.

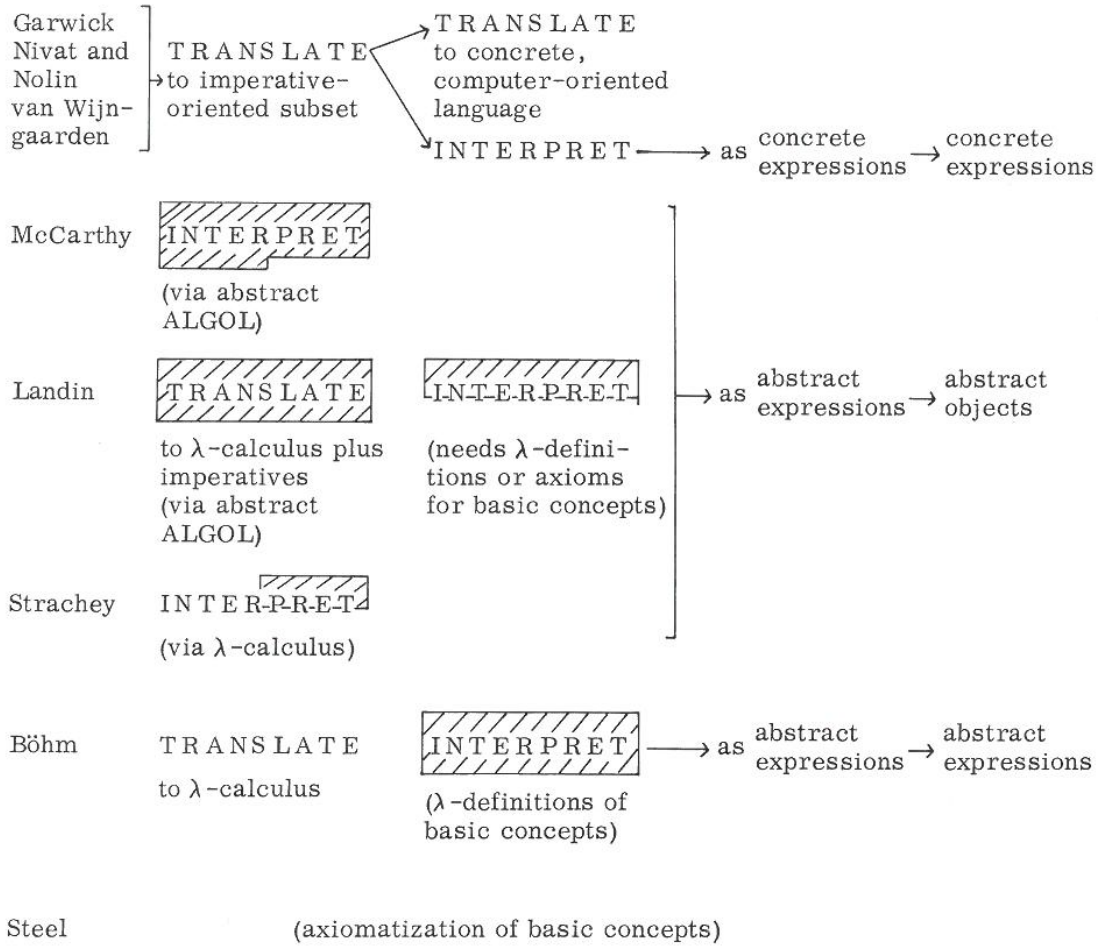


Figure 4.5: Peter Landin's categorisation chart of semantic approaches at FLDL.

was much more interested in striving for the greatest abstraction possible.

The final useful distinction is between translating and interpreting. McCarthy and Strachey, said Landin, were using an "abstract interpreter"[16] function to provide outcomes or meanings from their programs, whereas others were translating or compiling into another language or metalanguage. The use of formalism in these approaches is indicated by the shading on the diagram. Landin hinted he had a theory that there is little difference between the two, but did not have the time to develop this further.[17] Indeed, following Strachey's paper, McCarthy explained that he thought the really important part of semantics was describing the process

[16]This term gained a great deal of traction for the description of approaches like McCarthy's; this use by Landin seems to be its first appearance. Some writers, such as Jones (2003b,a), have attributed the term to McCarthy, but in fact he did not use it himself.

[17]In a recent paper, the current author argues this point from a technical perspective (Jones and Astarte 2018).

of computation, which is to say the transformation of the variables in the program, a sentiment with which Strachey (1966a, p. 219) agreed. Landin commented that Steel "must be annoyed" by his chart, because he could not be easily fit into any of the categorisations; but as previously mentioned, Steel saw a program as "a complex transfer function from initial state to final state" (Steel 1966b, p. 36). This view was, ultimately, not terribly different to that of McCarthy and Strachey.

Landin clearly thought the conference a very valuable experience, talking about it in glowing terms in his 'Reminiscences' talk at the Science Museum in 2001, although he did say that of the talks given, "some [were] sensible, some silly". He also noted that there was "absurdly" a vote on "the best method" presented at the conference—and his won (Landin 2001). This sentiment was echoed by Mike Woodger (1964) in a letter sent to Ershov later in 1964, stating "[Landin's] paper at Baden was one of the best". Certainly it was one of the most completely realised compared with the preliminary nature of many others; as noted in Section 3.2, he had already been working on the ideas for a few years by that stage.

More is said about Strachey's paper 'Towards a formal semantics' (Strachey 1966a) in Chapter 6, so the current discussion is limited to specific points relating to FLDL. Strachey had become interested in programming language description through working with Landin and also the CPL project. The painful experience of trying to pin down a compiler for that language, which was large, complex, and full of novel ideas, had led Strachey to wish for a more robust and rigorous way to approach language description. Strachey had been happy to accept an invitation to the conference, but as he was still working as a freelance consultant at the time he would have struggled to meet the costs alone. However, he applied to the Royal Society for a travel grant and, perhaps thanks to his well-known family and connections with the Society's Lord Halsbury, was awarded £40—likely enough to cover his entire trip.[18] A photograph of Strachey speaking at this conference can be seen in Figure 4.6.

A few of the comments after Strachey's talk concerned his ideas about the syntax of line termination in programs, which predictably did not interest him very much. Garwick asked why Strachey preferred to give semantics directly to a full and complex language (Strachey had used CPL for all the illustrative examples in his paper) rather than translating into a smaller subset language first, as that would have simplified the semantics. Strachey replied that he found commands were rather more

[18] This is documented in a pair of letters in the Strachey archive (Strachey 1964b; Martin 1964).

Figure 4.6: Christopher Strachey speaking at FLDL. Photograph from Auerbach (1986a, p. 82).

difficult to manipulation than functions; as common mathematical objects, the principles of function application were well known, and so it was worth making the translation early. He reiterated the point in response to Samelson, who had argued against the formalism and abstraction of Strachey's approach. Samelson said that people know what programs mean because they have experience with programming languages; and computers provide the ultimate semantics because they actually perform the manipulation of objects and produce results. Strachey countered by saying that it is very difficult to work out whether two machines which operate in different ways will produce the same results. By translating both to functions, the task becomes much simpler, as it is considerably easier to determine whether two functions are equivalent. These, and the interaction with van Wijngaarden where Strachey argued against the former's removal of functions in favour of procedures, clearly demonstrate that Strachey was convinced that a functional base for semantics was the best approach, something that would go on to characterise his denotational semantics.

Paul (in Strachey 1966a, p. 218) had a rather interesting comment: he urged caution in the field of programming language description and in particular paying too much heed to the views of mathematicians, who he viewed as the users of such formalisms, saying "A great deal of harm has been done in the past because the machine designers would not listen to us - the programmers - who wanted to use the machine." Strachey's reply seemed to be favouring boldness in its approach,

and was evidently popular: “In an unfamiliar situation like this one, you shouldn’t be too much bound by their first reaction. [Applause]” (Strachey 1966a, p. 218). This is the only example in the transcription of the discussions where applause is recorded, demonstrating that—despite the numerous occasions in which criticism was levied at nascent ideas in presentations—the attendees were generally in favour of the exciting aspects of a newly developing field of study.

Strachey was generally a vigorous and enthusiastic conference participant, interacting with most of the presenters who discussed programming language description. One particularly interesting moment came in a discussion of McCarthy’s approach to the state of his computation, into which he had folded the program counter metavariable. Strachey said he much preferred a separated approach to program positions, values, and declarations, handling each as individual objects. This presaged the eventual separation of the environment, store, and continuation objects as parameters to the meaning function in denotational semantics; but these notions were not yet apparent in Strachey’s own paper at FLDL. While a number of the finer points of what would become denotational semantics were lacking—as indeed was the fundamental mathematical foundation—one notion that shone through with extreme clarity was Strachey’s fervent belief in the function as the most important basic object to describe computation.

4.3.6 Session 6

The final session was held on the last Friday afternoon, and was the only session not to have a strong unifying theme. The three papers, delivered by Peter Ingerman, Brian Napper, and Saul Gorn covered a variety of different topics, although there was some commonality around the idea of self-generating languages.

Ingerman’s talk, ‘The parameterization of the translation process’ (Ingerman 1966), was an attempt to address the universal language question by providing a universal assembly language; such low-level languages were the topic of much of Ingerman’s previous work. This was discussed by means of a translation system that would take as parameters details of the machine on which the language would be used, and properties of the high-level language from which the system would translate—hence ‘parameterization’ in the title. The translation was applied in a symbolic fashion and used a modified form of BNF for its specification. Steel (in Ingerman 1966, p. 230) noticed that the work was in an intermediate and unfinished stage,

commenting “I feel as if I had come in at the beginning of the second act of a three-act play, and somehow the curtain came down at the end of the second act and never quite came up again”. Ingerman accepted this criticism, saying he was still “flying by the seat of [his] pants” (Ingerman 1966, p. 230). Ingerman did go on to continue to work with translation, publishing more papers and a book on generalised assembly languages and syntax-oriented translators.

The paper presented by Napper was entitled ‘A system for defining language and writing programs in “natural English”’ (Napper 1966). This had a roughly similar sentiment to the previous talk, and presented a ‘compiler compiler’. This was a generic compiler to which you could feed both a program and a definition of the language in which the program was written. The compiler itself was written in an ‘English-like’ language, with the intention being to making programming more intuitive. In particular, output and debugging information could be made more intelligible for a human. Again, this work was in something of a preliminary stage, with discussion responses showing that Napper still had plenty of work to do on the system to make it more usable.

The very final paper of the conference was delivered by Gorn, and was not on the initial May 1964 draft programme (IFIP 1964a), although at that time Gorn had accepted an invitation (Zemanek 1964b). The talk as eventually given was called ‘Language naming languages in prefix form’ (Gorn 1966) and was a language theory paper. The ‘prefix form’ of the title was somewhat similar to Polish notation, but was prefix-based rather than the usual suffix application of Polish notation. The paper built on Gorn’s previous work and could be applied to various languages including two-dimensional ones such as flow charts.

This particular part of Gorn’s work is not terribly relevant to the story of programming language description, but it is worth reviewing some of Gorn’s other contributions. A photograph of Gorn can be seen in Figure 4.7. As described in Section 2.2, Gorn was one of the early workers pushing the language metaphor for programming (Nofre, Priestley, and Alberts 2014, pp. 52–3). He was part of the American Standards Association (ASA), the precursor of the American National Standards Institute (ANSI), in their task group X3.4 “Language Structure Task Group”. It was this group’s job to define the terms necessary for discussing ‘mechanical languages’ (i.e. machine languages and coding systems as well as programming languages), and in a report for this group, Gorn (1961a, p. 337) presented a good early definition of syntax and semantics:

Figure 4.7: A photograph of Saul Gorn taken in 1966.

> *Semantics* is concerned with the relationship between symbols and their "meanings". We will interpret the word *meaning* also in a mechanical context. The *meaning* of an expression in a mechanical language at a given moment might therefore be a storage location if the interpreter is an "address selector", or a command if the interpreter is an "order type decoder" or a "subroutine caller", or any other expression if the interpreter is a "table-look-up substitutor", or, more generally, a "replacement producer". *Syntax* is concerned with the relationships of expressions among themselves, independent of their *semantic* "content" or *pragmatic* "context".

At this time, Gorn was working on 'specification languages', and presented a paper on that subject to the ASA X3.4 committee (Gorn 1961b). Of particular interest was his split between 'recognitional' and 'generative' languages, where the former was concerned with identifying objects based on their qualities or behaviours, and the latter describing the compositional methods of languages. This distinction later became useful to those working on the semantics of programming languages, and matches McCarthy's 'analytical' and 'synthetic' syntax respectively. Gorn's 1961 paper was largely based on mechanical and graphical descriptions, and the majority of presented approaches concerned language syntax. However, the three logical lan-

guages mentioned towards the end of the paper approach ideas of semantics, but did not possess the richness of other approaches. Furthermore, this paper did not see much impact on later semantics research—even amongst those, like Rod Burstall[19] who used an approach based on logic themselves.

In his summary of the 1963 *Working Conference on Mechanical Language Structures*, Gorn (1964) used terms such as "background machine", "idealized machine", and "interpreting machine". However, he was not referring to an abstract machine of the sort imagined by Landin, but rather the conception of the operating environment in the mind of a reader of a program who is trying to envision its meaning or effect. This environment was concerned with factors such as control counters, instruction registers, and address selectors: in other words, very similar to a physical machine. Gorn argued that this machine is always involved in the mental interpretation of a program or language, and that in order to fully specify the meaning, these features should be clearly defined as well, writing "I am one of those extremists who feel that it is impossible to separate a language from its interpreting machine" (Gorn 1964, p. 133).

Realising this is central to understanding the way in which semantic techniques were interpreted by Gorn, and others at the FLDL conference and working in the field at the time who shared his very practical view of programming. This makes sense of Gorn's comments on van Wijngaarden's paper when he asked about efficiency of programs manipulated in this way, and argued that 'raw data' in programs (such as comments) should have a semantic meaning as well, because such data would still take up storage space. Approaching the correspondence between language and machine from the mechanical perspective, Gorn queried whether languages could be automatically derived from Elgot's machines, arguing "This is the sort of thing you want to get out of a formalism" (Gorn, in Elgot 1966b, p. 176). The inextricable linking of machine and language was not an uncommon view held by practitioners at the time, and explains a good deal of the resistance felt by those like McCarthy, Strachey, and Landin who tried to use as much abstraction as possible in their work.

Despite this, Gorn clearly had a strong desire for unification between varieties of language analysis, as shown when he commented on Caracciolo's FLDL paper, saying "It is very inconvenient to specify all syntactics without semantics" (Gorn, in Caracciolo di Forino 1966, p. 47). Gorn argued that as soon as rules need to be combined, semantic territory is entered. Following the idea further, he contested

[19] See Section 7.4.

that it was impractical to consider semantics without pragmatics; he also asked a similar question of Čulík at FLDL. Gorn was known for being prominent in pragmatics, which writers such as Lucas (1978) and Zemanek (1966) considered as an important part of understanding the meaning of languages. At a later conference in August 1965 on *Programming Languages and Pragmatics*, Julien Green noted:

> When the word "pragmatics" is used in relation to the work we do, the name of a particular individual comes to my mind. I am certain you all think of the same name: Saul Gorn.
> (Green, in Woodger and Green 1966, p. 224)

Gorn (1961b, p. 337) defined pragmatics as "the relationship between symbols and their 'users' or 'interpreters' ". Zemanek (1966, p. 139, quoting Morris 1946) also provided a definition: pragmatics "deals with the origin, uses, and effects of signs within the behaviour in which they occur".[20]

This ends the discussion of the papers presented at *Formal Language Description Languages* and the people who presented them.

4.4 Other attendees

The conference, however, was attended by more people than just those who gave papers, and some of these made interesting comments or are worthy of discussion for other reasons.

One such attendee has already been mentioned a few times: Edsger Dijkstra was invited as a member of WG 2.1, and was not shy about making his opinions known. He made an important comment on Strachey' talk: Strachey had argued that the term 'variable' means something different in a programming context than in ordinary mathematics. In maths, a variable's value does not change, but is not determined during the writing of the text. However, in programming, variables can change their value over time, and so only their *current* value can be considered (Strachey 1966a, p. 201). Dijkstra argued that the fairest way to consider variables is as a succession of constants over time. He added: "of course it is very important that a variable

[20] In classic Zemanek style, he did not start by defining the term as used in computing. The section of the paper on pragmatics begins instead "Pragmatics is a word which any Austrian would associate with the history of the 18th century..." and described the Emperor Charles VI's 'Pragmatic Sanction', the decree allowing his daughter Maria Theresa to succeed him on the throne (Zemanek 1966).

in a programming language can change its value, but it is equally important that between assignment statements its value remains a constant" (Dijkstra, in Strachey 1966a, p. 217). Strachey did not reply to this remark at the time, but the idea would crop up in his later work, where he argued the changing value property of (imperative) programming caused some difficulty to a purely mathematical approach to defining programming languages. Furthermore, the task of proving that variables do not change between commands would be one with which many practitioners of semantics would struggle.

Dijkstra became particularly notorious for his letter to the *CACM* in which he argued that go to statements could be 'considered harmful' (Dijkstra 1968). It appears the germs of this idea were emerging during the *Formal Language Description Languages* conference: Doug McIlroy (2018) recalls that he was sitting at a table with Dijkstra after van Wijngaarden had given his talk showing a mechanism for eliminating go to statements. Dijkstra said "I've been playing and writing all kinds of programs; I think the go to doesn't matter. It would be better not to have it. But not because you can substitute it out of existence; because it makes better software, better structured programs."[21] Dijkstra may already have been having ideas about the removal of go to statements before the conference, but the influence of van Wijngaarden's approach should not be overlooked.

Klaus Samelson, long-time collaborator with Bauer and firmly avowed "safetishist" in programming languages (as described in Section 3.3), was rather sceptical about the core notion of defining semantics. He asked McCarthy, after the latter's talk, whether he would kindly now define the semantics of his metalanguage. In other words, Samelson was arguing that you could not solve the problem by translation. His view of semantics (in McCarthy 1966, p. 11) was quite different:

> I would rather avoid the word "semantics" altogether. Semantics is what we have in our heads; as soon as we write it down it's not semantics anymore.

Samelson also made a good point in response to Landin's paper, noting the contortions of lambda calculus Landin was forced into using in order to cope with certain aspects of ALGOL such as block structure. He said "I just want to say here that the

[21] As with all oral history interviews, some care should be taken when examining the use of terminology especially. It is possible that Dijkstra used the term 'structured programs', but given that his paper 'Notes on structured programming' (Dijkstra 1970b) was not published until 1970, it is more likely that McIlroy is remembering this term and re-applying it to the older conversation.

two concepts don't fit together too well - λ-calculus and ALGOL - and the question is still open as to which one is better" (Samelson, in Landin 1966c, p. 292). On reflection, Samelson is certainly right—many of the people using lambda calculus in some way or another, like Landin, Böhm, and Strachey, needed all kinds of tricks to make a framework that would fit programming languages. Fundamentally, however, these people believed the trade-offs were worth it, but it is not surprising that the purist Samelson would disagree.

One very significant group of attendees was the IBM Science Group under Zemanek in Vienna. The entire team was in attendance at FLDL, many as scientific secretaries. Crucially, this included Peter Lucas and Hans Bekič, who many of the lab members[22] agreed were the "real geniuses" behind the language description work the lab took on throughout the rest of the 1960s. Prior to the conference, there was no real knowledge of the field of programming language description, and so being exposed to all of this work had a profound effect on the group. Zemanek (1981, p. 14) recalled:

> For the collaborators of the Vienna IBM Laboratory it was, however, a magnificent opportunity to meet all the people active in the field of formal definition. The contents of the papers (of course some more than others) were the basis for the development of the Vienna Definition Method to be applied for the formal definition of PL/I, not only the syntax, but also the semantics.

Very soon after the conference finished, the group took on the task of providing a formal description of IBM's massive PL/I programming language (Lucas 1981, p. 549), and the group was officially transformed into an IBM research laboratory in December 1964 (IBM Informationsabteilung 1964). Bekič (1964), almost immediately after the conference, began sketching out methods for providing the semantics of languages, publishing a technical report in that December as well. More on how the story of the Viennese work on language description came out of FLDL is presented in Chapter 5.

4.5 Concluding the conference

The *Formal Language Description Languages* conference concluded with a banquet dinner on the Friday evening, after which there was an arranged speech. The speaker

[22] The author has spoken to Jones, Walk, Neuhold, and Maurer; all agree.

was Fraser Duncan, who had been proposed by Bauer (Utman 1964b, p. 12). However, the speech came very close to not happening at all.

Brian Randell, newly a member of WG 2.1[23] and present at FLDL in that capacity, already knew Duncan well as they had both worked for English Electric making ALGOL 60 compilers at Whetstone for Randell and Kidsgrove for Duncan. Duncan was also the editor of the *ALGOL Bulletin* at the time. Their experience as compiler writers put them both firmly in the 'practitioners' camp, and, as Randell (2016) explained, Duncan became somewhat disillusioned by all the complicated theory being presented at the conference:

> Fraser Duncan was a lovely guy, but could get really very het-up and morose if he didn't like what was happening. I would come out fighting and he would sulk. He got so fed up during the conference that he disappeared. For about a couple of days he wasn't seen. Since we were all staying at the same hotel, his absence was noticeable. It was becoming very noticeable because he was due to be the after-dinner speaker.

Duncan eventually returned on the final morning of the conference, having gone sightseeing in Vienna. He vowed that he would not make the speech, due to his distaste at the way the conference had proceeded, so Randell hatched a plan to convince him. Randell spoke to Zemanek's assistant, Norbert Teufelhart, who had a reputation for being a "real Mr Fix-it", and instructed him to feed Duncan enough whisky that he would give the speech anyway (Randell 2016).

Randell's plan worked, and a well-oiled Duncan gave an excellent speech entitled 'Our ultimate metalanguage'[24] in which he gently poked fun at all the formalists, pointing out that eventually formalism must give way to natural language when programming is discussed (Duncan 1966). He also included a generous number of single-letter identifiers to stand for various concepts in his speech, and it became

[23] Randell had been proposed as a member by Bauer at the May 1964 meeting of TC-2, to unanimous agreement (Utman 1964b, p. 10). This consensus was likely due to the very positive reaction to Randell and Russell's ALGOL 60 compiler they produced at English Electric Whetstone (see Section 2.2). In fact, Randell had already been nominated at the first meeting of TC-2 (Utman 1962a, p. 5), but had been unable to accept while working for English Electric. Two years later, Randell accepted an offer to go and work for the IBM Research Centre in Yorktown Heights, New York, and part of the condition he had for accepting was that IBM support his joining WG 2.1. Not only did IBM agree to this, they also sponsored Randell's attendance at the next meeting, the one wrapped around FLDL, which happened to fall before Randell even moved across the Atlantic. The generosity with which money was provided, coupled with the considerably larger salary, was a real change for Randell from the tighter finances of the British state-owned English Electric (Randell 2016).

[24] The title is a quotation from Naur (1963) in which he urged people to pay attention to the natural language used when discussing notations.

increasingly obvious that these were the first letters of surnames of attendees (Randell 2011). Duncan joked about the overuse of certain formalisms, asking "Is your Chomsky really necessary?" (Duncan 1966, p. 298), and argued that 'language' meant something different to everybody. He wished the word had never been applied to computer code, but saw it as a natural result of "[a] little puff to the verbal inflation that goes on all the time. [...] the most baroque example of this point that I can quote [...] obliges the IBM Laboratory Vienna to put "Austria, Europe" on their paper, when "Austria" should be sufficient" (Duncan 1966, p. 296). (A footnote adds "Comment of the IBM Laboratory Vienna: We do not like to receive our US-mail via Australia".)

When the proceedings came to be written up, a sober Duncan carefully removed all the jokes and jibes—and the editor promptly put them all back in (Randell 2016). This was despite Duncan's best (or perhaps simply humorous) attempt to prevent such an occurrence; his speech began:

> Everything I am about to say is my own personal copyright (that is to say the copyright of P. Z. Ingerman) and may not be reproduced by IFIP or any other sordid body without the express written permission of Saul Gorn.
> (Duncan 1966, p. 295)

A footnote to the above sentence reads:

> The Editor has in his possession a holograph with the following text: "IFIP TC2 hereby has my permission to print Fraser Duncan's speech without change. S. G. 20 September 64"
> (Steel, in Duncan 1966, p. 295)

Duncan, a photograph of whom can be seen in Figure 4.8, was known to be humorous; Zemanek recalled another of his jokes:

> *ALGOL-like*, by the way, was also a word that became a fashion at and through this conference with the culminating proposal or joke—the distinction between proposal and joke was not always clear in WG 2.1 and TC 2—that *ALGOL was not an ALGOL-like language.*
> (Zemanek 1981, p. 14)

Specifically, Duncan had commented on Steel's paper, asking "Is ALGOL 60 an ALGOL-like language? [Laughter] A lot of people would give the answer 'No'. I

Figure 4.8: Fraser Duncan at FLDL. Photograph from Auerbach (1986a, p. 82).

would myself" (Duncan, in Steel 1966b, p. 34). Steel himself also said no to this question.

The end-of-conference dinner was not the only social activity which took place. As well as presentation sessions, the conference schedule included many diversions. The attendees were given Wednesday and Thursday afternoons off for excursions, enjoying an exhibition of 18th Century automata on the first of those days, and one of Romanic art and the Wachau Valley on the second. There were also two visits to the Vienna State Opera scheduled for both evenings (IFIP 1964b). The touch of Zemanek is clearly seen here: art, classical music, and Austrian history were all great joys of his. The excursions were not just for fun and for Zemanek to show off, however. They also served as more time to discuss the work of the conference, as Steel (1966c) noted: "Unrecorded but equally valuable were the informal conversations occurring during meals, on the excursions, and far into the evenings."

The meeting of WG 2.1 held directly after FLDL finished affords an excellent view into the way the conference was viewed by language practitioners at the time. The reception was, in fact, rather lukewarm. Naur (in Utman 1964c, p. 7) indicated he was "somewhat discouraged about inapplicability of conference papers to the WG 2.1 objective of defining the future ALGOL". Randell (in Utman 1964c, p. 7) felt that "perhaps the conference was too far ahead of us". The proliferation of different approaches was confusing; some members of 2.1 had hoped for an obvious best ap-

proach that could be used for ALGOL X or Y. Seegmüller (in Utman 1964c, p. 8) was particularly pessimistic: he "concluded from the conference that it will take 15–20 man-years to design AY, and he does not see how WG 2.1 can do it with the committee approach". This remark would turn out to be somewhat prescient: when the next version of ALGOL was eventually designed and released, it caused a rupture in the committee.[25]

Ingerman, however, reminded the committee that FLDL was not intended as a solution to WG 2.1's problems (Utman 1964c, p. 8). McCarthy, too, was more optimistic, seeing a lot of progress in the field over recent years. Finally, closing out the WG 2.1 meeting, a vote was taken on how useful the papers at FLDL had been for 2.1's purposes. Böhm and Landin were joint winners with eight votes each, followed by van Wijngaarden with six (Utman 1964c, p. 9). This may be the vote recalled by Landin in his reminisces that he remembers winning (Landin 2001), although he was not present at the 2.1 meeting and he was not the sole winner. It is interesting that Böhm got such a positive result given the relative lack of engagement with his ideas in the discussion session at the conference; both his and Landin's approaches were also very theoretical in nature, which may do something to counteract the suggestion that the 2.1 practitioners were uninterested in such ideas.

Looking back now, it is clear to see how important and influential *Formal Language Description Languages* was on the field of language description. A lot of foundational papers in semantics were presented for the scrutiny of an international audience, including those of van Wijngaarden, McCarthy, Strachey, and Landin. There were many interesting interactions between these practitioners and people outside this very specialised group; even if the tone of many of the comments was one of criticism and lack of understanding, the interchange of ideas cannot have been anything other than valuable. The Vienna group benefited particularly from the conference, as can be seen in the work of Bekič in December of that year, which contained clear influence from papers delivered at FLDL.

Despite Duncan's jokes, he too ultimately praised the conference, ending his speech in a tone of celebration:

> It has for each one of us, I think, been an extremely valuable meeting - we have been provided with excellent opportunities for talking to people who before were just names and for getting glimpses, at least, into prob-

[25] More on that story is given in Section 7.1.

> lems and approaches to solutions which could well become important in more than one corner of the field of mechanical languages. We have not actually got all or perhaps any of the ultimate answers. But we have a lot of ideas of where to look and where not to look.
> (Duncan 1966, p. 299)

Bauer, too, was very positive about the conference, saying "the arrangements and conduct of the meeting had been excellent", a sentiment echoed by the rest of TC-2 (Utman 1965, p. 3).

September 1964 marked a very important point in formal semantics, and can be regarded as the time that a community of researchers formed out of a group of individuals scattered across the globe. This is clearly demonstrated by the formation of TC-2's second working group, WG 2.2, tentatively titled 'Language Description', discussion of which began at the next TC-2 meeting after FLDL (Utman 1965).[26] Perhaps the greatest impact, however, was on the IBM Vienna Science Group, which is the topic of the next chapter.

[26] See more about this working group in Section 7.2.

CHAPTER 5

Abstract interpreting machines at the IBM Laboratory Vienna

In the first few years of the 1960s, IBM began a huge project to replace its entire computer product range, a risky manoeuvre later dubbed "IBM's $5 billion gamble" by *Fortune* journalist Tom Wise (quoted in Campbell-Kelly and Aspray 1996, p. 127), resulting ultimately in the System/360 series of computers. To accompany this new product line, a general-purpose programming language was developed: initially known as New Programming Language, and later Programming Language One, or PL/I, the language was an equally monumental work. In 1964, the task of writing a full formal description of this giant language was passed to the IBM Vienna Laboratory. The diverse group, drawing on their previous experience of implementing ALGOL 60 and led by Heinz Zemanek, a charismatic and skilled manager, accomplished this massive task in 1966. They had defined the entirety of PL/I, but the result was so vast and intricate as to be almost unusable.

This chapter traces that development and reactions to the end product. Section 5.1 gives some background on the group and its leader prior to the Laboratory foundation. Section 5.2 explains the beginnings of the group working at IBM. Section 5.3

discusses PL/I and the reasons it presented a difficult task for formal description. Section 5.4 documents the first stages of the development of formalism. Section 5.5 explains the details of the formal description. Section 5.6 looks at the reaction this work had within IBM and beyond. Section 5.7 concludes the chapter and looks at applications of the work. A timeline of the major events detailed in this chapter can be found in Figure 5.1.

5.1 Before the Lab

The leader of the work of the Vienna Group during this period was Heinz Zemanek (1920–2014), who assumes a similar role as the 'founding father' figure in computing for Austria as van Wijngaarden in the Netherlands, or Bauer in West Germany. Zemanek was always a very proud citizen of his country, as is evident from the first words he spoke in an interview with Aspray: "I am a true Austrian. I was born in Vienna. My ancestors came from Austria in the old sense, namely $\frac{3}{4}$ of them from the part which is today Czechoslovakia and $\frac{1}{4}$ from a little east of Vienna" (Zemanek and Aspray 1987, p. 3). Links to Vienna in particular were always important to Zemanek: his grandmother had worked at the Spanish Riding School, home of the *Lipizzaner* horses, at the imperial palace in Vienna.[1] Zemanek studied engineering at the request of his father who had been an accountant and saw that it was the engineers who really understood the businesses (Zemanek and Aspray 1987, p. 4). He grew up in a musical family and learnt piano as a child, a skill which served him well during the Second World War as he took on the role of company musician, earning himself perks such as more food (Zemanek and Aspray 1987, p. 5).

This would have been particularly welcome to Zemanek, whose poor upbringing meant that his attendance at university was very uncertain. His family managed to scrape enough together that he was able to attend the *Technische Hochschule* in Vienna[2] from 1938–1944 (the duration lengthened because of the war). Zemanek chose to specialise in telecommunications engineering because he was less skilled at producing the detailed drawings needed in other kinds of engineering, and got involved particularly in low-frequency telecommunications because he felt there was a better connection to the underlying theory (Zemanek and Aspray 1987, pp. 7–8).

[1] Further evidence of Zemanek's links to Austrian leaders is given in an anecdote from Jones (2015, p. 58): "I used to arrive early to work so had to be told to ignore the car of the *Bundespräsident* parked outside Parkring 10 because Franz Jonas was learning about computers from my boss."

[2] Later the *Technische Universität Wien* (TUW), Technical University of Vienna.

Figure 5.1: A timeline of important events in the story of programming language semantics in Vienna.

Figure 5.2: A portrait of Heinz Zemanek painted by Konrad Zuse. Taken from Zemanek's memorial website, `http://www.zemanek.at/`.

Zemanek was always very proud of Austrian and German computing and previous academic and philosophic work, frequently linking his and his team's work with historical greats. McIlroy (2018) remembered visiting Zemanek in 1964: "He took me to his office ... and he had, in the corner, a piece of a Konrad Zuse machine. At least the drum from the machine". Zemanek did indeed know the German well: Zuse even painted a portrait of him, which can be seen in Figure 5.2.

In 1947, Zemanek joined the TUW as an assistant professor at the lowest grade. The chair of the department had been prosecuted by the Nazi government and fled to Holland; on his return he was not interested in overseeing projects in great detail and let his assistant professors run things. This meant that Zemanek now had a good deal of freedom, as well as a "staff" for his work: the diploma students. Zemanek had an unusual working model in which his students worked in teams on larger projects, but each produced their own written piece of work on which the assessment was made; this concession was made so the university would allow group work (Zemanek and Aspray 1987, p. 17).

Around the early 1950s, Zemanek shifted his focus from digital communications technology into computing, and at the TU his teams made a few experimental mechanical relay machines. Zemanek had travelled in France towards the end of the 1940s and seen a number of groups who had been unsuccessful at making computers: "I knew, for another time, to be very careful before you enter a computer enterprise.

I had all the necessary warnings to keep back until the right time was coming. And the right time was coming when the transistor was around" (Zemanek and Aspray 1987, p. 22).

In Spring 1954, work started at the TU on a fully electronic computer, the Mailüfterl. This was an early European machine which had the distinction of being fully transistorised: no tubes or relays were used in its construction. A small grant for the project was provided by Zemanek's boss Eugen Skudrzyk, but the 30,000 schillings was an order of magnitude too small (Zemanek and Aspray 1987, p. 27). Zemanek travelled around Europe gathering information and making partnerships with industry; he took as much support as he could from these industrial partners, especially Philips. Ultimately, the majority of the financial support came from a banker, Josef Joham, whom Zemanek had known from his time in the Boy Scouts. Joham contributed 250,000 schillings in total to the project (Zemanek and Aspray 1987, p. 42).

The Mailüfterl team was built from diploma students who worked on various specialities for their qualification, and then joined the team full-time after graduating (Zemanek and Aspray 1987, p. 28). They were:
Kurt Bandat: core store
Rudolf Bodo: layout (now known as architecture)[3]
Viktor Kudielka: drum store
Kurt Walk: transistor physics
Peter Lucas: programming (joined later)
Eugen Muhldorf: tape reader (part time).

A lot of the construction of Mailüfterl was innovative for the time. The magnetic drum store, in particular, was novel, and had to be trialled and developed extensively in-house. The team had to develop their own kind of plastic to coat the read heads. This contributed to an atmosphere of innovation and problem-solving (Zemanek and Aspray 1987, pp. 29–31).

Other problems for the group came from components which were not of the highest quality. It was this in particular that led to the computer's rather unique name.

> It was clear at the time that no other company than Philips could be the source for the transistors. It was, furthermore, clear that Philips was not yet in the production of fast transistors. What they could offer were

[3]Bodo was financed by Zuse, with the agreement that he would join Zuse's company after the computer was finished.

> relatively slow transistors intended for hearing aids, where no high cut-off frequency is necessary. This fact explains the name of my computer, *Mailüfterl*.
>
> I was saying at the time, "We are going to build a transistorized computer in Vienna. But since I am going to have to use hearing aids because I depend on the gifts, it will be a slow one. In other words, it will not be a typhoon, whirlwind, or hurricane,[4] but a nice Viennese spring breeze would be sufficient for our purposes." I said "Mailüfterl" in German. And then the German colleague said, "That's a fantastic name. You should keep it." Against the resistance of many very serious people (my prime minister included, who has a negative remark about my naming it in his memoirs) I kept the name and, meanwhile, it's [sic] tradition. (Zemanek and Aspray 1987, p. 32)

The dates during which work progressed on Mailüfterl are somewhat arbitrary; Zemanek said: "The construction work went on from 1956 to 1958, and being the *Mailüfterl*, it had to be in both cases May 1956 and May 1958; but it is almost correct" (Zemanek and Aspray 1987, p. 34). However, he had something of a penchant for picking appropriate dates after the fact, so this should be taken with a degree of scepticism. Construction was likely finished earlier than May 1958, as the spring edition of a survey by the US Office of Naval Research (Goldstein 1958, pp. 14–15) included Mailüfterl in its list. It was one of only four European computers mentioned; this contrasts with van Wijngaarden's survey in 1955[5] which implied there were plenty in Europe. The ONR survey failed to mention any in the Netherlands, which suggests it was not complete.

Following the completion of Mailüfterl, Zemanek realised a small group at the TUW had no chance of competing with major hardware manufacturers and so made the shift towards software (Astarte and Jones 2018, p. 89). Walk (2002, p. 78) remembers just how important programs were seen to be at that time: they represented the transformation of problems from the human world into the computer world. One of the first practical applications of the Mailüfterl was for a music theorist, who wanted to calculate the number of twelve-sound series in which each interval appears only once (Zemanek and Aspray 1987, pp. 36–7). Zemanek listed it as the first application, which may be slightly suspect due to his love of music and fondness for auspicious events. However, Walk (2002, p. 77) also remembered it as being "one of

[4] These were the names for other contemporary computers: the Whirlwind was developed at MIT in the late 1940s, and the RCA Typhoon was finished in 1951.

[5] Mentioned in Section 3.3.

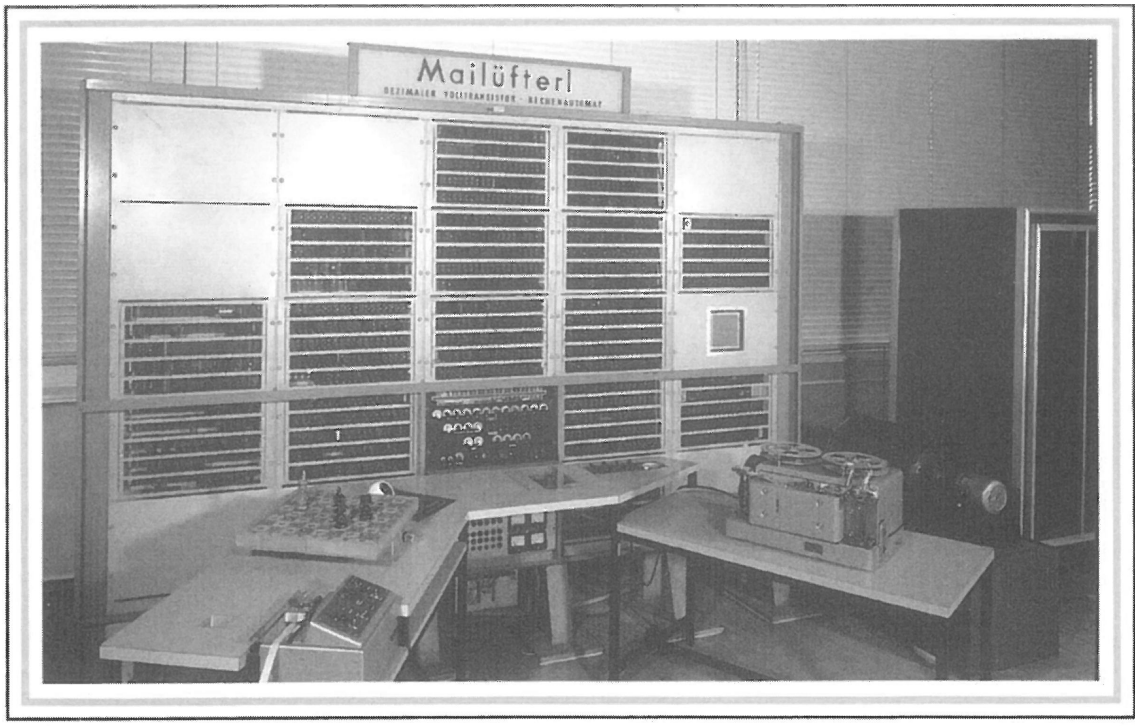


Figure 5.3: The Mailüfterl computer on display at the Technisches Museum Wien. From Kurt Walk's (2015) memorial piece.

the first application programs" and notes that it took all night to run despite Lucas' best efforts at optimisation. Zemanek recalled that they also tried to implement a chess program, but were too ambitious at that stage and did not manage to achieve anything. Nevertheless, the Mailüfterl was displayed at the Technisches Museum Wien alongside a chessboard, as can be seen in the postcard displayed in Figure 5.3.

In order to exploit the computational power of Mailüfterl properly and prove its worth as a machine, the group needed to implement a higher level programming language. Originally, FORTRAN was considered, but it was an IBM product and the company provided no support for building a compiler, perhaps because of their desire to keep industry secrets or sell their machines on the strength of their software. Bauer suggested to Zemanek that his group try implementing ALGOL instead, assuring them that the implementation would be trivial (Zemanek and Aspray 1987, p. 35). Zemanek and his team began to be involved in the discussions leading up to the development of ALGOL 60, as shown by Naur (1981b, pp. 121–2) recording contributions from the Mailüfterl group and Zemanek in particular.

The task of creating an ALGOL 60 compiler for Mailüfterl was not as trivial as Bauer had claimed it would be. Lucas, as lead programmer, had to dive deep into the intricacies of programming language theory and compilers, describing the experience later as "painful" (Lucas 1987, p. 2). In particular, hidden ambiguities and complexities in the language were uncovered by the implementation effort, and this began to lead to hopes of using some more rigorous and formal techniques:

> Unlike users of a language, who might only need a partial knowledge of the language, an implementer needs a complete understanding of the language to be implemented. At the time (1960), this was a rather demanding prerequisite. The combination of blocks, procedures as arguments, general goto statements, and recursion was especially difficult to master. It was hoped that a formal model could serve as the basis for a systematic design and justification of execution environments and compiler algorithms.
> (Lucas 1981, p. 550)

The ALGOL 60 compiler was completed in 1961 (Walk 2002, p. 80) and the experience was invaluable to Lucas and Hans Bekič, who had assisted Lucas on the project while still a PhD student. The two documented their experiences and lessons learnt in a technical report released the following year (Lucas and Bekič 1962). It shows insight into the difficulty of implementing a programming language without a "complete and unambiguous" definition for reference, particularly the semantics. The complexity of the Mailüfterl machine code also meant that the programmers had to learn to be very ingenious, skills that would serve them well in the next stage of their activities (Zemanek and Aspray 1987, p. 39).

5.2 Move to IBM and the Science Group

In 1961, Zemanek was beginning to feel that his group was growing too big for the technical university. It could not afford to hire all of Zemanek's students as assistants once they had finished their diplomas, and Zemanek required a large group to be able to work on the tasks he thought important (Zemanek and Aspray 1987, pp. 41–2). At the *International Congress in Information Processing* in 1959, the precursor to the IFIP Congresses, Zemanek had been introduced to Emanuel Piore, Director of Research at IBM, and the two became strong partners: Zemanek described Piore as his "protector" during his time at IBM (Zemanek and Aspray 1987, p. 41).

Zemanek needed an industrial home for his computer and the group of people around it, and investigated Siemens as well as IBM. In some respects Siemens was preferable, especially as a German-speaking company, but they would not have allowed the group to stay in Vienna, offering a home instead in Munich. IBM, although not German-speaking, was larger and richer, and could provide accommodation in Vienna (Zemanek and Aspray 1987, pp. 42–3). Another strong advantage was that IBM granted access to the American stream of knowledge, which was a good way to get the best ideas in computing moving into the country—and of course Zemanek wanted Austria to benefit from this (Zemanek and Aspray 1987, p. 74). For these reasons, Zemanek chose IBM, although he did acknowledge the interest was mutual:

> At the same time IBM had gotten interested in my group and, therefore, the interconnection grew into two directions. If you listen to IBM people, they would say they called me. If I describe the history, I would say I selected IBM. To a certain degree both descriptions are true.
> (Zemanek and Aspray 1987, p. 42)

A further motivation on IBM's part was to complement its Austrian sales office. IBM was under pressure from national governments to spend money where it made money: reinvesting by creating a laboratory was a good way to address this.

Mailüfterl and its team moved to IBM offices on the Parkring in central Vienna on 3rd September 1961. The greater funding allowed Zemanek to hire more people, with around two dozen staff starting off the group (Walk 2002, p. 77). The team became known as the IBM Science Group Vienna, and reported to the Böblingen laboratory in Germany (IBM Informationsabteilung 1964).

This mention of different offices is worth highlighting: it is a mistake to think of IBM as one monumental organisation—although management was sometimes keen to present it as such. For example, IBM's president Thomas Watson Jr., following publication of a journalistic piece suggesting that the decision making during the development of System/360 was chaotic, "issued a memorandum suggesting that the appearance of the piece should serve as a lesson to everyone in the company to remain uncommunicative and present a unified front to the public, keeping internal differences behind closed doors" (Campbell-Kelly and Aspray 1996, p. 127).

In reality, different parts of IBM could have quite radically different ideas and styles of work.[6] Zemanek explain that there was tension between the parts of the company

[6]See also the discussion of Rochester's paper at FLDL in Section 4.3. There, Rochester

that preferred decentralisation and those that favoured centralisation. Parallel development in different locations could lead to multiple ideas for similar products, the best of which could then be chosen; but stronger central control and strictly divided work could lead to cost benefits (Zemanek and Aspray 1987, p. 73). This meant there was a certain atmosphere of competition between centres, with management frequently moving products and development streams.

In Europe in the 1960s there were six IBM locations: three large groups in Hursley, England, La Gaude, France, and Böblingen, Germany; and three smaller in Uithoorn, Netherlands, Lidingo, Sweden, and Vienna, Austria. Most of these, including Hursley and Vienna, were 'development laboratories': they made products to sell. However, Vienna's role was somewhat different: although not part of IBM's Research wing, it did not have a particular product responsibility (Endres 2013, p. 6). Instead, the office was simply conceived as a software laboratory in general (Zemanek and Aspray 1987, p. 72). This lack of expectation allowed the Vienna group to take more time on longer-form projects without an obvious commercial goal in mind. An early piece of work was on voice synthesis, or vocoders; once the idea was recognised as sufficiently workable, it was moved to Germany for development, then to France, and finally to America for actual production (Zemanek and Aspray 1987, p. 74).

The photograph shown in Figure 5.4 depicts a number of the early members of the Vienna Science Group. Again, Bruegel's *Tower of Babel* makes an appearance: it covered an entire wall of the conference room at the group's offices. Sometimes, after tough meetings, various senior people involved in IBM projects were linked with the figures in the bottom left corner as a way to poke fun at the enormous organisation (Astarte and Jones 2018, p. 87).

The Vienna Group was a mix of engineers and mathematicians, many of whom had studied under Zemanek at the technical university. Most ex-members of the group agreed that Lucas and Bekič were the real drivers of the technical concepts behind the language work;[7] Walk, one level higher in the management structure, was adept at recognising and promoting talent in other members—while also maintaining understanding of all technical work (Neuhold 2016). When Zemanek eventually left the Lab in 1976 to become an IBM Fellow, Walk took his place as Lab manager.

explained that even with the relatively simple concern of syntax description, there was significant disagreement between IBM locations.

[7] Personal communication with Jones; Maurer; and Walk, 2016.

Figure 5.4: An image of some of the early members of the IBM Science Group in Vienna. Likely taken around 1964. From left to right: (standing) Peter Lucas, George Leser, Viktor Kudielka, Kurt Walk; (seated) Ernst Rothauser, Kurt Bandat, Heinz Zemanek, Norbert Teufelhart.

An aspect of Zemanek's leadership many of his ex-employees remember is how supportive he was for his workers to get qualifications and find good jobs. He would try his hardest to secure new positions and assignments for departing employees, whether inside IBM or outside. Zemanek's help was especially needed because there were not many career opportunities inside IBM Austria, apart from sales and support. Any career step made away from the group would have to be a big one, requiring either a move of company or country (Neuhold 2016). Zemanek supported staff while they worked on acquiring PhDs, such as for Maurer who worked part-time at the Lab for two years from 1963 to 1965 while he finished his academic work.[8]

Neuhold (2016) remembered Zemanek as a caring boss:

[8]Personal communication with Maurer, 2016.

> I think for me, Zemanek was a higher-level manager who really cared for his people. It was the old type of entrepreneur: the people were important for him. The success of the Lab was not the top priority. Success was for the people to become something. In German, you say he considered the Lab in part as a *Durchlauferhitzer*. It's a boiler where the water flows in cold and comes out hot.

However, a number of the team—diplomatically—suggested that Zemanek's main skills were not those required for research. Maurer said that although he tried to hide it, Zemanek was not much of a researcher, and Walk agreed, noting that he had carefully avoided calling Zemanek a strong scientist in his obituary.[9] Instead, Walk (2015) referred to him as "*ein Meister der Kommunikation* [a master of communication]", adding "Er hat mit aller Welt kommuniziert, mit Universitäten und Firmen in damals Ost und West. [He communicated with the whole world, with universities and companies in what was then the East and West]."[10] Neuhold (2016) concurred, saying Zemanek's most useful aspect was his charisma: he made good connections and knew how to skilfully utilise them.

Zemanek was also skilled as a lecturer, a task he continued to perform at TUW throughout his career, as well as supervising students. Neuhold (2016) remembers that seeing Zemanek lecture was what made him consider applying to join IBM:[11]

> I think I only had one lecture during my university related to computers at all! Everything else was switching and antenna, and all kinds of stuff, and this was an introductory lecture by Zemanek. I still remember it: he came into the lecture hall, and said "How do you count to thirty-two with one hand?"
>
> And nobody understood! Nobody knew it, I mean we were like a hundred people, and so binary encoding was totally unknown. So this was the kind of way he had when he lectured.

Another important part of Zemanek's life was his involvement with IFIP, the International Federation for Information Processing. Zemanek had been part of the organisation from the very start, having taken his team from TUW with him to the

[9] Personal communications with Maurer and Walk, 2016.

[10] Translated from German by the present author.

[11] Coincidentally, Neuhold joined the Vienna group on the very day Mailüfterl broke down; in an interview, he said "I heard, the first day I was there, a loud noise, like [a high-pitched squeal] and somebody next to me said "Oh no, the last drum is gone!" And it never worked after that." (Neuhold 2016)

1959 ICIP Congress in Paris, an experience Walk (2015) explained was very valuable for them:

> Als wir 1959 mit ihm beim ersten ICIP Kongress in Paris waren, waren wir selbst verblüfft über die Unzahl von einschlägigen Leuten, die dort beireits zu seinen Bekannten zählten. Für uns war das selbstverständiglich eine Starthilfe für internationalen Zusammenarbeit und die Veröffentlichung von Ergebnissen.
>
> [When we were with him at the first ICIP Congress in Paris, we were amazed by the sheer number of relevant people who counted themselves amongst his acquaintances. For us that was obviously a head start for international collaboration and publication of results.][12]

When IFIP began as a fully-fledged organisation, Zemanek was chosen as the chair of TC-2. Auerbach (1986a, p. 81), first president of IFIP, records that it was Zemanek's unique communication skills that made him the best candidate:

> Finding a Chairman for TC-2 Programming Languages, proved to be a formidable task. There were very deep feelings among the various groups within Europe about ALGOL's origin and maintenance. To minimize conflict, a Chairman was needed who was not involved as one of the thirteen originators of the language, but who had intimate knowledge of the current evolution of the ALGOL language itself. Fortunately, Dr. Heinz Zemanek was such a man. He was a master diplomat, and for seven years he managed to stimulate significant output from the Technical Committee and Working Groups.

IBM allowed Zemanek plenty of freedom to pursue his duties in IFIP, and even provided generous financial support. When he was elected president of the organisation in 1974 (Zemanek 1986a, p. S3), he was able to hire—as IBM employees—both a secretary and assistant purely for IFIP purposes (Zemanek and Aspray 1987, p. 61). The assistants he chose were first a lawyer and then an economist: not trivial expenses. Zemanek managed to navigate a path carefully between loyalty to both IBM and IFIP; while he occasionally had to be careful what he said in interviews, he felt that IBM realised that not only was the international collaboration that IFIP provided good for IBM, it also was part of their duty to the professional community (Zemanek and Aspray 1987, p. 60).

[12] Translation by the present author.

IBM had also been supportive of the *Formal Language Description Languages* conference, so helpful to the language description community as a whole and Zemanek's team in particular, contributing financially to its hosting[13] (Utman 1964b). This conference was a major turning point in the direction of the Vienna group: Zemanek described it as "the key tool to entering the field" (Zemanek and Aspray 1987, p. 44). Prior to the event, there was no real experience of language description amongst the team, and it was a wonderful opportunity for them to learn about the state of the art. Members of the team, in later recollection, believed Zemanek to have been the driving force behind the conference—which would certainly have been how he presented it internally.[14]

Shortly after FLDL, the Vienna Science Group received a welcome upgrade, moving to a Laboratory in its own right and no longer needing to report to Böblingen. The IBM press briefing at the time gives the date of this shift as 6th December 1964 (IBM Informationsabteilung 1964). However, Zemanek was not keen to commit to a date himself, saying "the date is not a 'constant of nature'. We had arguments and there are different answers. But around 1964 it became certainly an independent European laboratory of IBM" (Zemanek and Aspray 1987, p. 43). Jones, who worked at the Lab from 1968, recalls that on some occasions, Zemanek had given the date as 1st January 1965—coincidentally his own birthday.[15]

This time period was also when the group embarked on the large task that would dominate their work until 1970: the formal description of PL/I. Because of their experience implementing ALGOL 60 for Mailüfterl, Zemanek's group was seen as IBM's language expert. They had also learnt FORTRAN, and when NPL (as PL/I was initially called) was first being proposed in Spring 1964, the group wrote an extensive critique of the new language (Zemanek and Aspray 1987, p. 43). Before getting deeply into the Vienna work on PL/I, it is worth reviewing a little of the history of the language's genesis and its characteristics.

5.3 PL/I

In the early 1960s, IBM computers were aimed at particular problem domains: scientific, commercial, and special-purpose computing. However, the needs of users were

[13] See Section 4.2.

[14] Neuhold (2016) and personal communication with Maurer, 2016.

[15] Jones, personal communication, 2016.

somewhat converging; scientific users, for example, also wanted the neat reports that business users demanded. IBM decided the best way to address this situation was to produce a range of computers intended to meet the needs of all users. The computers in the System/360 range (the name was chosen to indicate the family's full-circle coverage) came in different sizes with different hardware power, but were supposed to be inter-compatible (Radin 1981, p. 552).[16] This was designed to decrease the cost to customers of migrating between machines, especially to more powerful ones: a higher-tier System/360 model was supposed to be able to run exactly the same programs a lower-tier machine could. This was in contrast to earlier computers, when the instruction sets differed so greatly across IBM's range that most programs would have to be recoded if a new machine was purchased. David Beech, an employee at IBM Hursley from 1963, remembered that the early days of the range's development were shrouded in secrecy: employees were not allowed to mention the new product line by name and instead had to use the names of various competitor machines to fool any spies into thinking they were talking about development for those rather than new IBM machines (Beech 2016).

To accompany this new range of machines, IBM and their SHARE usergroup formed an Advanced Language Development Committee to develop a new programming language that was also sufficiently broad (Radin 1981, p. 552). It was to be called 'New Programming Language', or NPL; this nicely coincided with 'New Product Line', an early name for System/360. The language was renamed in 1965 to PL/I, for Programming Language One,[17] due to complaints from the National Physical Laboratory in Teddington, U.K., who claimed they had used the acronym first (Radin 1981, p. 556).

The Advanced Language Development Committee was composed of three IBM employees and three members of SHARE, and began work in October 1963. The initial deadline for completion was very tight: December of the same year, in order to allow the launch to coincide with that of System/360 in January 1964 (Radin 1981, p. 553). The group was primarily drawn from FORTRAN users, and, indeed, for the first few meetings the language was called FORTRAN VI, but the idea of making the new language a backwards-compatible extension to FORTRAN was dropped as

[16] For more on the history of System/360 and its importance, see Campbell-Kelly and Aspray (1996, Ch. 6).

[17] IBM trademarked 'PL/x' for a number of x above 1 (Peláez Valdez 1988, p. 92), but did not think to trademark 'PL/0'. Niklaus Wirth (1976) was therefore able to use the name 'PL/0' for the example miniature language in his book *Algorithms + Data Structures = Programs*—although this was a subset of Wirth's PASCAL language and in fact bore no relation to PL/I.

doing this would place more restrictions on the language than would outweigh the benefits of compatibility (Radin 1981, p. 555). As the project progressed, deadlines slipped, but the continued sense of urgency may well have contributed to the mixed success of the end product.

As described by Fred Brooks, Turing Award winning architect of the System/360, PL/I was intended to be "a universal programming language that would meld and displace FORTRAN and COBOL" (Brooks and Shustek 2015, p. 40). Some in IBM believed this would free them from the need to maintain compilers for these two languages on their machines (Astarte and Jones 2018, p. 87). Walk (2002, p. 81) described PL/I as "the most complex language seen so far", and noted that this was because it was intended to cover both business and numerical applications as well as system capabilities such as input and output, and file control. Beech (2016) added that PL/I was wryly called "a FORTRAN, COBOL, and ALGOL combination as understood by FORTRAN programmers".

A unique aspect of PL/I was that it was intended to be easy to use for a novice programmer: the user should not need to know every single feature of the language in order to use it. This meant many of the language concepts had default options if not explicitly supplied. Another angle was that the syntax was supposed to be as pragmatic and similar to natural language as possible: the programmer was even referred to as the "speaker" in some documents (Priestley 2011, p. 237). This shows that the language metaphor, as highlighted by Nofre, Priestley, and Alberts (2014), was a core design principle. Also linked to the idea of loose syntax was a desire to move away from the rigid structure of FORTRAN code presentation, such as the identification of labels being determined by their position on the program cards (Radin 1981, p. 581).

Another important language design aspect was "*Anything goes.*[18] If a particular combination of symbols has a reasonably sensible meaning, that meaning will be made official" (Radin and Rogoway 1965, cited in Priestley 2011, p. 237). This, combined with the looseness of the syntax, and the large range of default options, meant that PL/I compilers would have to be very sophisticated indeed. Such loose definitions would prove awkward for language implementers and definers alike. Indeed the design criteria for PL/I are noteworthy for some of the concepts they *lack*: no mention is made of trying to provide a solid theoretical base, ease of creating well-

[18] Original emphasis.

structured programs, or the possibility of proving correctness (Radin 1981, p. 588). This is in stark contrast with the design criteria put forward by Hoare (1973) in his 'Hints on Programming Language Design', a paper in which he despaired at what he saw as the abysmal situation programming languages had reached. At the 1963 *Working Conference on Mechanical Language Structures*, Alan Perlis (in Gorn 1964) made a sarcastic comment in which he wished ALGOL had been more complicated in order to give the people who wrote about grammars and semantics something to get really stuck into. PL/I certainly granted this wish.

The first draft of NPL was presented at a SHARE in March 1964. George Radin (1981, p. 556), one of the original committee of six, recounted how the language provoked strong feelings amongst the attendees:

> The reaction was anything but indifferent. [NPL] was widely praised for its breadth, its innovations, and its attention to programmers of varying sophistication. It was criticized for its formlessness, complexity, and redundancy. One SHARE member later compared it to a 100-blade Swiss knife; another wondered why the kitchen sink had been omitted from the language.

Shortly afterwards, the Vienna Science Group produced an intensive fifty-two page criticism of the language description document, pointing out errors and making suggestions for the language's design (Bandat et al. 1964). This shows their interest and involvement in PL/I almost from the very beginning (Radin 1981, p. 558).

In May 1964, members of IFIP TC-2 became concerned that IBM's attempts to create a universal language would clash with IFIP's interests. Communication between IFIP and the NPL development committee was desired, and TC-2 resolved to write a letter to the language development board expressing their desire for co-operation. Van der Poel even thought there could be a linking of their ideas: "Prof v.d. Poel felt that IFIP should, as soon as possible, describe NPL in the new ALGOL language of WG 2.1" (Utman 1964b, p. 13). The closed-door attitude of IBM towards to its new products was also a concern for TC-2. Steel thought that IBM was not likely to be particular bothered by what IFIP thought, and, in any case, there was obvious ALGOL influence in NPL (Utman 1964b, pp. 12–4).

A second version of the NPL report was produced in June 1964 and IBM management thought it good enough to proceed with development. Control was transferred to IBM Hursley to start implementing a compiler (Radin 1981, p. 557). At this

point, language development was now solely in the hands of IBM, with SHARE no longer involved. The laboratory at Hursley was located in Hursley House, a historic manor outside of Winchester in the UK in which Oliver Cromwell's son had lived in the 17th Century, and where the iconic Spitfire aeroplane was designed during the Second World War (Endres 2013, p. 6). The Hursley team produced in November of 1964 a two hundred page document describing PL/I which became the definition on which they would base their compilers (IBM 1964). This was also the basis for the 'Systems Reference Library' (SRL) document, 'Form C28-6571-x' (the final number changed with versions), produced by Language Control at the New York Programming Centre: the official IBM standard for PL/I until the ANSI standard in 1976. The creation of the Hursley report was a huge job, and ended up consuming every single one of the laboratory's programmers, as noted by Jim Cox (in Radin 1981, p. 595). Radin (1981, p. 572) praised these efforts of IBM Hursley: they had managed to transform the Language Development Committee's output into a well-defined and usable programming language.

One noteworthy change made by Hursley was the inclusion of ALGOL-style BEGIN and END block delimiters. This was made at the request of the Vienna Group who generally wanted the language to be more like or even based on ALGOL (Radin 1981, p. 593). This may have been due to their familiarity with ALGOL thanks to implementing the compiler for Mailüfterl. The Vienna Group also argued for the removal of the ALLOCATE system which allowed dynamic storage to be allocated at other points than just block entry. However, Radin and Rogoway objected to this removal—and so both were kept (Radin 1981, p. 557). This was characteristic of the design decisions in PL/I: every language feature should be present. Illustrating this, Radin (1981, pp. 585–6) quoted some "typical dialogue" from the presentation of the language to the "European University Computer Leaders" which took place in June 1964 at Hursley:

> Notice, by the way, this typical dialogue:
>
> "What about this language?"
> "Well, it's awfully big and complex."
> "Okay. What shall we do about it?"
> "Well, you should add the following..."
>
> It really did happen all the time.

There are two other important parts of PL/I which are worth mentioning. The first

is the richness of data types, any combination of which was permitted: for example, arrays of structures and structures with array components. Modifiers for types could be declared, such as fixed or free length, and all these many attributes had also to have default values to fit with the criteria that the programmer need not know all language features in order to use it. The second crucial feature is the notion of 'tasking', which Radin (1981, p. 578) described as "processes which can execute asynchronously and which can synchronize on events". This concept of parallelism would be the cause of a great deal of complexity in the Vienna description.

Radin (1981, p. 572) summed up PL/I in his historical piece:

> NPL was a language defined too quickly. Its virtues resulted primarily from the combined intuitions (and programming experience) of its definers. (They convened "knowing" what they wanted in the language.) Its faults were largely ambiguity and incompleteness.

Radin argued that NPL took good aspects from both FORTRAN and COBOL, and its strongest feature was its breadth and detail. This allowed the writing of both application and systems programs entirely in PL/I, meaning that a programmer need have no knowledge of system functions or machine code (Radin 1981, p. 573). Albert Endres (2013), however, argued that this aim for full coverage was doomed to failure: business and scientific users always had different needs and would have preferred specialised languages.

Reaction from the academic programming community was generally negative, with Hoare joking about PL/I's size:

> Tony Hoare was the chairman of the European PL/I standards committee, and at a meeting in Vienna, we were discussing whether we could define a subset in the "spirit" of PL/I, and Tony Hoare said, "No, it's impossible. The spirit of PL/I is to be a *big* language!"
> (Wexelblat, in Radin 1981, p. 596)

Dijkstra also commented on the size of PL/I and the way it had expanded throughout its development in a vicious diatribe in his Turing Award acceptance speech (Dijkstra 1972, p. 862):

> Finally, although the subject is not a pleasant one, I must mention PL/I, a programming language for which the defining documentation is of a

> frightening size and complexity. Using PL/I must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control. And if I have to describe the influence PL/I can have on its users, the closest metaphor that comes to my mind is that of a drug. I remember from a symposium on higher level programming languages a lecture given in defense of PL/I by a man who described himself as one of its devoted users. But within a one-hour lecture in praise of PL/I he managed to ask for the addition of about 50 new "features," little supposing that the main source of his problems could very well be that it contained already far too many "features." The speaker displayed all the depressing symptoms of addiction, reduced as he was to the state of mental stagnation in which he could only ask for more, more, more...

In part due to the vastness and complexity of PL/I, and the consequent difficulty of implementing a compiler for it, IBM began to explore the possibility of formal definition. In 1965, Tom Steel reported to IFIP TC-2 that formal descriptions were being developed in a number of places: at Hursley; at IBM Poughkeepsie (using a notation developed by Ken Iverson); by the ACM Special Interest Group for Programming Languages (with a syntax-directed compiler approach); and by the American Standards Agency (Steel, in Teufelhart 1966, p. 10). However, the most fully developed definitions were produced by the IBM Laboratory Vienna, which are described in the remainder of this chapter.

5.4 Towards a formal definition

As soon as the first document on NPL reached the IBM Vienna Science Group in March 1964, Zemanek went to management to argue that the language needed formal definition. In part this was because of System/360: while in theory there would be compatibility and equal performance across the range, in reality the machines differed quite significantly because of their hardware capabilities. A formal definition of NPL would provide much-needed standardisation and reference for implementers of compilers across System/360 (Zemanek and Aspray 1987, pp. 43–4). In fact, IBM had been considering the use of formal definition methods already: an internal memo dated June 1964 showed the results of a survey of the field, and recognised the importance of formalisation used *before* language design.

Zemanek himself had been pondering how to go about approaching the task of formal definition, and eventually published the results of these speculations in a piece entitled 'Semiotics of programming languages', first as an IBM technical report in August 1965 and then in *CACM* in March 1966[19]. Characteristically, Zemanek focused on the work of the *Wiener Kreis*, a group of philosophers based in Vienna in the 1920s who looked at logical empiricism and its application to language. Formalism for natural languages, argued Zemanek, seemed less immediately useful, but the tools became of great use when applied to programming languages. Zemanek explained that Russell described some principles which were useful when considering programming languages:

> Russell once gave four reasons for formalization which still apply to both theory and practice: (1) security of operation is assured, (2) tacit preassumptions are excluded, (3) notions are clarified, and (4) resolving structures can be applied to many other problems.
> (Zemanek 1966, p. 141)

This demonstrates how Zemanek saw the core motivations for language descriptions held by the Vienna Lab.

Walk (2002, p. 80) explained how the attitudes of the programming community at the time influenced the group, and that the experience of writing the ALGOL 60 compiler for Mailüfterl highlighted these problems:

> The languages were announced as being machine independent, but the manuals failed to say on what these languages now depended, if it was not the real machine. Understanding language manuals relied on obvious analogies with arithmetics or real machine constructs, and on manually following and interpreting the program text. The question began to be recognized in the early sixties. The community was in search for well founded language definitions to serve as input to compiler writing, to investigate the correctness of programs, and to improve the conceptual understanding and teaching of programming languages. The feeling for the lack of these foundations was building up in Vienna in the course of writing a compiler for the ALGOL60 language, completed in 1961.

A retrospective written by Lucas (1981, p. 549) a little after the main activity of the Vienna Lab was over, added another good reason: language reference and standardisation:

[19] Page references are made to the *CACM* publication.

> To remain stable both in form and meaning, a programming language needs a rigorously precise and implementation-independent definition. Such definitions do not emerge casually, as committees struggle to standardize a language. This has been painfully clear ever since FORTRAN and ALGOL 60 appeared on the scene.

According to the same source, the group began investigating formalising PL/I in late 1964 (Lucas 1981, p. 549). Bekič (1964) had already performed some preliminary work on language description immediately after *Formal Language Description Languages*, authoring a paper entitled 'Defining a language in its own terms'. This described the semantics of mechanical languages by reducing them to elementary terms, and initially focused on expression languages, particularly LISP—showing very early on that McCarthy was a big influence on the Vienna group. The fourth section of Bekič's paper addressed 'programming languages', which is to say those which contained statements, and included the important comment that "a statement can be interpreted as a function mapping states into states" (Bekič 1964, p. 24). The paper shows clearly the influence of various different FLDL presenters, especially van Wijngaarden, Landin, Strachey, and McCarthy.

Another direction was also explored: translation into a language whose semantics was already known. The idea was also presented in another paper by Bekič (1965b). Lucas (1987, p. 3) later explained that this idea was rejected because the new language still needed defining, and for PL/I, there was no obvious way to pick which the new language ought be as there was no core subset of PL/I that could be chosen.

In July 1965, four papers, edited by Bandat (1965) set out the basics of the approach that the Vienna Lab would eventually use. Lucas also gave a presentation on the same topic at the IBM internal Programming Symposium at Skytop, Pennsylvania, and he showed a preliminary version to IFIP WG 2.1 in May 1965. That report was called "Definition of Micro XXX by Finite Local Statevectors"[20] and according to Woodger (1965, p. 6), who wrote a report on the meeting, the approach was presented as an extension to McCarthy's work. The method in the paper could handle declarations, blocks, and parameters called by name, but was not yet sufficiently powerful to handle the entirety of PL/I. The reaction from the working group was rather cool; Woodger wrote "It seems to me that this work adds little if anything to what has already been published by Landin using very similar techniques".

[20]The 'XXX' was a result of deleting 'NPL' from the title; PL/I was at that time not yet settled as a name.

In the meantime, IBM Hursley had been working on the development of a PL/I compiler, and producing their own internal documentation on the language to aid in this goal. An early Hursley document from September 1965 shows that the initial plan was for Hursley to write a formal definition of PL/I (Larner and Nicholls 1965). The authors hoped that such a description would free the language from being tied to any particular compiler, would ease the process of language extension, and would help with the compatibility of implementations—a key goal for System/360. The initial plan was to finish the definition by the first quarter of 1966, before the delivery of their PL/I compiler; they hoped, somewhat ambitiously, that two or three people working on the task from September 1965 would suffice—despite recognising a significant risk that staff from the project would likely be poached for fighting crises emerging in the compiler project. One final interesting note about this document is that it cited the Bandat-edited collection of papers (1965) from Vienna, showing that Hursley was up to date with the technical work going on there.

In October 1965, a meeting took place between workers from Vienna and Hursley to further discuss the creation of an 'unambiguous' definition of PL/I. The aim of this would be to aid compiler writers, product definers, language extenders, and language users (de Vere Roberts 1965). The meeting invitation shows that there was a clear desire to involve the Vienna group in the language definition process, but it was not yet obvious to whom the full responsibility would fall. The minutes of the meeting (Bandat et al. 1965) show that it was only when Hursley and Vienna got together to discuss the matter that this was worked out. The term 'Universal Language Document' was coined here to refer to various descriptions of PL/I and there were to be three levels. ULD-I would refer to the SRL, the natural language PL/I standard already in existence; ULD-II would be a short, semi-formal description produced by Hursley; and ULD-III would be a fully formal document written by Vienna, with input from Hursley.[21] The ideas moving between Hursley and Vienna are an example of the way that IBM management would shuffle work about between their different branches, as mentioned earlier in the current chapter.

Whether this communication between Vienna and Hursley was competitive is uncertain. Beech (2016) reported that Hursley was happy to let Vienna do the full

[21]Note that a lot of confusion remains over the names. No document was officially entitled 'ULD-I', although the SRL was referred to as this by Vienna and Hursley. Hursley's description did actually use the name 'ULD-II', but the Vienna descriptions often omitted the 'III' and called the three versions of the description they produced 'ULD 1', 'ULD 2', and 'ULD 3'. There was no consistent use of either Roman or Arabic numerals.

The present work will refer to documents by the names they are given in their titles.

formalism, but Neuhold (2016) had the impression that Hursley wanted to do the whole thing "whatever Beech said". Walk recalls that most conflicts between the two Laboratories were over the precise meaning of certain language constructs.[22] Jones, who was at Hursley first and then transferred to Vienna in 1968 remembers a particular debate, which took place at a meeting at IBM Nice. An executive from language control in New York was flown in to arbitrate, and said "Hursley wins on one point and Vienna on two; take your pick for which."[23]

Hursley's ULD-II was intended to cover the following four points, according to minutes of the October meeting (Bandat et al. 1965):

1. Exposition based on fundamental language concepts, emphasising structures
2. Syntax separate and formal
3. Semantics independent of syntax
4. Semantics to be based on an abstract machine

The plan was for ULD-II to be delivered by Hursley in the first quarter of 1966, and ULD-III by Vienna in the final quarter of 1966. ULD-III would be revised from ULD-II to include any changes to PL/I in the intervening period, and the two should be consistent in their use of concepts and terminology. ULD-II and -III personnel would not be members of language control but should still have participated in language development. Ted Codd, later of relational database fame, was the IBM Poughkeepsie representative and would help to isolate the fundamental concepts of PL/I and review ULD-II and -III as they were developed (Bandat et al. 1965).

The Hursley effort was led by David Beech, a Cambridge trained mathematician. He explained later that their aim was to create a description which was precise but still readable by their compiler developers. Beech saw the goal as being a description which was sufficiently legible that it could serve as a daily reference. The publication was in four parts over the end of 1966, although the dates on the final documents were fairly arbitrary; Beech (2016) explained that everyone in the team had access to draft versions as they were developed. The four parts were: concrete syntax (Beech et al. 1966), abstract syntax (Beech et al. 1967), a translator from concrete to abstract syntax (Beech, Nicholls, and Rowe 1966), and an abstract interpreter (Allen et al. 1966).

[22] Personal communications, 2016.
[23] Personal communications, 2016.

The concrete syntax was written in a grammar similar to BNF with some extensions to cope with the notational complexity of PL/I. The abstract syntax was given in two-dimensional charts of trees showing the structure of the language, and the translator was a parser function that turned concrete programs into their abstract structure.

The most important document was the interpreter, which was based on ideas developed from the compiler implementation, with abstraction used to remove machine-dependent features. The core state of the interpreter is an abstract store for data, with some data sets, and an interpreter processing unit with operations over the two. It allows sequential interpretation only, and the parallelism of PL/I tasks is handled with an informal description, something allowed by the lack of a completely mathematical interpretation function. The semantics showed the changes that occur in the state during interpretation. Full formalisation was not attained or even intended: diagrams and prose reign throughout. However, there was some use of carefully structured prose to describe state transitions; Beech (2016) stated that his goal was to make a sufficiently formal description of the state that "mathematician's English" would suffice for the transitions.

An interesting passage from the interpreter document (Allen et al. 1966, p. 22) explains this goal:

> It is typical of attempts at precise definition that they may seem to express some simple intuitions in elaborately artificial ways. One may have an enquiry to make of this semantic definition of PL/I, and the writers of the definition may have been conscious that the interpreter must proceed in such a way as to give a certain answer to that question - yet the answer does not show on the surface. The definer has been put to the labour of constructing a means of giving the desired result, and the enquirer must then prise out the original idea from this disguise.

The Hursley team's hope was that the names given to the intermediate interpretation functions would be sufficiently clear that a reader would not have to chase to the bottom of the interpretation chain to work out the result. Such a result would not be completely precise, but would be clear enough to suit the day-to-day user. This served the ULD-II aim of not being completely formal—that is, able to provide a complete answer—but would enable a language worker to figure out the meaning.

The introduction to the interpreter summed this up: "The main aim of the method presented here has been to produce a semantic definition of PL/I which is rigorous

yet readable" (Allen et al. 1966, p. 4). The description provided very nearly complete coverage of the language and was less than half the length of ULD-III, at (only) 155 pages. Beech remarked that he much preferred the narrative style of ULD-II than pages and pages of formulae, likening ULD-II to *Principles of Mathematics* (Russell 1903), where Vienna's ULD-III was more like the densely equation-packed *Principia Mathematica* (Russell and Whitehead 1912).

There was no external publication of ULD-II, but Beech used some concepts from the work in a piece in ACM's *Computing Surveys* that explained PL/I (Beech 1970). This paper, incidentally, provides another indication of the size of PL/I: in the very first sentence, Beech apologises for the complexity of the language, which is a noteworthy admission for someone so invested in its success.

ULD-II was not ultimately ready quickly enough to be a starting point for Vienna as had been initially planned, so the two laboratories ended up working somewhat in parallel. Indeed, an acknowledgement in ULD-II mentions the co-operation and advice of the Vienna group (Beech, Nicholls, and Rowe 1966, p. 1). It is interesting to wonder why work on ULD-II was continued once ULD-III was well under way; Beech (2016) explained it was because there were very different users in mind for the two definitions, and the thought was that ULD-II would be more usable.

As Vienna began working on their own definition, they understood there would be inconsistencies and ambiguities in the SRL standard of PL/I from which they were working, as it was written out in natural language. So, when questions arose from the Vienna group, they would ask Hursley for clarification and model whatever they said. The Lab was clear it would take no responsibility for the content of PL/I (Zemanek and Aspray 1987, p. 44). These conversations between Vienna and Hursley were recorded in a series of numbered notes exchanged between the two locations. Those from Hursley were called LDH-x (for Language Definition Hursley) and those from Vienna LDV-x. The series ran between October 1965 through to summer 1966, and followed a general trend of Vienna asking questions and requesting clarification. In addition to written communication, there were many visits in person between the two groups; Beech (2016) recalls eight trips to Vienna in one year. Walk (1967b) wrote that the communication was not as smooth as he would have liked; for example, it took sixteen memos to language control to determine what they wanted for input/output. According to Beech (2016), this meant Vienna played an important role as error checkers: their formalism showed up mistakes and inconsistencies in PL/I.

One interesting part of the LDV/LDH discussion was a proposition from Maurer who suggested using rewriting rules to simplify PL/I programs. This resulted in procedures with only one entry point, no blocks, and the only conditional statements being of the form 'IF expression THEN GO TO label' (Maurer 1966b). This was also published as a Vienna technical report (Maurer 1966a), but the ideas were ultimately not used in ULD-III. The approach bears some similarity to the ideas of van Wijngaarden, whose talk at FLDL Maurer would certainly have attended as he was the scientific secretary for this session. However, Maurer's papers did not directly cite van Wijngaarden's work.

As the communication between Vienna and Hursley went on, the technical depth of the questions and answers increased. Vienna began to use formalism in an attempt to make precise questions and pin down the answers from Hursley and Language Control. Ultimately, the notation used in this communication became the method Vienna used to write their full description of PL/I, ULD-III. The following section takes a look at that work.

5.5 The Universal Language Definitions

The first version of the Vienna group's full formal definition of PL/I was published as a Lab technical report in December 1966—on time by the schedule agreed the previous year—under the name 'Formal Definition of PL/I (Universal Language Document No. 3)' (ULD-III 1966).[24] It is a very large document, running to over three hundred pages, and covers every aspect of the definition (unlike later versions, which would split up the different parts into different reports). The author on the front of the report is given as 'PL/I – Definition Group of the Vienna Laboratory', and the full list of authors is given inside. These constitute the majority of workers present at the Lab at the time: Kurt Walk, Peter Lucas, Hans Bekič, Kurt Bandat, Gerhard Chroust, Paul Oliva, Viktor Kudielka, and Klaus Alber. The introduction notes that the method of approach and basic notions are largely the work of Lucas and Walk; however, according to Neuhold (2016), knowledge was shared frequently within the Lab with talks and presentations and nearly everyone had a decent understanding of the approach as a whole. The specific details of particular sections were likely known only by their authors, and by Lucas and Walk, who had the best knowledge of the work as a whole.

[24] Citations to this document in the present work use this deliberately shortened label.

A remark in the preface warns that the document "contains essentially the formal description as such, with a minimum of reading aids" (ULD-III 1966, p. iii), although a later note adds "It may be expected that especially a programmer will not find it difficult to penetrate the formal building" (ULD-III 1966, p. 1-2).[25] As Jones noted, although the conversational language of the Lab was German, anything written down was almost exclusively in English.[26] The presence of a few German-language technical report publications indicates this was not a blanket IBM rule, but the company would likely have looked askance had the majority of the group's output been unintelligible to the rest of the company. In any case, Zemanek required his staff to be confident in English, the better to fit with his international collaboration aims.

The central idea to the PL/I definition method was described in slightly different ways by the three main writers of the Lab.[27] Zemanek's explanation relied on the notion of a reader pulling meaning from the description: "The idea is an abstract machine of the language you define. The meaning of the program is defined by the behavior of this machine. [...] The meaning appears to the human mind by looking at what happens when a text is applied to the structure of the machine" (Zemanek and Aspray 1987, pp. 45–6).

Walk's (2002, p. 81) description is similar, but a little more technical:

> The program to be interpreted defined the initial state of the abstract machine, and the successive application of the transition function defined the computation steps until, hopefully, an end state is reached. The behavior of the abstract machine and, in particular, the end state defined the 'meaning' of the program.

Lucas (1981, p. 551) described the most important idea being the notion of state transition:

> Central to main line programming languages, including PL/I, is a category of imperative sentences ("command" and "executable statement" are synonyms). How can the meaning of an imperative sentence be explained? Intuitively, the intent of such sentences is to effect change.

[25] Pages in large Vienna technical reports are numbered by section.

[26] Personal communication, 2018.

[27] While Hans Bekič was one of the main theoretical contributors, he did not produce as much published material—and his life was tragically cut short in a hiking accident in October 1982 and thus he was unable to write the kind of retrospective piece cited here. Indeed, much of Bekič's important work is fairly unknown; the interested reader is directed towards *Programming Languages and Their Definition: Selected Papers of H. Bekič* (Bekič and Jones 1984).

> The command "Paint this wall green," faithfully executed by an obedient painter, will turn a possibly white wall into a green one.
>
> The most direct explicate of the meaning of an imperative sentence is a function over a set of potential states, mapping given "current" states into successor states, thus indicating the change of state intended to occur when the command is executed. Referring to the quoted example sentence above, the universe of states would be the relations between some set of walls (that can be referred to or pointed to) and a set of colors and patterns that walls can be given.

The same basic idea is recognisable in these three quotations, but for each writer there are subtle differences. Zemanek was always keen on bringing the human mind into notions of meaning (as seen in his 1965 paper on semiotics); Walk had the most machine-related view, as befitting his role as the hardware physicist on Mailüfterl; and Lucas the most abstract.

The ULD-III description method was based on three important notions: an abstract structural view of the syntax; an abstract machine of many parts, whose state represents the state of computation; and transition function mapping machine states into machine states. The concepts of abstract syntax, state, and state-to-state transition functions are due to McCarthy; the machine of many parts comes from Landin's work; and the notion of mathematically modelling machine states, and functions mapping those states also was put forward by Elgot in his RASP work. All three are cited in later work by the Vienna Lab as influences (e.g. Lucas 1981, p. 551). The introduction to ULD-III also contains acknowledgements to these researchers, noting that the ideas of instruction and computation are particular due to Elgot[28] (ULD-III 1966, p. 1-3).

An interesting side note on the citation of McCarthy's work is that his FLDL paper is cited with the name given in the program—'Syntax and the Formal Description of the Semantics of Programming Languages'—rather than that of the proceedings—'A formal description of a subset of ALGOL'. This may be because the proceedings' publication had not been finalised at the time the Viennese wrote that portion of ULD-III, or could imply that the group had access to preliminary version of McCarthy's work. In fact, McCarthy had been to the Vienna Laboratory in December 1965 to instruct them on his definition method, so there was a close connection between them.

[28] It is also worth mentioning that Elgot visited the Lab in 1966, so his ideas may have been more prominently in the group's mind at that time. Elgot was not working on language description in Vienna, but rather more abstract notions of algorithms (see for example Elgot 1966a).

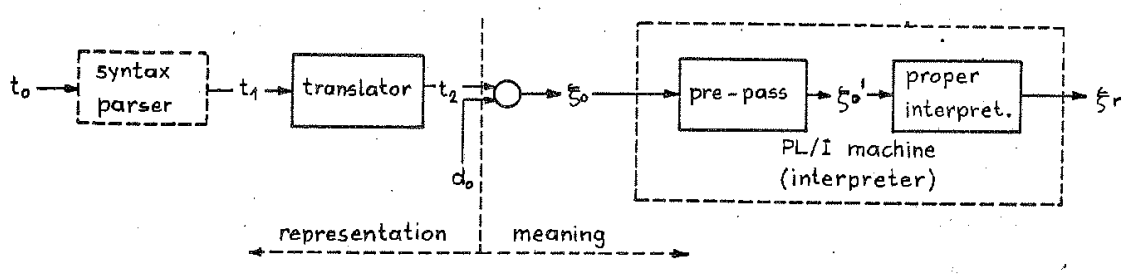


Figure 5.5: A diagram showing how PL/I programs are interpreted in the ULD style. Taken from the 'Informal Introduction' to the method (Alber et al. 1969).

However, as McCarthy explained in a letter to Ershov, he did not approve of the confidentiality of ongoing research: "They have written two reports, the latter of which is 'IBM Confidential.'[29] This is very annoying to me and I am complaining since there is no commercial reason for it, and my work is heavily involved" (McCarthy 1965).

One problem faced by the Vienna Lab in writing their definition was avoiding copy and typing errors—no easy task given the huge size of PL/I. To cope with this, they developed an editing system that knew the grammar of the metalanguage and could spot errors in it, as well as automatically producing an index of the definition locations of terms and every place in which they were used (Zemanek and Aspray 1987, p. 47). This system, somewhat akin to the development environments used by today's programmers, was presented in a separate report (Schwarzenberger and Zemanek 1966).

ULD-III was composed of four main parts: a description of the concrete syntax of PL/I; a description of the abstract syntax; a translator from concrete into abstract syntax; a description of the abstract machine states, and the transitions of that machine (called the interpreter). The high-level flow of the interpretation process is shown in Figure 5.5.[30]

The concrete syntax and translator were described in a separate report by Alber (1967). This used BNF to define the legal strings of PL/I programs, and a text-replacing function to describe the translator, which converted from concrete programs into an abstract notation. A lot of influence from McCarthy can be seen in the abstract syntax, with the concept of language components being built up from

[29]Note, however, that all finished ULD Reports were published without the 'Confidential' restriction.

[30]This comes from the third version of ULD-III, but there was no change to this process between versions.

small parts which could be accessed with selector functions. Lucas (1978, p. 8) explained the reason for using an abstract syntax: "It is, at least for complicated languages, much more concise than the concrete syntax. Thus, a semantic definition on the basis of an abstract syntax becomes independent of notational details and is also more concise". A guide to the PL/I standard, which took a similarly abstract approach to syntax, noted: "There is much in PL/I that is intended to provide flexibility, abbreviation or convenience for the user which needs to be put into some standard form first before beginning to interpret. That is the function of the translator" (Beech and Marcotty 1973, p. 29).

The Vienna definition's notation was based on a notion of objects: every concept in the metalanguage is an object. A guide to the object notion was written by Ollongren (1971) which explains the properties and consequences of these objects; this came considerably later than the main definition work, but the object concepts were being used consistently for many years until that point. Objects are either 'elementary', like real numbers or Booleans, or Ω for the 'empty object'; or 'composite', comprised of selectors yielding other objects. There are some notational differences in the Vienna style: rather than having a separate constructor for every object as McCarthy had, there is a universal μ_0 operator to create objects. Also, where McCarthy was explicit about the selector functions for every construct, the ULD-III notation implicitly allows any composite part to be selected by name. The advantage of this rather heavy weather which is made of objects meant there could be universality: any function could be applied to any object. However, it did make for some unnecessary notational complexity in the treatment of frequently-occurring data structures such as lists and maps. Later notation developed at the Lab, known as META-IV, addressed this problem by using a richer set of primitives.[31]

Error checking—for example, for application of a function outside of its domain—is performed dynamically in the semantics. Errors are produced by distinguished cases in the interpretation functions, particularly default cases, and raise the error condition present in PL/I.

The essential idea of the semantics also takes strong inspiration from McCarthy. The language is modelled by imagining it being interpreted by an abstract machine, on which Walk (2002, p. 81) wrote "We constructed an abstract machine which was characterized, like a real machine, by the set of states it can assume, and by the

[31] More on this development is discussed in Section 7.3.

transitions from one state to the next". Walk also ascribed the core influence for taking this direction to be the work of McCarthy. The states of the interpreting machine in the ULD-III method are called Ξ and the interpretation of most statements causes some change to this state; this is the core idea of most model-based semantic approaches. Lucas (1981, p. 554) noted that the larger set of language constructs in PL/I than in the languages McCarthy had addressed required adding extra parts to the state. For example, allowing value sharing across blocks required an 'environment' component.

This was just one of the many components in Ξ, whose multi-part nature was influenced by Landin's SECD machine. These components are:
E environment
D dump
DN denotation directory
UN unique name counter
C control
CI control information.
By 1969, the term 'grand state' was being used for this style of state with many components (Walk 1969b, p. 33). This was usually employed pejoratively: the complexity of determining which state components to adjust for each statement proves challenging. Each state component will be discussed here in turn, but not in great technical depth.[32]

The environment E links local program identifiers with globally-unique names. By performing a lookup in the environment, a unique reference to any identifier anywhere in the program can be determined. The environment is modified at phrase entry to ensure that it contains only references to variables which are declared within that particular scope. Only one environment can be active at a time and this allows the phrase structure of PL/I (and other languages with blocks and procedures) to be described.

A dump component D is a stack of environments for phrases which have been entered but not yet terminated. Entering a phrase pushes the current environment onto the dump and creates a new one; the final step of interpreting the phrase is to take the top element of the dump and make it current. Looking backwards in the dump allows access to variables declared in enclosing scopes. Note that special

[32]For a simplified exposition of the approach, see Neuhold (1971); for an exhaustive description see Lucas and Walk (1969).

actions need to be taken to manipulate the dump when a phrase is left abnormally (such as with a jump).

DN, the denotation directory, links globally-unique names with their denotations. For variables, the denotation is the variable's value, along with its typing and modifier information; for procedures, it is the text of the declaration. The globally-unique names are the same as those in the environment, which enables the sharing of values across contexts. A variable can be looked up in the appropriate environment to determine its unique name, the denotation of which can then be determined from the DN. Multiple environments can have identifiers pointing to the same unique global name, which has only one entry in the denotation directory, so modifications made in one context can be seen in others. No values need ever be removed from the DN; old values which are no longer referred to in either the current environment or any in the dump simply become inaccessible.

The unique name counter UN is very simple: it is an integer which increases every time a new identifier is detected, and provides a unique name for each. This unique name is used within the DN.

The control part C is a tree which represents the flow of program control. Using a tree allows modelling the parallelism in PL/I tasks, as well as non-deterministic choices like order of expression evaluation. Branches of the tree indicate possibly parallelisable actions: at any point in time the leaves of the tree are (equally valid) candidates for the next interpretation. A fine-grained splitting up of actions allows interleaving at a very low level: half of a procedure call could be executed and another leaf chosen from another part of the tree before control eventually returns to the procedure.

Finally, the control information CI has three parts: the text of the entire program; an index pointing to the part next to be executed, and a control dump for keeping track of nested calls to procedures. The first two parts are similar to the approach taken by McCarthy in his Micro-ALGOL description, but extended to cope with the larger language under definition. This retention of program text is necessary in order to handle jumps backward. C and CI tend to model similar information about the program's progress, but where C is more abstract, CI has a much more obvious correspondence with the program's structure as written.

The control and control information parts together act in a way analogous to Landin's stack and control; the dump and environment components are more obviously similar to the SECD components with the same names. The influence of both

McCarthy and Landin are clear to see in the ULD-III method.

Interpretation functions either modify state objects directly, or call on further interpretation or evaluation functions. A typical program interpretation starts with *int-program* and traces down through *int-statement* and into evaluation. The path this follows is based on the structure of the abstract program, as Lucas (1981, p. 556) explains:

> Given a composite grammatical construct, e.g., a conditional statement or iteration statement, one would like to compose the state transition of the compound from the state transitions associated with the parts. A statement, primitive or composite, could then be said to denote a function, viz., the state transition function to be applied to the state when the statement is executed.[33]

The non-determinism in PL/I means that the interpretation functions have to be able to produce multiple possible outcomes (all of which are equally correct). This can be seen in the type signature for *int-program*:

int-program : $AP \times \Xi \rightarrow \Xi$**-set**

where AP refers to 'abstract program'. The ultimate product of a PL/I program interpretation in the ULD style is then a set of states, each of which shows how the program could affect the state of the machine. The results of all interpretation functions are states, state components, or basic types; there are no side effects. Reynolds (1972) makes an interesting point: this makes the metalanguage purely applicative, like in most other approaches.

It is worth expanding on the handling of jumps in ULD-III which is awkward, and requires careful manipulation of the C and CI objects. This, along with the parallelism, is the cause of a lot of the complexity in ULD-III, and also the necessity of placing a whole stack of environments in the state. Conceptually, the jump interpretation is similar to McCarthy's, complicated significantly by the phrase structure of PL/I. There are four steps:

1. Close environments in the dump until the environment containing the identifier for the destination label is found;
2. Close control dump elements until the labelled statement is found;
3. Advance the control information index pointer to the labelled statement;

[33] This bears a strong similarity to the 'compositionality' principle of the later denotational semantics; see Section 6.4.

4. Update control information with the index and resume sequential interpretation.

Higher-order functions and procedures—those that can take functions as parameters or produce them as results—are handled easily in the ULD-III method. Denotations of procedures are simply the text of the procedure and its environment, so it is simple to pass one procedure as an argument with its denoted text being interpreted as needed. Reynolds (1972) noted that the metalanguage itself, however, is not higher-order.

There is an interesting story worth telling about recursion in PL/I. The language control definition team made a mistake in determining the name binding, essentially failing to qualify procedure names in the same way as variable names.[34] Zemanek noted "We had a big battle, for instance, about recursivity in PL/I. The people who really did it had no mathematical training" (Zemanek and Aspray 1987, p. 46). The first LDV note in the file was written by Bekič (1965a) on this subject. He had to convince language control that they had the bindings wrong, and in a memo to the language control board Bekič (1965d) showed that Knuth's 'Man or Boy' problem (Knuth 1964b) did not work properly as per the PL/I rules. Ultimately, Bekič's changes were incorporated into the language; a letter from the chairman of the PL/I language control board reads "The first release of the F-level Compiler will indeed contain an implementation of a special kind of recursion. In the second release, however, recursion will be implemented as defined in Dr. Bekic's report" (quoted in Bekič and Jones 1984, p. XVI). As a result, Hursley and the language control team referred to the correct way as "Bekič's interpretation of recursion" (Bekič 1965c, p. 1). In commentary on this interchange, Jones notes "It now seems reasonable to publish this history: although of no great depth, Hans Bekic's contribution to this debate may have prevented a widely used programming language disseminating a strange form of recursion" (Bekič and Jones 1984, p. XVI).

Once the first version of ULD-III was completed, a review was undertaken by Jim Cox of the language control board. Together with the Vienna Lab, it was agreed that another version of ULD-III was required, and should include new aspects of PL/I, including the compile-time facilities (Cox 1967). This sentiment was echoed by Dave Allen of Hursley who sent a memo to his boss John Nicholls in April 1967 explaining that a new version of ULD-III was required because of the lack of compile-time facilities, and a number of errors in the existing document (Allen 1967). A work-

[34] This was in fact the same mistake McCarthy made in LISP, as discussed in Section 3.1.

shop on the model and its notation would be held in May 1967; in the meantime, language control would use the first version (Cox 1967). Shortly before the workshop, Bandat (1967) wrote an introductory piece on the method, which was also published externally the following year (Bandat 1968). At the workshop, Zemanek distributed an internal 'Review of the formal definition of PL/I' which described the method in glowing terms, and, characteristically, linked his group's semantics work to Wittgenstein and the Wiener Kreis (Zemanek 1967b). Zemanek also chastised those who mixed up the shortcomings of PL/I for those of the ULD method.

The second version of ULD-III, sometimes referred to as ULD Version II, appeared as multiple reports in June 1968 (Alber, Oliva, and Urscler 1968; Lucas et al. 1968; Walk et al. 1968; Alber and Oliva 1968; Fleck and Neuhold 1968). Updated with the new version of PL/I and correcting errors from the previous ULD, it also had improved readability (Walk 2002, p. 81).

The major new feature was an axiomatic model of the storage of the PL/I machine. This definition, given implicitly in contrast to the rest of the model, means that the word 'constructive', often applied to the ULD definitions, is inaccurate, argued Lucas (1981, p. 555). A simple mapping of storage locations to values would be inadequate to cope with all the properties of PL/I variables, such as the explicit allocation and freeing of storage, and the handling of pointers. Instead, "certain domains are postulated together with functions and predicates on these domains; the properties of the domains, functions, and predicates are given by axioms" (Lucas 1981, p. 555). This strategy was first described in the explanatory guide to the metalanguage written as part of the ULD Version II effort (Lucas, Lauer, and Stigleitner 1968), and was generalised by Bekič and Walk (1971) to also allow description of ALGOL 68's storage.

The notational guide just mentioned (Lucas, Lauer, and Stigleitner 1968) was intended to address some of the concerns of IBM management who had expressed a desire for a primer on the method and notation. The preface explained that the ULD approach was considered as a method of general applicability:

> The applicability of the method developed by the Vienna Laboratory in the course of the formal definition of PL/I is, however, not limited to any specific programming language and it is the intent of the authors to present here those features of the method which reflect its generality. (Lucas, Lauer, and Stigleitner 1968, p. i)

The inclusion of formal modelling of the PL/I compile-time facilities brought ULD

Version II in line with the current version of PL/I at the time. These compile-time facilities were essentially another, separate, high-level language, whose purpose was the textual manipulation of PL/I programs. The formal definition uses an interpreter written in a similar notation to the main PL/I definition to process out the compile-time functionality in PL/I functions, leaving a concrete PL/I program suitable for being fed into the ULD translation and interpretation machines (Fleck and Neuhold 1968, pp. 1-1–2).

A third and final version of ULD, called ULD Version III, was produced in 1969, also in multiple reports (Walk et al. 1969; Alber et al. 1969; Fleck 1969; Urschler 1969a,b). None of the documents make it clear what changes were made from the second version, apart from covering the most recent version of PL/I. The preface notes that this now includes "attention handling, input stream and string scanning, and file variables" (Walk et al. 1969, p. iii). Publication of ULD Version III was also accompanied by an external publication giving an extensive description of the meta-language and an overview of the definition of PL/I by Lucas and Walk (1969).

5.6 Reaction to ULD

Looking retrospectively at the achievements of the Vienna group on language definition, Walk (2002, pp. 81–2) wrote "The ULD project demonstrated that formal definition of a language the size of PL/I was actually possible and it gave insights into properties of PL/I that were difficult to grasp without a formal notation". In contrast, Lucas (1981, p. 550) summed up the goals which were desired:

> The intended practical role of a formal definition, of both syntax and semantics alike, is threefold: first, to provide an authoritative reference document from which, possibly less formal, user manuals can be derived, and also as an arbiter concerning subtle questions of form and meaning; second, to provide a basis from which implementations can be derived systematically, if not formally or even mechanically, thus rendering portability of programs as an uncompromised reality. A third role for a formal definition is that it provides a basis for reasoning about programs, for program proofs.

This would require that the formal definition be the official standard, and be available before implementation. The ULD definitions did not enjoy either of these conditions; Lucas (1981, p. 559) explained that part of the problem was the lack of

proper education of computing professionals on the concepts of formal definition. Frustration at the difficulty of modelling such a huge language as PL/I and the resultant complexity of the definition was felt by the team: Maurer left in the middle of 1966 partly as a result of this.[35]

In early 1967, the first version of ULD-III was sent to various members of IBM language development. A memo from G. W. Bonsall et al. (1967) of language control to Jim Cox at Hursley summarised these reactions; it was also distributed to the Vienna Lab.

Bonsall began with some comments of his own. They are faintly optimistic about the idea of a formal definition in general, but rather condemning of ULD-III in particular. He noted "we have not attempted a detailed study or check on its technical accuracy" as his team was put off by the size and complexity.[36] One thing which is interesting to note is that Language Development appears to have considered ULD-III firstly as an aid to language extension and secondly as a means of replacing the SRL. Bonsall listed ten questions that must be answered when considering a language feature, and stated that ULD-III only assisted in two of these. Therefore it did not achieve its intended purpose (as perceived by language control) of aiding the process of language extension. Bonsall wryly noted, echoing Beech's comments, that trying to use ULD-III to handle PL/I is like being asked to read *Principia Mathematica* (Russell and Whitehead 1912) in order to do arithmetic, and commented that better teaching might be needed. Following his own comments, Bonsall included a number of attachments with reactions from other members of IBM management.

In M. R. Barnett's attachment 'ULD-III in language development', the author noted that workers developing a language need a clear understanding of its principles, and that the natural language standard was not useful for this as it gave a set of statements which were answers to presupposed questions about the language. A good model is needed in order to allow users to deduce their own answers to their own questions. ULD-III, however, did not provide these fundamental philosophies, and rather just translated the SRL into mathematics without any insight into the connectivity of the language as a whole. He was one of the commentators whose call for a guide to the method led to the publication of just such a document alongside ULD Version II.

[35] Personal communication, 2016.

[36] The document was known somewhat humorously as "the Vienna telephone directory" (Rosen 1972, p. 592, quoted in Peláez Valdez 1988, p. 92).

H. Morrison's 'Comments on ULD-3' noted that the document was frightening at first glance; with more familiarity the situation improved, but using the document was never enjoyable. PL/I was intended to be easy to use for its customers; this advantage would be lost by requiring learning of a harder metalanguage. ULD-3, argued Morrison, was of no use to customers, and as customers and implementers should share the same standard, of little use to implementers either. He believed the SRL could become adequate for this purpose if the time and talent spent on ULD had been spent on the SRL instead.

Shortly after this round of feedback, in March 1967, Radin (1967), then working in language development, wrote to Zemanek to give his own views on ULD-III. According to Walk and Jones, Radin was well-known by the Vienna Lab and was a friend of the group.[37] Radin, in the letter to Zemanek, said he was amazed by the outstanding contribution they had managed in such a short time. He particularly noted the significance of the abstract machine, and that their abstract interpreter solution was a much better approach than trying to define the machine using a compiler. However, he had qualms about ULD-III. He wrote that the structure and definitions of the model "don't serve to display the structure [of PL/I] and its basic concepts. One gets the feeling it is quite accidental that the abstract machine is executing PL/I as opposed to another language which may be drastically different" (Radin 1967).

In fact, Radin was exactly right about that. The Vienna group were indeed attempting to make a universal method for language definition, although the generic approach could also have been a reaction to the moving targets of both the language itself, and the machines for which it was being developed. The introduction to the first version of ULD-III (1966, p. 1-1) makes the desire for universal applicability very clear, stating "The definition method, evidently, is applicable to a class of languages where PL/I is only a member". Walk (2002, pp. 81–2) also noted that there was an effort to make the mechanism as abstract and generally applicable as possible.

A further criticism from Radin was that the structure of the definitions did not lend itself well to language modification, which was one of the hopes of the language management; or to identifying areas of PL/I where improvement was needed. However, Neuhold (1971, pp. 110–1), in a later paper explaining the basic concepts of the approach, disagreed, writing "formally defining language extensions helps to contain the often serious problems of identifying all parts of the language affected

[37] Personal communications, 2016.

and possibly changing them to accommodate the new facilities".

Another IBM critic was Dave Allen of IBM Hursley, who wrote that users of PL/I were unlikely to accept ULD-III because of its size, notation, and lack of explanation (Allen 1967). It did not suit language extension or product testing, but he suggested an 'implementable' version of ULD could help with testing—and the style in which it was written might lend itself better to this goal than language definition methods based on the lambda calculus.

Zemanek, understandably, was not pleased with the very negative reaction from within IBM:

> IBM management was deeply disappointed because they wanted the following: they wanted you to come in with a piece of text and the Vienna Laboratory to produce an engine, meaning a program, so that you insert the text into the machine and on the output you get the meaning. That does not work. They were deeply disappointed. They thought you had the meaning machine, which produces the meaning in a written form. This is impossible. Meaning is much more on the mental side and meaning actually makes sense only together with the human mind's reaction.
> (Zemanek and Aspray 1987, p. 46)

Such a 'meaning machine', Zemanek continued, would be impossible to write because of problems of undecidability. Zemanek also noted that IBM management was surprised at the problems the definition highlighted in PL/I:

> Moreover, in particular in the case of PL/I, the people's reaction was: PL/I cannot be as ugly as it appears in your definition. The fact is, of course, it is as ugly. Let me take default definitions and all the exceptions (they have made for 100 good reasons), as example. They make the whole language such a complicated thing. So that I used to make the joke that PL/I is the language where a description of the subset is longer than the description of the full language.
> (Zemanek and Aspray 1987, p. 50)

The reaction from IBM fits with Zemanek's assessment of their attitude towards his team's work in general: they were not particularly interested in trying to gain technically from their experience (Zemanek and Aspray 1987, p. 61)

Walk (2002, p. 82) explained that he believed another reason for difficulties was that the language had already been (mostly) completed before work on the definition started:

> To note, the main cause of difficulties and tediousness to cope with was the fact that the language existed first, before the universe, or better call it the abstract system, was defined it was supposed to address. In hindsight, it is the wrong order to create notation first, and then to make the attempt to clarify its meaning by inventing an appropriate abstract system.

Response from the academic community at large was mixed. The ULD definitions were generally agreed as the first fully-formed complete system applied to a full language, as Sammet (1972, p. 607) noted in her early history of programming languages: "the first actual development of the formal semantic definition of a major language was done by the IBM Vienna Laboratory." Generally, as in a survey of formal descriptions written by Marcotty, Ledgard, and Bochmann (1976), the completeness and detail of the ULD system was praised, as well as its ability to detect errors and inconsistencies in languages, but the complexity and lack of clarity was criticised.

A few presentations on the Vienna method were given at IFIP WG 2.2[38] meetings, and Lucas would often comment on claims from other members that the proving of deep properties about programs was difficult. For example, at the group's second meeting, he said "Proofs based on the Vienna method will have real practical application, e.g. the proofs of correctness of compiler methods" (Lucas, in Walk 1968, p. 14). However, little was said in response to this. The same was true at the fourth meeting, as Lucas explained some of the important proofs made by the Vienna group with no real impact on the meeting. Instead, McCarthy criticised the approach for not being able to model termination, adding "It is a bad aspect of Vienna work that people were forced to define PL/I, which is too complicated to be done nicely" (McCarthy, in Walk 1969b, p. 16). Some praise was given for the constructive approach taken, with Paul (in Walk 1968, p. 22) saying "A constructive definition conforms better with the working of a programmer". Pair (in Walk 1968, p. 23) summed up: "The Vienna definition is a compromise between simplicity and implementation. Its generality makes it difficult to use."

R. Tabory, who taught at IBM Intermediate Programming Training, said at a later WG 2.2 meeting (Walk 1970, p. 3) that attempts to teach the Vienna method had not been very successful. In general, he said, "Formal methods are counterproductive in a practical sense and, furthermore, in compiler production, the big

[38] The working group's subject was formal description of programming languages; some of its history is described in Section 7.2.

problems are system-oriented and not language oriented.". Lucas objected to this, saying that the applicability to compiler correctness was very important, and also the use of abstraction could suggest more efficient solutions for implementation.

In a 1974 letter to Hans Bekič Dijkstra gave his thoughts on interpreter methods like ULD for programming language description:

> My suspicions with regard to defining the semantics of programming languages by means of an interpreter are of quite long standing, at least compared to the youth of our field. My first suspicion was roused by the observation that whenever we invoke via our program an available primitive –such as an ADD-instruction, say–[39] its occurrence in that program can only be justified by what that instruction achieves, by "what it does for you", rather than by "how it works".
> (Dijkstra 1974b)

However, this was exactly how Dijkstra and others had described the functioning of their ARRA II computer in 1953 (quoted in Dijkstra 1980):

> In the sequel this machine will be described as far as is relevant for the person that uses the machine: we shall describe what the machine does, and not how the machine works. (Underlining as in the original Dutch text.)[40]

Dijkstra found that his fears about interpreters were confirmed when he received a copy of the ULD definition of PL/I; in the same letter to Bekič he (characteristically emotively) described this, including a couple of sly digs at their choice of approach cleverly disguised as compliments:

> When I first heard that the Vienna crew were tackling the formal definition of PL/I, I was horrified, because on account of what I knew of it, PL/I seemed to me one of the most unattractive objects to give a formal definition of. I certainly did not envy them their job. The next thing I heard was that a definition by means of an abstract interpreter was chosen, a message to which I could give only one interpretation, viz. that, indeed, PL/I was a most unattractive object to give a formal definition of. Without further knowledge I have assumed without hesitation that

[39] This particular typography of dashes—essentially used as brackets—is a hallmark of Dijkstra's writing style. It appears to have some wider, although by no means exclusive, use in the Netherlands. This is mentioned only so the reader does not infer some copying error in the reproduction of this quotation.

[40] Parenthetical comment from Dijkstra's commentary.

> 'an interpreter' was at least your second choice, as I could not envisage the Vienna crew not sharing a good fraction of my misgivings about mechanistic definition of semantics. (And I still believe that they are shared by many of its members.) And when I received what is known as 'The Vienna Telephone Directory', I was once more horrified; also very much impressed. I had not the slightest intention of being 'unfair' to the Vienna crew and, if you smelled criticism, it was criticism regarding PL/I rather than VDL. But you understand that the whole project struck me as a relatively successful attempt to make the best of a bad job.
> (Dijkstra 1974b)

The 'grand state' approach, with many complex components, was mentioned by McCarthy (1966, p. 11) as the way in which he would have handled a more complex language, and later (in Walk 1969b, p. 33) he also argued that the idea of a large state representing different elements of computation could be useful to an implementer. However, the complexity of this state was very off-putting to Gordon Plotkin (2004, p. 4), who wrote:

> I remember attending a seminar at Edinburgh where the intricacies of their PL/I abstract machine were explained. The states of these machines are tuples of various kinds of complex trees and there is also a stack of environments; the transition rules involve much tree traversal to access syntactical control points, handle jumps, and to manage concurrency. I recall not much liking this way of doing operational semantics. It seemed far too complex, burying essential semantical ideas in masses of detail; further, the machine states were too big. The lesson I took from this was that abstract interpreting machines do not scale up well when used as a human-oriented method of specification for real languages.

This influenced Plotkin's work on first denotational and later operational semantics; he kept a focus on a small state throughout.[41]

The state complexity was also criticised by JAN Lee, who noted that a lot of it came from the need to handle aspects unique to PL/I and so would not have been necessary when describing other languages (Lee and Delmore 1969). Another difficulty highlighted by Lee was the inability to easily make changes to large parts of the state as the method allowed only immediate components to be changed easily. This was a result of the universal notion of object. However, Lee did acknowledge that the ULD method had a number of useful attributes, and that most of the problems came from the close association with PL/I. He demonstrated the general utility of the method

[41] More on Plotkin's contributions is discussed in Section 7.4.

by providing a description of BASIC in the ULD style, as well as giving a new name for the approach: 'Vienna Definition Language', or VDL, a name which stuck.

This criticism of too much association with PL/I must have rankled the members of the Vienna group: it was a common point made by people outside of IBM. However, as seen earlier in Radin's comments, people inside IBM criticised the descriptions for being *too* general and not sufficiently tied to PL/I.

In summary, Zemanek said he felt his group was hard done by: "I think that we did not find for what we have done there the recognition and the compensation that we have deserved" (Zemanek and Aspray 1987, p. 47). A large part of the problem, he reasoned, was that for serious uptake by IBM, all of their programmers would have to be educated in the method, as well as every user of the language. It would be too costly to do that, and with informatics training at universities still rather basic, the group's work was at the very least too far ahead of its time (Zemanek and Aspray 1987, p. 48). Zemanek did acknowledge problems in full formal definition, though: complete language coverage was very time-costly, and in some ways intuitive understanding is simply more straightforward and easier to acquire (Zemanek and Aspray 1987, pp. 48–9). This was part of the reason that Zemanek shifted his focus away from full formalism to abstract problems in computer architecture; he decided he "would less care for technical details and more for a universal overview" (Zemanek and Aspray 1987, p. 60).

5.7 VDL: flawed but generally applicable

The ULD definitions of PL/I were heavily criticised from all sides, but this did not prevent the Vienna Lab from finding other uses for their formalism beyond the definition of one language. The general utility of the method was demonstrated with a formal definition of ALGOL 60, completed in 1968 by Peter Lauer.

Lauer was a member of the Lab from 1967 to 1972, and was, according to his colleague Wolfgang Henhapl, the specialist logician. He wrote some theoretical pieces on algorithms (Lauer 1967, 1968a) as well as contributing to the method and notation piece for ULD Version II (Lucas, Lauer, and Stigleitner 1968) and the definition ULD Version III.

By the time work started on the formal definition of ALGOL 60 (Lauer 1968b), the ULD method had already been used to describe the entirety of PL/I, a much larger

language, and the technique was reasonably mature, with Version II published. A definition of ALGOL 60 was likely chosen as it was a simpler and more elegant language than PL/I, and so could be a response to the criticisms that the method would inevitably produce large and unwieldy descriptions. Zemanek was fond of ALGOL 60, as evidenced by his decision to produce an ALGOL 60 compiler for Mailüfterl, and a strong critic of ALGOL 68, which was published around the same time as Lauer's ALGOL 60 definition. Lauer was also still fairly new to the Lab at the time of his writing the piece, and the task would have allowed him to familiarise himself with the method and approach. For similar reasons, Jones was chosen to proofread the ALGOL 60 definition as one of his first tasks upon arriving in Vienna in August 1968.[42]

Lauer modelled the entirety of ALGOL 60, including the contentious 'own' variables. This required an awkward first step: a pass made over the program translating all own variables into uniquely generated integer identifiers. The clunkiness of this somewhat counteracted the aim of a smaller language definition, but Lauer's description is unique amongst formal descriptions of ALGOL 60 in actually modelling the concept.[43] Another interesting aspect of the description is that the translator turns abstract programs into concrete syntax rather than the other way around, as had been done for the PL/I definitions. This direction is similar to Landin's approach, and works for ALGOL only because of its relative paucity of syntactic redundancy.

Following his work on the ALGOL 60 definition, Lauer took a year's leave of absence (1970–71) and went to Queen's University Belfast to study a PhD under Tony Hoare. He returned after this year and finished writing up his thesis whilst back working at the Vienna Lab, showing again Zemanek's support for his workers to achieve qualifications. This work showed the consistency of Hoare's axiomatic semantics with respect to an operational model and represents an interesting early attempt to link different approaches of programming language description (Lauer 1971).

Another area in which it had been hoped that the ULD definitions would be useful was in compiler development. The Vienna Lab was not directly involved in this, but thoughts about their work being used for this purpose existed at the time. The introduction to ULD-III (1966, p. 1-2) states:

[42] Jones, personal communication, 2016.

[43] For more on the technicalities of this description, and for discussion of three other ALGOL 60 formal descriptions, see Astarte and Jones (2018).

> It should be clear that the definition does not solve the problem of compiler-design for PL/I, specifically it does not reflect any considerations of efficient implementation. On the other hand, the definition may be considered to state precisely the problem to the compiler designer. The definition also specifies for him those areas of the language where he is free to choose any interpretation out of a given class.

Walk (2002, p. 82) wrote that one of the questions that remained after finishing the definitions was about their utility in proof:

> Is the definition adequate for attempts to formally prove general properties of the defined language? The principal answer is yes and practical feasibility was shown with smaller examples. Proofs, however, may become unsurmountably lengthy and tedious for a language the size of PL/I.

These "smaller examples" were largely the work of Lucas. He argued later that rather than creating a whole formal model and a full language from it, and then trying to do a monumental proof of equivalence, it was better to isolate individual language concepts and make proofs about their accuracy (Lucas 1981, p. 559).

The first application of this principle was to the block structure of PL/I, published in a report entitled 'Two constructive realisations of the block concept and their equivalence' (Lucas 1968). Lucas used the idea of a 'twin machine' which linked the model used for stack variables in ULD-III with its counterpart in Hursley' ULD-II. The approach to establishing equivalence was as follows: combine the states of the two interpreting machines and link them by logical constraints on their properties such that when one machine executes, so must the other. Lucas then showed that if the combined machine preserved the linking invariant, one could then "erase the algorithm that one no longer required". These proofs were in some ways similar in concept to those performed by McCarthy and Painter, but ended up considerably larger due to the size of the models.

Continuing the thread of proof work was the main aim of Jones' assignment to Vienna for the years 1968 to 1970. His mission was to determine whether Hursley's difficulties in producing the PL/I compiler could be solved by basing their design on a formal description. Some results were published in a paper on proving the correctness of implementation techniques authored by Jones and Lucas (1970). This demonstrated that Dijkstra's 'display' method was a valid implementation of the

ULD-III model of referencing stack variables. Although this showed that proofs about compiling techniques for language concepts could be based on ULD-style descriptions, it also indicated that it was more difficult than need be. An essential step and key lemma (Lemma 10) in that paper had to show that the environment was the same for the execution of successive statements in a given block even though the first statement could be a nested block or a procedure call whose interpretation required that a new environment be temporarily used. Due to the stack of environments in the state, the proof of this lemma was gratuitously difficult. It was thus clear to Jones, even in operational semantics, that the traditional 'grand state' made reasoning difficult and that a 'small state' approach would be preferable (Astarte and Jones 2018, p. 99).

Jones also produced that year, along with Henhapl, a report showing that a ULD model could be used as a basis for a post-facto proof about the stack variable reference mechanism used in Hursley (Henhapl and Jones 1970b). This attempt to establish correctness revealed that a number of bugs were in the Hursley PL/I compiler. These were subsequently corrected, but the Hursley developers took the attitude that a couple of months of a mathematician's time was too high a price to pay to uncover errors in their products (Astarte and Jones 2018, p. 98).

The proof line of work continued into ideas of refinement and program development: Lucas (1972) wrote about axioms that could be used to define an 'abstract data type' stack and the relation of this device to its implementation. A further paper was on the stepwise development of programs based on ideas of formal semantics (Lucas 1973).

Another outcome of the ULD work, according to Zemanek, was the cleaning up of PL/I. Every request sent to language control helped fix a problem that had not been identified (Zemanek and Aspray 1987, p. 49). This led the Vienna Group to believe it would be preferable to use formalism in the design process for both programming languages and programs, the latter thread of which continued from Lucas' ideas mentioned above and into the VDM approach, carried forward by Dines Bjørner and Jones[44] (Zemanek and Aspray 1987, p. 50).

The VDL approach was used by a few groups outside of the Vienna Lab. The ANSI (1976) PL/I standard uses some ideas from the ULD definitions; employing the same basic components: concrete syntax, abstract syntax, translator, and abstract

[44]See further commentary on this topic in Section 7.3.

machine. The states of the machine are formalised, although in a different way to the VDL style, with a closer linking to implementation data structures;[45] and the state transitions are written in stylised English rather than a formal notation (Lucas 1981, p. 553). A useful guide to the ANSI standard was written by Beech and Marcotty (1973) and shows the clear influence from the Vienna method. It is interesting to observe that despite the participation of Beech in the ANSI work, there is no reference made to the Hursley ULD-II definition, as there is a clear influence from that work as well. A formal definition of large parts of FORTRAN in the VDL style was published by Zimmermann (1969), and Lee (1972) wrote a full formal description of BASIC. Much later, Arbab and Berry (1987) based a denotational semantics of Prolog on a VDL definition.

Neuhold continued to publish on VDL after leaving the Vienna Lab, writing a good primer about the method with examples (Neuhold 1971). A similar paper was published by Wegner at Brown University, showing there was some international interest in the approach (Wegner 1972). In 1969, Jaco de Bakker was teaching on the PL/I definition method at Leiden; Lucas also taught at TUW on how to create language implementations based on the idea of an abstract interpreter (Walk 1970, p. 3).

In conclusion, the Vienna group achieved their goal of fully formalising the syntax and semantics of PL/I, and managed to show its applicability in formal proofs of correctness, but at the cost of an unwieldy and large, if powerful, formalism. The ULD definitions of PL/I were the first members of a very small group of realistically-sized languages that received a full formal definition. The cool reaction of members of IFIP's Working Group 2.2 reflects that of the academic community as a whole; in particular, the large state was considered a real stumbling block for formal description. This led to a general attitude of negativity towards operational semantics (which is to say semantics based on an abstract machine), and it was in this atmosphere that Christopher Strachey's 'mathematical' approach to semantics, first seen in its infancy at *Formal Language Description Languages*, really began to flower. This is the subject of the next chapter.

[45] Part of this change was that certain abstractions such as sets were omitted and lists used instead. This led to difficulties for implementers, who had to check every use of any such construction to check that the ordering was not important when creating their run-time state (Jones 1999, p. 29)

CHAPTER 6

Foundations and functions at the Oxford Programming Research Group

By the end of the 1960s, there were a number of different techniques in use for describing semantics of programming languages, but for Christopher Strachey, none of them had a sufficiently *mathematical* approach. Strachey was the driving force behind a definition method which, in an attempt to address this, used the concept of functions to define the meaning of programming languages. The other major contributor to the work was Dana Scott, a logician, who provided the necessary fundamental basis for the work to be mathematically sound; further contributions were made by a number of others, especially students and researchers at the Programming Research Group at Oxford University towards the end of the 1960s and throughout the early 1970s. The basic concepts of what came to be called denotational semantics were relatively straightforward and tractable, but handling increasingly complex programming constructs required ever more convoluted contortions. This chapter tells the story of denotational semantics at Oxford University.

Section 6.1 begins with a look at the life of Strachey. Section 6.2 explores his time working on programming languages and codes, especially CPL. Section 6.3 looks at Strachey's early attempts to work out semantics. Section 6.4 discusses the impact of the Scott and his work codifying the foundations of Strachey's approach. Section 6.5 considers the various expansions of the method, especially thanks to students at Strachey's Programming Research Group. Finally, Section 6.6 looks at reception to the approach and further work. A timeline of events from this chapter can be found in Figure 6.1.

A brief note on nomenclature is in order. The technique was typically called 'mathematical semantics' in its first iterations, or 'Scott-Strachey' semantics after its chief contributors. Strachey himself tended not to use these terms, referring instead just to 'semantics', 'formal semantics', or 'standard semantics'. Stoy, Strachey's chief assistant at Oxford, noted

> We didn't call it 'denotational semantics'; it was 'mathematical semantics'. Slightly pejorative, because it implied that all the other ways of defining semantics were non-mathematical, whereas in fact they could be formalised pretty well as well. So we backed off. Christopher made a point of saying that his semantics was mathematical semantics.
> (Stoy 2016b)

Finally, a note on sourcing: a large part of the content in this chapter is drawn from material held in the archive of Strachey's papers. These are housed in the Bodleian Library in Oxford, Department of Special Collections. Original cataloguing was performed by archivists from the Contemporary Scientific Archives Centre, Jeannine Alton, Harriot Weiskittel and Julia Latham-Jackson.[1] Published sources are cited in this chapter in standard style, where sufficient bibliographic information is available; for some others, only a box and folder reference to the archive is given. The full citations can be made up by completing the template 'MS. Eng. misc. b. $X/Y.Z$' where X is a three digit number, Y is a letter, and Z is a number.

[1] Online version available at `http://www.bodley.ox.ac.uk/dept/scwmss/wmss/online/modern/strachey-c/strachey-c.html`

Figure 6.1: A timeline of important events in the story of denotational semantics.

Jul 1951	Strachey visits the Mark I computer at Manchester and begins experimenting with programming
Nov 1951	Strachey takes a job at the National Research Development Corporation, programming and developing microcodes
Apr 1959	Scott works with Rabin on finite automata
Jun 1959	Strachey sets up a consulting business and employs Landin, who gets him interested in lambda calculus
Jun 1962	Work begins on CPL with Strachey a main contributor
Mid 1963	Together with Gill, Strachey organises a summer school in nonnumerical computing at Oxford
Sep 1964	Motivated by the difficulties of defining CPL, Strachey presents some ideas on formal semantics at FLDL
Apr 1966	Strachey sets up the Oxford Programming Research Group
Aug 1967	Strachey develops his semantics ideas and writes 'Fundamental concepts in programming languages'
1968–1969	Scott works with de Bakker at MC on a theory of programs
Apr 1969	Strachey and Scott meet at a WG 2.2 meeting
Late 1969	Scott spends a term at the Programming Research Group and develops a logical foundation for Strachey's semantics
Oct 1971	Scott and Strachey publish the first major joint piece explaining their 'mathematical' semantics
Jan 1974	Publication of Wadsworth and Strachey's paper on continuations, semantics for jumps
Jan 1974	A formal description of ALGOL 60 is written by Mosses
Jun 1974	Mosses publishes MSL, a formalised notation for the semantics metalanguage
Late 1974	Milne and Strachey write an extensive essay on semantics
Apr 1975	Using MSL, Mosses writes SIS: a system which generates compilers from semantic descriptions
May 1975	Strachey suddenly dies of infectious hepatitis
1976	Milne rewrites the joint essay as a book
1977	Stoy publishes a textbook on denotational semantics

6.1 Christopher Strachey

The main force behind the mathematical approach to semantics was Christopher S. Strachey (1916–1975).[2] A comprehensive biographical study has been written by Campbell-Kelly (1985), but the relevant parts of Strachey's life will be covered in this section.

Christopher was born into the Strachey family, part of the well-known Bloomsbury group of early twentieth century intellectuals, which also included the writers Virginia and Leonard Woolf and the economist John Maynard Keynes. Christopher's uncle was the writer Lytton Strachey, the biographer of Queen Victoria; Christopher's father Oliver was a cryptographer who worked at Bletchley Park during the Second World War, and later at Canada's cryptographic Examination Unit.

Christopher's mother Ray Costelloe was an American suffragist and writer of many works on the women's movement including *The Cause* (Strachey 1928). Unusually for a women of her time, she was trained as a mathematician and electrical engineer and, from her, Christopher took a deep love of mathematics. Ray and Oliver had two children; Christopher's older sister was named Barbara (later taking the surname Halpern) and she became a writer herself, writing, amongst other things, a history of her mother's family named *Remarkable Relations: the Story of the Pearsall Smith Family* (Strachey 1980). A photograph of the young Strachey children with their mother can be seen in Figure 6.2.

From his father Christopher inherited a love of puzzles and games of many kinds:[3]

[2] Most references to Strachey do not include his middle initial, which is given here only for completeness.

[3] Another nice story is found in a letter from 'Joe' (likely Stoy) to 'Barbara' (likely Strachey). It is undated and is kept in the Strachey archive (Box 253, Folder A.72). Joe explains:

> Peter [likely Mosses] told us of the "family rumour" that Christopher once won the *New Scientist* competition three times running and was thereafter barred from competing. We rang the *New Scientist* who denied all knowledge and said it was impossible.

However, they searched in the library, and discovered that Strachey had entered a competition run three times in T. H. O'Beirne's column in 1961, coming runner up once and winning twice.

Rules for the first competition read "Evident use of a computer will constitute a misdemeanour". and the announcement of his victory noted "We are aware that to forgo use of a computer is no mere *necessary* virtue, in *his* case". The third competition's results announcement included the message "We are unable to deny the first prize of £5 to Mr. C. Strachey (London) who now has two firsts and a second in our three competitions; we must award solely on merit as we see it, and we have no wish to tempt him to submit entries other than in his own name". The competition, however, ceased its run after 1961.

> Christopher was a complete Strachey, all sensitivity and intellect, who was heard explaining to his nurse at the age of five what a gradient of one in four meant, and who used to insist on playing imaginary three-dimensional noughts and crosses in his mother's bed in the early morning.
> (Strachey 1980, p. 275, quoted in Campbell-Kelly 1985, p. 20)

Strachey also enjoyed posing puzzles to people he knew, even before he got to know them. Indeed, Dijkstra (1970a) remembered that on first meeting Strachey (via van Wijngaarden), Strachey posed him a puzzle to solve regarding a transposition algorithm.

Christopher Strachey's[4] education began at Gresham's School in Norfolk where he was academically "undistinguished", although he could perform exceptionally well when it suited him (Campbell-Kelly 1985, p. 21). This was reflected in a vocational guidance report on Strachey in 1938, where he was indicated as a person of "very marked ability", scoring considerably above the average on intelligence and practical tests[5] (Rodger 1938). Strachey attended King's College, Cambridge, in 1935, where he studied the Mathematical Tripos, before transferring to Physics. King's was renowned for homosexuality and Marxism at the time, and though Strachey showed interest in the first, he never demonstrated much political allegiance throughout his life (Campbell-Kelly 1985, p. 21).

Strachey got his first introduction to computing machines while working for Standard Telephones and Cables during the Second World War, using a differential analyser to help with tricky calculations, which sparked his interested in the subject. Strachey's team leader noted of him:

> Strachey's brightness didn't make him the most useful in industrial research. In any new problem that came up he was extraordinarily good; seeing the implications and useful directions of work better than any of the others, but he was difficult to keep on a long and detailed investigation.
> (Fremlin, quoted in Campbell-Kelly 1985, p. 22)

[4] From this point onwards, references to 'Strachey' mean Christopher unless specifically indicated otherwise.

[5] The same report suggested that he would likely be suited to research work or the legal profession, although his preference for the direction of scientific research was noted even at the age of 21. It further remarked upon his likely success at teaching, though more at a university level than school.

Figure 6.2: Christopher Strachey, with his mother Ray and elder sister Barbara. Photograph taken in 1928, at the Mud House in Surrey, which Ray had built herself, and where Strachey spent many happy years as a child and adult (Campbell-Kelly 1985, p. 21). National Portrait Gallery Photographs Collection, NPG Ax161158.

This was very characteristic of Strachey: he was very interested in problems and novel solutions to them, but was less able to see through the minutiae of completion and even less keen to produce complete publications.

Following the war, Strachey spent six years working as a schoolmaster, first at St. Edmund's in Canterbury and then at Harrow School. He found joy in encouraging his pupils to think deeply, be imaginative, and appreciate music; he also ran a number of extra-curricular activities including lectures on wildly varying subjects such as interplanetary travel and the surface tension of soap films (Campbell-Kelly 1985, p. 23). One of Strachey's students was Michael A. Jackson, who went on to be a notable computer scientist himself. Jackson (2000, p. 73) remembers that Strachey conveyed his own wide-ranging interests, including computing, to the boys, much to their advantage:

> At the age of fourteen I, along with some 20 other boys in the Classical Lower Fifth, was fortunate enough to be among his pupils. We had already chosen the focus of our academic work—to specialise in the language, literature and history of the ancient Greeks and Romans. For us the study of mathematics was an option, unconstrained by any examination syllabus or any need to adhere to a predefined curriculum. Christopher treated it as a free-ranging exploration of mathematical ideas. He discussed Klein bottles and Moebius strips. He introduced us to some of Cantor's ideas about countable and uncountable infinities, to group theory and simple applications of matrix algebra. We explored binary notation and other number representations; we analysed the game of nim, and we went to see Nimrod, the Ferranti nim-playing computer exhibited at the Festival of Britain in 1951.

Strachey also developed a schoolmasterly persona which stuck with him all his life: he was given to lengthy pronouncements delivered in an inimitable voice, which shifted rapidly in pitch and speed as he spoke.[6] A photograph of Strachey in 1951 can be seen in Figure 6.3.

During this time Strachey continued to be interested in computers and visited the National Physical Laboratory in 1951 where he learnt about their Pilot ACE computer, which was at that time under construction. He wrote for it a program which would play draughts (checkers)[7] (Campbell-Kelly 1985, p. 24). The Strachey Archive has a lot of diagrams of his plans for the program, one of which is shown in Figure 6.4.

[6] At time of writing, a video clip survives of Strachey speaking at a 1973 debate on artificial intelligence (Strachey 1973c). John McCarthy can also be seen speaking in the same clip.

[7] Strachey wrote a few papers detailing this program, a later one of which—published in

Figure 6.3: Christopher Strachey's passport photograph from 1951. From the Christopher Strachey Collection, Bodleian Library, Oxford. Box 253, A.75.

This experience generally led Strachey to become more interested in non-numerical problems in computing, his major focus as he moved into the field himself. Indeed the title of the paper in which he first presented his draughts program was 'Logical or non-mathematical programmes' (Strachey 1952).

Strachey shared this experience with his pupils:

> Christopher also introduced us to computer programming. At the time, he was writing experimental programs for the computer at Manchester university: neat columns of numeric machine-code instructions, the jumps carefully annotated with pencilled arrows leading to their targets. He encouraged us to write programs of our own, expressed as flowcharts. [...] I remember my pride in my programs for integer multiplication and

Scientific American—introduced some important concepts of data representation (Strachey 1966b), as explained by Burstall (2000, p. 54): "He started with an abstract notion of board state and then refined it to two different representations, an obvious one and an efficient one. It is a beautiful exposition of how to write a program, still valid today."

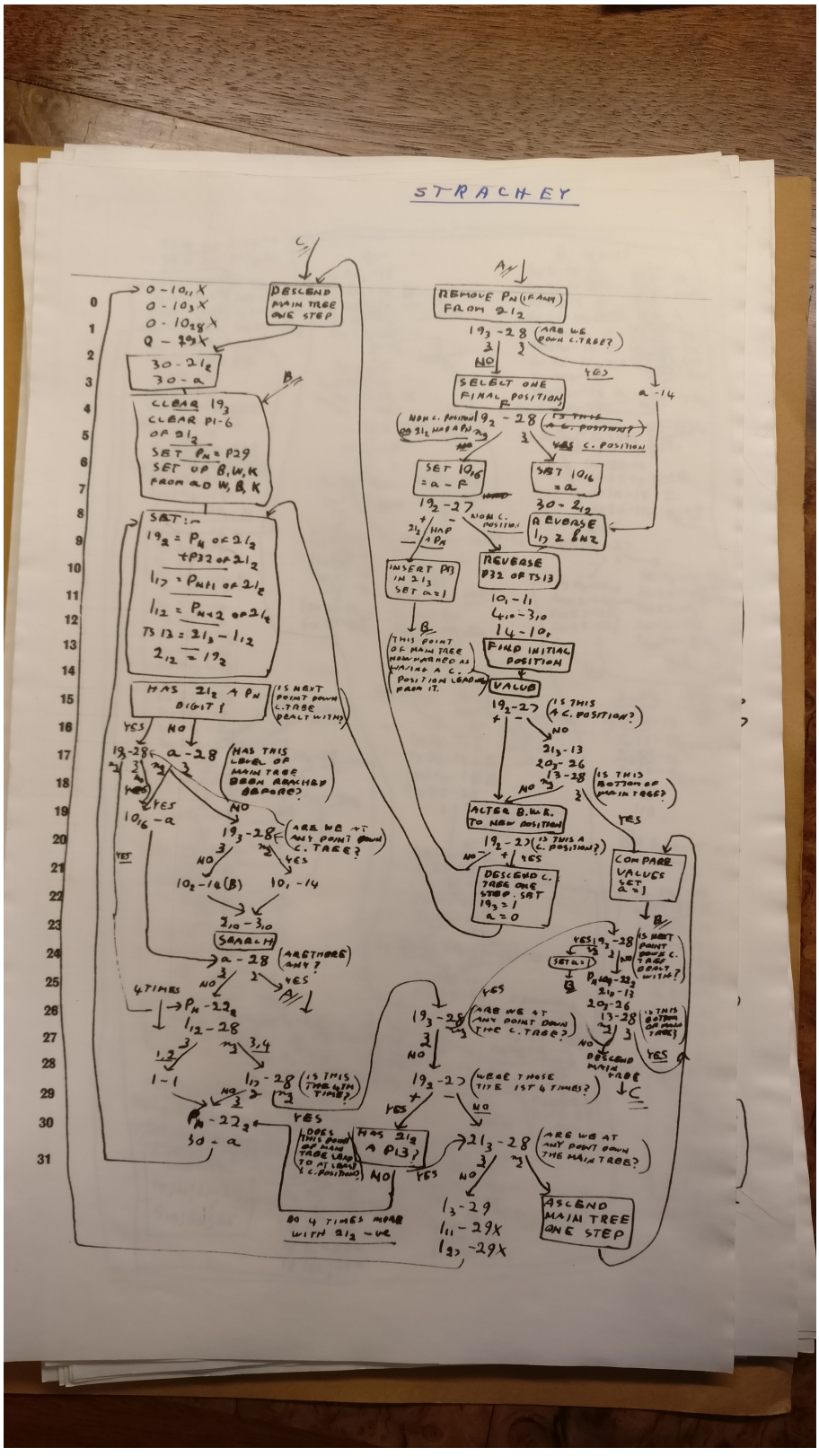


Figure 6.4: A diagram showing some of Strachey's planning for the draughts program. From the Strachey Collection, Box 258.

division.
(Jackson 2000, p. 73)

Strachey's reputation as a programmer was established when he visited Manchester University to see its Ferranti Mark I machine later in 1951. Turing suggested Strachey try writing a program to make the machine simulate itself, which he duly did. The program was by far the longest written for the machine at the time, and when it was completed, the computer played the National Anthem on its 'hooter', in a typically Strachey flourish (Campbell-Kelly 1985, pp. 24–5).

Hearing the success of this, Lord Halsbury offered Strachey a job at the National Research Development Corporation (NRDC), a semi-autonomous organisation whose aim was to move research into commercial applications, and the two also became firm friends (Campbell-Kelly 1985, p. 25). Strachey moved into his sister Barbara's London flat in 1953 and this allowed him to resume his piano playing. He was a great music aficionado and competent pianist; something which many people would remember of him.

Strachey's work at the NRDC included a large quantity of machine code programming, and oversight of much of the design of the Ferranti Pegasus computer in 1955. This included the overall machine architecture and the instruction set, as well as lots of the initial software (Strachey 1971a).[8] All of this gave Strachey deep knowledge of the difficulties of language design and the importance of good programming languages. During this period, he also worked with Roger Penrose, later a mathematician, physicist, and philosopher of science of some renown. Strachey's father and Strachey himself knew Roger's father Lionel, and Roger through him (Strachey 1971b). Indeed, Strachey was responsible for the hiring of Penrose, which was the young man's first job; he worked as a mathematical assistant to Strachey because the latter considered himself "not a mathematician" (Penrose 2000, p. 83). The two worked together on calculations of aircraft wing flutter, and Penrose remembers also having "long and penetrating" discussions on the subject of computers and human minds which "had a considerable influence on [his] own philosophical views" (Penrose 2000, p. 83).

Another important project for Strachey at the NRDC was a program for simulating the backwater fluid dynamics of the St. Lawrence Seaway in Toronto, Canada. Com-

[8] This source is an extended CV sent to the *Times* newspaper in case of their need for obituary information.

pleted in 1956, the program was extremely large—some 10,000 instructions—and its input data tape was estimated at one and a half miles in length (Campbell-Kelly 1985, p. 27). Achieving this successful result, which opened up the American Great Lakes to sea shipping for the first time, on an unreliable computer with a limited instruction set (lacking even built-in floating point arithmetic) was a great undertaking. It cemented Strachey's reputation as "the person who wrote perfect programs", according to C. C. Gotlieb (quoted in Campbell-Kelly 1985, p. 27), who led the computing work at University of Toronto, where Strachey worked for this project. All these experiences at the NRDC gave Strachey a firm grasp of the practical problems of computing on which his later theoretical work could be grounded.

In the late 1950s, towards the end of his time at the NRDC, Strachey was involved in the development of a large high-speed British computer at Manchester University under Tom Kilburn, which eventually became called Atlas. Strachey was uncertain that the project would be successful in the less financially-rich environment in the UK, and withdrew, requesting Kilburn remove his name from the report (Campbell-Kelly 1985, p. 29). However, the prospect of much more powerful computing machines led Strachey to speculate on the topic of multi-user access time-sharing, and he published one of the earliest papers on the topic (Strachey 1959). Strachey later remarked that when he wrote the paper he did not realise how complex and difficult the topic would become; he might not have been so enthusiastic had this been clearer (Strachey 1971a).

Around this time, March 1959, Lord Halsbury resigned from the NRDC to work as a freelance consultant. Strachey, inspired by this and not wishing to work under a new boss, decided to follow the same path (Campbell-Kelly 1985, p. 30). The business took off almost immediately, particularly as Strachey was able to continue working on a number of projects started at the NRDC.

6.2 Problems in programming languages

Strachey's private consulting business ran from 1959 to 1964, rather more successfully than he had initially anticipated. His main work was in computer design and programming, including plenty of fairly prosaic application programming. This furthered his knowledge of the "data processing realities of large users" (Campbell-Kelly 1985, p. 31), which would keep him grounded practically during his later time as a university researcher.

From 1960, Strachey employed Peter Landin as his sole employee. One major task for them was the development of a scientific autocode for the new Ferranti Orion machine, which provided more practical programming language design experience for the two. As Strachey noted in his 1971 CV, he also urged Landin to work on high-level programming language theory, adding "It is an interesting comment on the state of the subject that this work which at the time was probably the only work of its sort being carried out anywhere (certainly anywhere in England) was being financed privately by me" (also quoted by Campbell-Kelly (1985, p. 31)).

At his house in Bedford Gardens, accompanied by "three grand pianos and two cats" (Landin 2001), Strachey began to formulate his "grander vision of tugging the study of computing out from furtive commercial patronage and into the universities" (Landin 2000, p. 76). He also hosted monthly discussion meetings with others working in computing, as Barron (1975, p. 8) remembers:

> At this time the research community in computing in this country was small and fairly closely-knit, and Strachey formed a select discussion group for people interested in the then new field of 'automatic programming'. It met monthly at the Strachey family home in Kensington, and I was lucky enough to be able to go along with Maurice Wilkes. The group would converge on Bedford Gardens at mid-morning, and go through the house to a studio at the back, which was furnished with a variety of Victorian chairs and a large number of cushions on the floor. The walls were covered with excruciatingly bad portraits of members of the Bloomsbury Group (most of whom, as Christopher liked to remark, were his uncles and aunts) painted by a member of the family, and there Christopher would hold court (sometimes appearing in a dressing gown), and we would argue the world to rights before dispersing to our various homes in the evening.

During this time, Strachey and Landin would have spent much time discussing their ideas for language definition, although they did not author any works together. Reynolds (1981, p. 1) thought that Landin was likely influenced by unpublished ideas of Strachey. One interesting question is how Strachey came to use lambda calculus in his work. The first introduction was in fact made by Penrose, who wrote:

> I cannot clearly remember at what stage I tried to persuade Christopher Strachey of the virtues of the lambda calculus. As I recall it, my own ideas were along the lines that the operations of the lambda calculus should somehow be 'hard-wired' into the computer itself, rather than that the calculus should feature importantly in a programming language.

> In any case, my recollections are that Strachey was initially rather cool about the whole idea. However, at some point his interest must have picked up, because he borrowed my copy of Church's book and did not return it for a long time, perhaps even years later. When I eventually learnt that he and Dana Scott had picked up on the ideas of lambda calculus, it came as something of a surprise to me, as I do not recall his mentioning to me that he had taken a serious interest in Church's procedures.
> (Penrose 2000, p. 84)

Given this, it seems more likely that it was Landin who influenced Strachey's early thoughts; the success and impact of Landin's work would undoubtedly have been obvious to Strachey. As mentioned earlier (see Section 3.2), Landin knew about lambda calculus from university and was interested in ideas of applying it to programming languages. Strachey probably saw Landin using lambda calculus and recognised it from Penrose's earlier introduction; Landin (2001) noted "If it's true that Penrose had told Strachey about lambda calculus I think he'd forgotten by the time he met me" and Strachey himself later wrote "At the time this suggestion [from Penrose] fell on stony ground and had virtually no influence" (Milne and Strachey 1974, p. 25, quoted by Scott 2000, p. 107). Rod Burstall (2017), who worked with Strachey towards the end of the consulting business, said that Strachey was thinking about using combinators before Landin's influence; although he did also point out that the two were working so closely together it was difficult to pinpoint exactly which ideas came from which person (Burstall 1993). Using combinators would have allowed operators that both changed the state of computation and could produce side effects[9] (Burstall 2000).

Strachey said of Landin's part in the story:

> Although Landin's influence on the developments of our semantic method has been considerable, it has been largely indirect. We prefer in the first instance to use a more abstract approach and the standard semantics differs from the work of Landin in that it does not depend in any way on a machine involving anonymous quantities and linkage information not explicitly available to the programmer.
> (Milne and Strachey 1974, p. 27)

Another area of language investigation for Strachey was LISP. He was contracted by the NRDC and the music company EMI to examine LISP. Although Strachey was

[9] This is remarkably similar to the approach later taken by the Vienna Laboratory when they began to use denotational semantics; see Section 7.3.

"captivated by the possibilities that the language offered" (Campbell-Kelly 1985, p. 32), most British computers, particularly those at universities, did not have sufficient memory to implement a LISP compiler, which served to reinforce Strachey's view that computing in Britain was lagging sorely behind the US.

Another problem was that university research in the UK was not sufficiently focused on exploring the potential of computing applications, instead tending to treat computers purely as machines for numerical calculation. As mentioned in Section 3.2, this was exemplified by Leslie Fox's (1961) article in *The Computer Journal*, which proposed that numerical applications were the best use of computers. Strachey and Stanley Gill struck back in the journal's correspondence, writing:

> [Computers] are destined to play a part so basic and revolutionary that a correct appraisal of them is essential to our survival as an important nation. If this is so then the professional mathematicians must play a leading part [...] as true mathematicians. They must do for computer programs what the famous mathematicians of the past have done for real and complex numbers.
> (Strachey and Gill 1962, quoted in Stoy 2016a)

This shows a similarity with McCarthy's desires, discussed in Section 3.1, to provide a mathematical respectability for computing *itself*, not just as a means to achieving calculation.

Responding to their letter, Fox invited Strachey and Gill to host a summer school at Oxford in 1963, focusing on logical, symbolic, and non-numerical computation. Strachey's contribution was the core course on programming techniques, delivered in collaboration with David Barron. Barron was given the job of writing up the course into a chapter for the school proceedings, a non-trivial task, as he remembered:

> Christopher and I gave the core course on programming, and as he had an ingrained aversion to writing (as opposed to talking), like a Boswell to his Johnson, I had the job of producing a publishable version of the lectures. I had a tape-recorded transcript to work from, but this was hopeless, a typical excerpt being
>
> > ... and so we use a lambda function ... facing the board, inaudible ... and when we substitute ... talking too fast *again* ...
>
> Even if the transcription had been comprehensible, it would not have been much use without a filmed record of the blackboard, since Christopher's lectures were on the 'thinking aloud' variety, involving much use

> of the board duster; stimulating, but impossible to freeze into print. (Barron 1975, pp 8–9)

Scott (2000, p. 104) agreed with this estimation of Strachey's style, writing:

> His lecturing style was by turns crisp, muddled, cryptic, delightful, maddening, sarcastic, always highly spontaneous, but—to those that had ears—unfailingly stimulating.

The proceedings of the summer school (Fox 1966) were very successful: Stoy was given a copy of it and the proceedings of FLDL[10] (Steel 1966a) when he joined Strachey's Programming Research Group in 1967 as the only two books he needed to get into the subject of programming language description (Stoy 2016b).

In 1961, Strachey co-wrote a paper with Maurice Wilkes of the Cambridge University Mathematical Laboratory, the subject of which was problems they perceived with ALGOL 60 (Strachey and Wilkes 1961). The main points made were, contrary to the Amsterdam school, that ALGOL had *too much* generality; in particular, they highlighted that every procedure could be recursive and that side effects could be present in every procedure. The authors proposed extending the notation of ALGOL 60 so that procedures had to be explicitly declared as recursive, and would not be by default. Also, a 'function' designator would be introduced for procedures which were guaranteed to be side-effect free. The idea was that these changes would allow the writing of more efficient compilers; in particular, memory allocation for non-recursive procedures could be reduced, and compilers would be allowed to re-order certain instructions for better performance.

The paper was not well received by the ALGOL community. Wilkes, in a letter to Campbell-Kelly, recalled:

> ALGOL was almost a religion with some people at the time and we did not expect that our paper would please the real devotees. The one idea of permanent value that it contained was contributed by Landin.[11] Incidentally, Strachey insisted that we should use the word "otherwise"

[10] Stoy remarked that they pronounced this 'fiddle'.

[11] This was the suggestion that a **result of** notation should be introduced which allows values to be extracted from procedures without assigning to a variable.

> instead of the ALGOL "else" which he said was ungrammatical.[12] It was characteristic of him to insist on minor points of difference with the same force that he insisted on major points. This made conversation with him somewhat exhausting.
> (Wilkes 1983, quoted in Campbell-Kelly 1985, p. 31)

In particular, the Dutch school of language generalists, exemplified by Dijkstra and van Wijngaarden, did not agree with the points raised by Strachey and Wilkes. Indeed, Dijkstra prepared a riposte, published first as a Mathematisch Centrum Report, and then distributed to the *ALGOL Bulletin* mailing list (Dijkstra 1961b). In this, he argued against the proposals of the two English programmers, particularly that of the recursive marking. In summary, Dijkstra wrote "If these authors had their way, I should have few illusions left about the ease with which the eventual language could be used". He did, however, have the grace to write to Strachey and Wilkes in advance with a warning that he was about to publish, including that his language might be a bit harsh: "If, nevertheless, some of my paragraphs make a nasty impression – unwillingness to understand your honest intention of whatever it may be – please, be assured that this has not been my intention. This should be caused by my restricted master of the English language: the usual subtleness of the British is, alas, beyond my powers" (Dijkstra 1961a). The ideas in the paper, in any case, did not stay with Strachey forever; by 1968, he referred to it as "not a very good paper" (Strachey 1968b).

Building on their shared scepticism of ALGOL 60, in June 1962 Wilkes invited Strachey to work at the Mathematical Laboratory developing a new programming language and compiler for their forthcoming Titan computer (Campbell-Kelly 1985, p. 32). Strachey accepted, taking a post as a Senior Research Fellow at Churchill College (Strachey 1971a). The language project was led by David W. Barron; along-

[12] Another example of Strachey's pedantry is too amusing to omit. On 21st January 1972, he wrote a letter to the *Times*; when it was printed, the editor changed Strachey's use of the word 'program' to 'programme'. Strachey's response was as follows:

> Sir,
> The word *program* (so spelled) is an internationally accepted technical term for the detailed set of instructions given to a computer in the hope that it will perform some task; it is neither an Americanism nor an ignorant misspelling. Your editorial emendation of my letter published today, which makes me refer to the *programme* for a computer, there has had the unexpected effect of introducing a solecism instead of removing one. It seems that Procrustean proclivities are not confined to programs but are spreading, albeit in a more expansive and old fashioned way, to Printing House Square.
> (Strachey 1972b)

side Strachey, David F. Hartley was the other main worker. The language was to be called Cambridge Programming Language, or CPL. Hartley (2000, pp. 69–70) remembers that at first, the team felt a great deal of enthusiasm for the project:

> We had set out to produce an all-purpose language (in the context of a university computing system) which would be perfect in every way. It was to be easy to read, complete in all its functionality and capable of being used by simply everyone. After all, the new Atlas computers were so fast, we had no great need to worry about efficiency.

Strachey was keen to be involved, both because he wanted to begin investigating his ideas of programming languages that had been percolating for some time, and also because he wanted to redress the balance of computing research in universities away from numerical applications (Campbell-Kelly 1985, p. 32). By this time, Strachey was rather well-known and respected; Barron (1975, p. 8) remembers the first time he saw Strachey was at the end of the 1950s, being shown around the laboratory:

> One day, a visitor was shown round: although he did not speak to me (I don't suppose he even noticed me) he left the impression that he was someone special, and when I later enquired who the visitor was, I was told 'that was Christopher Strachey', as if that were explanation enough.

This sense of awe was not totally unwarranted; as well as being known for his computing prowess, Strachey had connections—through his family—to high places. Burstall (2000, p. 54) remembers:

> In 1967 at the Programming Research Group, David Park picked up the phone. A voice said "This is Buckingham Palace". David laughed, but it was indeed Buckingham Palace, inviting Christopher to have lunch with the Queen. A few months later, at the Copenhagen Summer School, Christopher delighted our three small daughters by telling them about the lunch. They were particularly interested in the fact that the Queen had taken her shoes off under the table. The next day they presented him with a drawing of the occasion. There was a very long table with the Queen sitting at one end wearing a crown, but no shoes, and Christopher sitting at the other end, with shoes and also wearing a crown.

By Autumn 1962, the Cambridge group began to collaborate with Eric Nixon and John N. Buxton of the London University Computer Unit to get the language working on their Atlas machine, a slightly more powerful version of the Cambridge Titan.

This caused a change in nomenclature: CPL now stood for Combined Programming Language. The shifting of names caused Landin to later remark "the names were as much a moving target as the language" (Landin 2001). Stoy agreed with this sentiment, noting at that same talk of Landin's "I was allowed to say [in (Stoy 2000)] that some people thought of it as Christopher's Programming Language—but not say it ever actually stood for that". Indeed, this consideration of CPL as largely Christopher's was somewhat justified. Hartley (2000, p. 70) wrote "Christopher's interests in regularity and the mathematical soundness of the system did actually fit with our aims for perfection, and much that CPL is now credited with came from his deeply philosophical mind".

The main features of CPL, not yet finalised, were published in 1963 and showed a powerful, flexible language (Barron et al. 1963). It was designed to be a fully general-purpose programming language to replace high-level machine code programming. The strong ties to the Titan and Atlas machines allowed the writers to get deep into systems programming as well, much like PL/I's links to System/360. CPL was based intellectually on ALGOL 60, but, as the 'Main Features' paper indicated, "CPL is not just another proposal for the extension of ALGOL 60, but has been designed from first principles and has a logically coherent structure" (Barron et al. 1963, p. 134). This provokes another comparison with PL/I: where CPL was ALGOL-based, so PL/I was based on FORTRAN. PL/I had ideas of looseness and generality in notation compared to CPL's rigorous precision, an important property for Strachey and Wilkes. CPL, like PL/I, had a proliferation of features for the program writer: multiple calling mechanisms for procedures, and the ability to jump out of procedures and into blocks, as well as some new ideas like structures, unions, and pointers (Richards 2000, p. 86). Critically, CPL was an equally ambitious and monumental task as PL/I.

It may seem odd that CPL was such a complicated language, given the proposals set forth by Strachey and Wilkes in their joint paper. Careful reading of that work, however, shows that their concerns were mostly over efficiency and lack of clarity (likely the focuses of Wilkes and Strachey respectively). While CPL was indeed complex, it had plenty of notational clarity to ensure the programmer was doing exactly what was intended (in contrast with PL/I's default options). As an example of this, CPL allowed bit patterns to be written in either octal or binary, the form being indicated by either an 8 or 2 preceding the string; further, although the pattern was assumed to be written right-justified, the programmer could choose to

make the justification explicit by placing the | character either to the left or right of the string (Barron et al. 1963, p. 135).

Wilkes reflected later, in his memoirs, that the project was doomed to failure from the start:

> The fact was that we were all over-ambitious and wholly lacking in experience of large software projects. We did not appreciate how many and how seductive were the opportunities for escalation, and having no insight into the dangers could establish no defences.
> (Wilkes 1985, quoted by Scott 2000, p. 109)

Aiming to define a complex language and still produce an efficient compiler was a real challenge. London's Atlas compiler was being developed traditionally, but the Cambridge effort was to be based on Landin's 'applicative expressions' approach to langauge description (Campbell-Kelly 1985, p. 33). This seriously derailed Strachey, who became more interested in determining the formal semantics of CPL than actually developing a working compiler. The project also lost Barron and in an effort to compensate this, David Park was brought in during summer 1964, as well as graduate student Martin Richards in October 1963—the latter of whom Strachey (1971d) described as "the best programmer I know". However, progress on the CPL compiler was still very slow, and Strachey's contributions diminished further (Campbell-Kelly 1985, p. 34).

The preface to the CPL Working Papers, made in 1966 as a collection of all major project documentation, states:

> The proper description of a programming language is no easy task, and CPL, which is very considerably larger and more sophisticated than ALGOL, presents a formidable problem. None of the authors have been able to spare the time to document the language adequately, and the fact that it has been evolving continuously has not simplified the problem.
> (Buxton et al. 1966, Preface)

Another part of the difficulty was irreconcilable differences between Strachey and Wilkes. Wilkes believed Strachey was no longer interested in the same practical concerns, and wrote in his memoirs

> The fundamental causes of the failure of the CPL project were those I have just indicated[13]. However the immediate cause was my appointment to the staff of Christopher Strachey[...] I liked Strachey personally, but I realized that he was not an easy man to work with, and that I was taking a chance in employing him.
> (Wilkes 1985, quoted by Scott 2000, p. 109).

Barron, the group's nominal leader, remembers how Strachey's interests dominated the Cambridge group and did not lead to the areas desired:

> I was staggered by the idea that Christopher should be working for me, and indeed I might as well have tried to hold a (friendly) tiger by the tail. In fact that is a fairly apt simile. We were supposed to be designing a language and writing a compiler. Christopher set off, and the rest of us were pulled somewhat breathlessly behind. We never did get a compiler, but at the end of it he had formulated some of the fundamental ideas of programming languages, had sown the seed for the work on formal semantics [...] and I gained an understanding of programming languages the depth of which I have only recently come to appreciate.
> (Barron 1975, p. 8)

Someone else who felt he owed a debt of gratitude to Strachey was Rod Burstall, who had first met Strachey through Landin. Burstall and Landin had become friends at Mervyn Pragnell's reading group sessions,[14] and when Burstall was visiting Landin at Strachey's house, he met the older man. Strachey was responsible for Burstall getting employment at Edinburgh University, an institution at which he stayed until his retirement: Donald Michie was looking for a new programmer in 1964 and asked Strachey who he should employ.[15] Strachey recommended Burstall, on the condition that he spend three months working with him first.

Burstall spent this time, October to December 1964, assisting Strachey in London (Burstall 2017). The work they did together was on the semantics of CPL, now occupying Strachey's time most thoroughly. Some of the outputs from this time show Burstall applying early ideas of Strachey's to CPL, and they remain visible in the Strachey archive (Box 271, Folder C.171). These papers show shared

[13] Quoted on Page 239.

[14] See Section 3.2.

[15] Adding colour to this story, Burstall remembers that he accepted Michie's suggestion they meet to talk about employment because Michie offered lunch at a Chinese restaurant: quite a luxury in 1964. Burstall (2000, p. 51) continues: "Thus lured to Edinburgh for further discussion of the proposal, I found myself conducted to the top of a hill, Arthur's Seat, with a beautiful view of the city; I accepted the job halfway down without actually having seen the University."

authorship between the two men, as they worked together, and annotated each others' work. Burstall had not previously considered compiler writing, and to him Strachey presented the task of language definition as a technical problem of both creating a compiler and ensuring it produced the right results (Burstall 1993). Strachey was always keen to remain in touch with the practicalities of computing with his work, perhaps due to his years of work on computer and machine code design. Burstall contrasted this with Landin, from whom he had learnt the basics of functional programming and computation concepts in the Duke of Marlborough pub after the Pragnell sessions, remarking "Strachey wanted implementations whereas Landin worked on paper (or beer mats)" (Burstall 2000, p. 52).

Strachey's interest in language definition was exclusively limited to the semantics; he showed no concern for syntactical considerations, as seen in the following quotation from his paper 'Systems analysis and programming':

> Much of the theoretical work now being done in the field of programming languages is concerned with language syntax. In essence this means the research is concerned not with *what* the language says but with *how* it says it. This approach seems to put almost insuperable barriers in the way of forming new concepts—at least as far as language *meaning* is concerned.[16]
> (Strachey 1966b, p. 124, quoted in Peláez Valdez 1988, p. 196)

Further, Strachey was scathing about the inability of formal syntax descriptions to enable postulation and proof of useful language properties:

> The introduction of a clear and relatively concise notation [BNF] revolutionized the syntactic description of programming languages even without making it possible to prove any properties of the language or programs written in it.
> (Milne and Strachey 1974, p. 33)

This, however, is not entirely accurate; for example, Floyd (1962) was able to prove that a phrase structure grammar did not exist for ALGOL 60. Strachey did acknowledge that BNF had one great advantage: it was well-known across the computing field (Strachey 1973a, p. 1).

So, rather than looking at syntax, Strachey's main motivation lay in bringing mathematical rigour to computation generally and programming languages specifically.

[16] All emphasis from Strachey's original text.

This shows a strong similarity to McCarthy's early motivations; indeed, Strachey referenced these parallel feelings of McCarthy:

> An unsuspecting mathematician who started looking at the theoretical basis of practical computing would be in for a nasty shock. The work that has any degree of mathematical rigour concerns topics, such as automaton theory, which bear no recognizable relation to actual computing. The work which deals with practical computing is at best description and belongs to what John McCarthy calls the "Oh look mummy!" class of research in which the author describes his work without giving any reasoned argument to support his activity.
> (Milne and Strachey 1974, p. 9)

6.3 Towards mathematical semantics

Strachey's thoughts on semantics first came together in his paper at the *Formal Language Description Languages* conference, 'Towards a formal semantics' (Strachey 1966a). In this, he explicitly tied his ideas to the work on CPL and the problems encountered therein:

> The development of basic semantic ideas has taken place alongside of the development of the programming language CPL and a compiler for it. The language is now (July 1964) virtually complete, though there are still some areas for which the semantics have not been satisfactorily formalised. This gives rise to a rather vague feeling of unease, and though we think we know what we mean about such things as lists, error exits from functions, and input-output, we are not altogether happy that we have really got to the bottom of the concepts involved.
> (Strachey 1966a, p. 216)

The base of Strachey's approach was to use functions as a model for programming languages; indeed, documents in the archive (Box 270) show that Strachey was using lambda functions and the Y combinator when sketching semantics for CPL. Later, Strachey wrote of the 1964 paper:

> ['Towards a formal semantics'] introduced the general method of giving mathematical semantics as a set of recursively defined functions from syntactic domains to semantic domains which included higher-order functions.The approach was deliberately informal, and, as subsequent events proved, gravely lacking in rigour—but, in spite of these defects,

> it certainly laid down the outline of our subsequent work."
> (Milne and Strachey 1974, p. 27, quoted in Scott 2000, p. 110).

Stoy (2016b) remarked that, at the time, Strachey was not terribly concerned by this lack of rigour:

> Christopher didn't care. He was using lambda notation to describe functions, programs, which he knew existed. So the fact that not every lambda expression might represent a function didn't bother him. He always claimed he was not a mathematician and was just applying the notation in practice.

For Strachey, the idea of a state of computation was at the core of semantics. This is demonstrated in a comment he made on McCarthy's paper at FLDL, in which he highlighted that taking care of what the state represents was very important to him:

> The only way of analysing semantics for programs is by having the state. But I would like to analyse the state very much more kindly than by calling it a single object like that. I would like to distinguish between the parts of the state which represent the program and the position in the program, the parts of the state which represent the values of the variables, the parts of the state which represent the declarations.
> (Strachey, in a comment on McCarthy 1966, p. 11)

Strachey was not keen on the notion of abstract interpreters used by the Vienna group, as he wrote in 1974:

> Semantic explanations based on even abstract machines are too much like those based on actual compilers to be mathematically satisfying. They always contain a great deal of arbitrary and possibly unnecessary mechanism; a small change in the properties of the language may entail a total reconstruction of its machine and next to nothing can be proved about either the language or the equivalence of two different models each purporting to describe it.
> (Milne and Strachey 1974, p. 11)

Strachey was also uninterested in using concrete textual approaches like van Wijngaarden's to describe programming langauges, baldly stating (in Walk 1969b, p. 3) "using string manipulation as a basis does not give insight into the language".[17]

[17] Stoy's textbook echoed this view, as he argued:

However Strachey was not unfamiliar with string manipulation techniques: another piece of work that came out of CPL was a General Purpose Macrogenerator, known as GPM (Strachey 1965a). This was published in 1965 in *The Computer Journal* of the British Computer Society, of which Strachey was a distinguished member.

GPM was a text manipulation system that allowed easy placement of repeated strings within text and was used for developing the compiler. Richards (2000, p. 87) remarked on its power: it was Turing complete, yet implementable in only 200 instructions on the Titan computer. The GPM became quite popular and was widely implemented on many machines, including by Strachey himself, but by 1971 he had grown distant from it, writing "It gives an elegant example of the use of a stack and shows the power of functional composition to construct complicated operations from simple primitives. It is not, however, of any very deep theoretical interest and I now regard it as a rather beguiling time-waster".[18] (Strachey 1971a)

So, rather than an interpreter, or a symbol manipulation strategy, Strachey preferred what he called a "mathematical" approach to language description. This he likened to the use of abstraction in high-level programming languages. A mathematical method for writing programs uses the mathematical notions of concepts such as sine and addition, allowing the programmer to ignore the precise machine steps used to achieve these actions. He wrote:

> We are chiefly interested in the *values* of the expressions and not in the

> We can apparently get quite a long way expounding the properties of a language with purely syntactic rules and transformations [...] But we must remember that when working like this all we are doing is manipulating symbols—we have *no idea at all* of what we are talking about. To solve any real problem, we must give some semantic interpretation.
> (Stoy 1977, p. 9)

This is similar to the point of view espoused by philosophers such as Searle (1980), in his famous Chinese Room thought experiment; he argued that merely being able to transform texts based on rules does not constitute understanding.

[18] Strachey had created another such fun text replacement program to create love letters in the mid 1950s. At IFIP's first Congress in 1959 he described it to a journalist, who explained:

> A few years ago as a stunt Mr. Strachey had one of his machines write love letters for him. He fed in the basic sentences, left out the adjectives, nouns, adverbs, and verbs, and then fed in words from Roget's "Thesaurus", which the machine could pick out at random. For example, he fed in My — — (adjective, noun) — — (adverb verb) your — — (adjective, noun).
>
> "It was quite all right," he said. "But, unfortunately, I had included the word 'erotic' and it made all the letters rather erotic."
> (Buchwald 1959)

This is another lovely example of Strachey's playful side.

> steps by which they are obtained. The alternative, earlier approach, which might be called 'operational', involves specifying in detail the sequence of steps by which the result can be obtained. While the ability to do this is also an important facet of computing, it should be regarded as a means to an end; the important thing is to compute the correct quantity.
> (Strachey 1973a, p. 2)

Strachey wanted to achieve this in language description as well, and for him, this involved the use of functions, represented as lambda expressions. His respect for functions can be seen in the following remark to van Wijngaarden at FLDL, who had, in his method, processed out all of the functions from his programs:

> The last thing I would want to do is remove a function (or, at least, what I call a function) because it seems a much better-understood mathematical entity than a procedure—which I call a routine—which is a complicated command. This is the direction we would like to go in—looking at the basic ideas that underly programs.
> (Strachey, in van Wijngaarden 1966b, p. 23)

The notion of considering a procedure as a function was not uniquely invented by Strachey. Bill Burge had proposed the idea at the *Working Conference on Mechanical Language Structures* in 1963 (Gorn 1964). In the same discussion, McCarthy said "What is meant by the semantics of a program is this function taking an old state vector into a new one".[19] McCarthy (quoted in discussion of 1966, p. 11) also agreed with this general idea, saying "It's the action that occurs going from the initial to the final state which is the meaning we all have in mind".

Strachey did acknowledge that the use of functions in programming language description was not his alone:

> The λ-calculus has been widely used as an aid in examining the semantics of programming languages precisely because it brings out very clearly the connection between a name and the entity it denotes, even in cases where the same name is used for more than one purpose.
> (Strachey, quoted by Scott 2000, p. 107).

The crucial difference in Strachey's approach is just how seriously he took the notion of functions as the core of semantics.

[19] Meyer (2011) also commented on the similarity between Strachey's and McCarthy's focus on functions

This was manifest in his desire to use functions to model not only the descriptive aspects of programming languages (expressions, declarations, and functions) but also the imperative features (assignments, jumps, and sequencing). All of these Strachey wanted to describe in a purely applicative framework. This contrasts with the work of Landin, who introduced a mechanism for evaluation, the SECD machine, to handle the imperative aspects. Strachey noted this choice of descriptive base affected how easy particular ideas were to model:

> What is 'difficult' very much depends on the frame-work of thinking. For example, assignment is difficult in the λ-calculus approach, recursion is difficult in other systems. But both occur in programming languages and are simple to use.
> (Strachey, in Walk 1969a, p. 9)

Strachey believed assignments were the core feature of programming languages, as they change the values of variables. Further, he noted that in programming languages, 'variables' become truly variable for the first time, which is to say they change over time; in contrast, in standard mathematics "a variable is really another name for a constant whose value we do not happen to know at present" (Strachey 1966a, p. 201). This fundamental property of programming languages was termed by Strachey 'referential opacity', and a key property of his approach to semantics was that in the metalanguage, all variables would have their 'referential transparency' retained.

The importance of treating names properly was a point of significance to Strachey. At the *Programming Languages and Pragmatics* conference in August 1965, he remarked:

> I note [...] that people have a rare ambivalent attitude about what a name means or what a variable means, and if you pin them down they will shift a little from saying well a name is an address or a name is not a variable, and it is very difficult indeed to find out precisely what they mean. Generally I think a great many of the things that some people refer to as names are in fact descriptions, not names at all.
>
> What I am trying to say is that what we ought to describe as names are complex entities. They are descriptions and they are subject to evaluation at various stages in the process between our thinking of them first and writing them down on paper, and before they are placed in context or bound.
> (Strachey, in Woodger and Green 1966, p. 223)

Strachey's way of treating names linked with his approach to assignments, which was to realise they are split into two parts: that to the left of the assignment symbol, and that to the right. Expressions could be present on both sides; for example, array indexing or conditional expressions would be situated on the left. This gave rise to his concept of *L-values* and *R-values*, first introduced in 'Towards a Formal Semantics' (Strachey 1966a). L-values ultimately represented positions in a store,[20] and R-values a value in the store.

The store was regarded as a partial function associating L-values with R-values, and performed a similar role to the 'denotation directory' in VDL.[21] Strachey referred to stores with the letter σ. Reflecting later on the history of the store function, Strachey wrote:

> The store, σ, is a function from locations to their contents, following a suggestion made by Rod Burstall to [Strachey] in 1964. Before then we had made no detailed model of the store and had merely assumed the existence of certain basic functions operating on it. The construction of a proper model helped to bring out the fact that storing commands (or procedures) led to severe mathematical problems.
> (Milne and Strachey 1974, pp. 27–8)

Commands, particularly assignment commands, ultimately have the result of changing the store, and so Strachey's approach was to treat them as functions $(\sigma \to \sigma)$. As each command in a sequence modifies the store produced by the previous command, sequences of commands were modelled as a composition of their function representations. A program as a whole is one function formed by the composition of all of its statements' representations (in early works, Strachey talked about the metalanguage 'representation' of statements rather than the later 'denotation').

Complications to this method were introduced by block structuring and procedures, which Strachey addressed in 'Towards a Formal Semantics' by two functions *Copy* and *Free*. These *introduced* new L- and R-values to cope with parameters and new block declarations and *removed* them, respectively. Loops and jumps also required more careful treatment, for which Strachey used Curry's 'paradoxical' Y combinator, as Landin had. This was used as the 'fixed point' operator, and even in the earliest works, Strachey noted that such a combinator might not be implementable

[20] Stoy noted in his textbook on denotational semantics "we prefer the less anthropomorphic English term 'store' rather than the American 'memory' " (Stoy 1977, p. 10).

[21] See Section 5.5.

and might not even exist; but he referred to Landin's work in showing a representation for it that could be utilised (Landin 1964).

As well as presenting this technical material, Strachey also engaged vigorously in the FLDL discussion sessions, making comments on many other papers. He was known to enjoy speaking about his ideas more than writing them down. Strachey clearly enjoyed the experience of the conference; he wrote in a letter to Michie at the end of that same September "Vienna was very interesting, though hard work, and on the whole encouraging" (Strachey 1964a). He also noted in the same letter that he felt that Europe was ahead of the US in terms of such theoretical work, echoing the comments of McIlroy (2018).

As Strachey moved away from working on CPL, his consultancy work was also drying up. He began to look for a permanent home as a researcher, competing for a chair of computing at Imperial College, London, and investigating the possibility of setting up a research group in Cambridge. Eventually, with the help of Fox, and spurred by the success of their summer school, Strachey secured a grant from the Department of Scientific and Industrial Research[22] to set up a group at Oxford University (Campbell-Kelly 1985, p. 34). This was called the Programming Research Group, or PRG, and was intended to focus on non-numerical computing applications, as was Strachey's interest.

Although the grant started from July 1965, Strachey spent the first months of the PRG time at the Massachusetts Institute of Technology's (MIT) Project MAC, their computer centre. This visit had been set up when Strachey visited in May 1965 to see their time-sharing system, a trip on which he had been joined by Martin Richards. During the longer visit, Strachey gave a course on "Mathematical Theory of Programming Languages" (Strachey 1965b), covering the concepts that interested him at the time, as also presented in his FLDL paper (Strachey 1966a). The course remained on the curriculum at MIT after Strachey left, being taught by Richards in his stead (Campbell-Kelly 1985, p. 35). Strachey found the experience valuable for developing his ideas on the topic, as revealed in a letter to Lord Halsbury (Strachey 1965c):

> I am finding my time at MIT personally very stimulating. I have been holding seminars on the mathematical theory of programming langauges

[22] Precursor to the Science Research Council, now Engineering and Physical Sciences Research Council.

> and, as a result, partly of the discussions in the seminars, and partly of the necessity of preparing them, I have been able to clarify a number of points which were unsatisfactorily vague in my mind before.

When Strachey returned to Oxford in April 1966, the PRG opened properly. Originally housed in 43 Banbury Road, it soon moved next door into larger premises at 45 as the group began to expand, a location in which it remained until well after Strachey's death. See Figure 6.5 for a photograph of the building, which was something of a rabbit warren of small rooms and corridors (McIlroy 2018). David Park was the first employee at the PRG, having moved from Cambridge to work with Strachey. Their first major work was closing the chapter on CPL: Strachey gathered together the group of workers to put out a final version of a report on the language. However, no-one but Strachey and Park contributed, frustrated by how difficult it had been trying to finalise the writing while Strachey was at MIT (Campbell-Kelly 1985, p. 35). Eventually, a collection of reports called the 'CPL Working Papers' was produced but, like much of Strachey's work, was not formally published, instead being circulated privately (Buxton et al. 1966).[23]

Richards, however, produced for his doctoral thesis an implementation of a CPL subset, called Basic CPL or BCPL, which was smaller and less demanding on the system. This was implemented by Richards at MIT in December 1966, and became a systems implementation language (Richards 2000, p. 86). Eventually, a retooled but conceptually similar core language was used to write the early versions of UNIX; Richie and Thompson called this 'B' as it was similar to BCPL but smaller. The next version of the language was then called C.[24] BCPL excited Strachey greatly and he wrote that it "marked a revolution in the way in which it was possible to write complicated software systems." (Strachey 1971d).

Strachey's own focus was now on writing up a set of lecture notes called 'Fundamental concepts in programming languages'. The ideas came from his FLDL paper, developed and expanded first at MIT and then as a series of lectures for the Summer School in Computer Programming held in Copenhagen in August 1967. The papers sat on the editor's desk for two years before proceedings publication was eventually scrapped; but, like many of Strachey's works, circulated privately for a long time

[23] The papers were not wholly complete; Stoy (2016b) recalls "the preface says 'Some sections are missing,'—a typical Christopher phrase—'merely because they have not been written' "

[24] Stoy (2016b) adds that the group at the PRG, following this development, wondered whether C's successor would be called D, following the alphabet trend, or P, as that was the next letter in 'CPL'.

Figure 6.5: 45 Banbury Road, Oxford, the location of the Programming Research Group.

and were rather influential. The work finally saw publication in 2000 in a special issue of *Higher-order and Symbolic Computation* dedicated to Strachey (Strachey 2000). In an introduction to the piece, Mosses (2000, p. 7) wrote "It is, however, one of Strachey's most significant and lengthy papers; widely circulated in the original typescript version, it has also been highly influential".

In this work, Strachey argued that computing was at a stage of infancy and confusion, lacking in proper rigour, and crying out for careful mathematical treatment:

> In computing we have what I believe to be a new field of mathematics which is at least as important as that opened up by the discovery (or should it be invention?) of calculus. We are still intellectually at the stage that calculus was at when it was called the 'Method of Fluxions' and everyone was arguing about how big a differential was.
> (Strachey 1967, p. 13)[25]

Here we see Strachey situating his work as fundamental to the core of computation; like McCarthy (1963) referencing the works of Kepler and Newton, Strachey was keen to emphasise the connection to mathematics. He continued to explain that the mathematisation of computing must be done carefully:

> We need to develop our insight into computing processes and to recognise and isolate the central concepts—things analogous to the concepts of continuity and convergence in analysis. To do this we must become familiar with them and give them names even before we are really satisfied that we have described them precisely. If we attempt to formalise our ideas before we have really sorted out the important concepts the result, though possibly rigorous, is of very little value—indeed it may well do more harm than good by making it harder to discover the really important concepts. Our motto should be 'No axiomatisation without insight'.
> (Strachey 1967, p. 13)

'Fundamental Concepts' expanded on the ideas discussed in 'Towards a formal semantics', although he acknowledged the work was still "far from complete and [could not] yet be regarded as adequate" (Strachey 1967, p. 47). Again, he used CPL as an illustrative language. The paper did not represent a radical shift from the 1964 ideas, with the notions of L- and R-values, store, functions $(\sigma \to \sigma)$, and the Y combinator remaining at the core. The last aside, all the ideas were expanded upon,

[25] Page references will be made to the republished paper as it is considerably more accessible; year citation is kept to the original paper.

and the paper also included a lengthy treatment of data structures and how they could be represented in Strachey's approach.

One point worth mentioning is that in the context of determining the value of expressions, Strachey showed more obvious influence from Landin, using his notion that 'let', 'where', and lambda abstraction are equivalent. This Strachey referred to as the environment in which expressions are evaluated. Later in the paper, Strachey introduced commands and a store for modelling them, but this was as a replacement for the environment rather than an addition, as it would be later in the denotational approach (and as the Vienna Group had used with their combination of environment and denotation directory).

Another important extension in this paper was the idea of higher-order functions, those which can be the arguments or outputs of other functions. Strachey objected to the omission of such functions in ALGOL 60, writing "Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression" (Strachey 1967, p. 32). Strachey was very keen to allow first class functions: as well as permitting higher-order functions, this would also allow functions to be values of variables, be assignable, and be the result of expressions. Strachey pointed out that higher-order functions are inherent in the use of lambda calculus, and indeed it was that property which required the use of Landin's closures.

A final note worth making about 'Fundamental Concepts' was pointed out by Mosses in his introduction: Strachey's way of describing the meaning of lambda expressions was somewhat operational (Mosses 2000, p. 8). This was likely due to the lingering influence of Landin and because Strachey had at that time no other base to give lambda calculus. He even noted that he thought the use of a machine was inevitable in describing programming languages, although it would be based on lambda calculus:

> The ultimative [sic] machine required (and all methods of describing semantics come to a machine ultimately) is in no way specialised. Its only requirement is that it should be able to evaluate pure λ-expressions. It achieves this result by explicitly bringing in the store of the computer in an abstract form, an operation which brings with it the unexpected bonus of being able to distinguish explicitly between manifest and latent properties.
> (Strachey 1967, p. 48)

That said, Strachey did not focus at all on the basis of the description, which is to say the interpretation of the λ expressions. He noted that it could be achieved with reduction rules, an essentially operational approach; all that was required is that the sequencing was correct, which was performed by the functional composition of the representations.

From Autumn 1967, Strachey became a 'Reader in Computation' at Oxford University (Strachey 1971a), and the PRG began attracting graduate students. This was part of Strachey's hope for the PRG as a whole:

> My chief aim here has been to introduce a degree of scientific and intellectual discipline into the subject of computing which seems to have grown in to an enormous sprawling mass of *ad hoc* tricks where fashion and state-of-the-art have taken the place of anything more solid. [...]
>
> The graduate school here has become one of the most satisfying features of the Programming Research Group from my point of view. We have managed to attract a small but continuing stream of really able mathematicians with the result that they have generated a highly critical and thoughtful atmosphere in which *ad hoc* or superficial ideas are given very short shrift.
> (Strachey 1971a)

Strachey was not, however, interested in developing an undergraduate computing course, and was against the idea of 'Computer Science', as evidenced by his paper 'Is Computing Science?', in which the answer is a (qualified) 'no' (Strachey 1970a).

Scott (2000, p. 105) wrote:

> On the other hand, we should note that Strachey was quite opposed to 'Computing Science' as an undergraduate specialist course of studies in itself (that is, according to the British pattern, where a student would study nothing else at the university). At that age students—he felt—should learn serious things that they will need later, rather than the latest 'state-of-the-art' mish-mash which will be out of date before they graduate.

This was typical of Strachey's lack of concern for what he saw as the surface level aspects of computing, and his desire to emphasise the deeper theoretical and technical points which would cross the "state-of-the-art mish mash". Many of the PRG's graduate students also shared Strachey's aims to produce both theoretical and practical work. Stoy (2016b) noted that many made significant contributions in both areas.

The early years of the PRG were tenuous and its future uncertain. Oxford University made no commitments to keep Strachey on should the SRC funding run out. Park made the decision to leave for a tenured post at the University of Warwick in 1968 (Campbell-Kelly 1985, p. 36), although he remained interested in the PRG work. This is demonstrated in a letter he wrote to Scott (Park 1970) discussing in some depth technical aspects of the semantics work; he also gave lectures on the topic at Edinburgh. Strachey was disappointed to have lost Park, and held him in great esteem, writing "I have a very high regard for his abilities; I was instrumental in getting him to come to Oxford with me in 1966 and very sorry when he decided to move to Warwick. In his own particular line I know of no one in the world who is his equal." (Strachey 1970b).

The PRG's situation began to improve towards the end of the 1960s, and Strachey also found another engagement for his time: he became involved in the newly created Wolfson College in 1969, remaining deeply connected with the college throughout his life. Many of the notes and sketches of ideas in the Strachey archive are even written on the backs of Wolfson management meeting minutes. Strachey also took accommodation in the first buildings of Wolfson College at 60 Banbury Road, where he would host musical evenings. He would play one of his two grand pianos, and inveigled guests, like Stoy, into playing the supporting parts on the other. Stoy (2016b) also remembers Strachey singing Victorian love songs in an amused, satirical manner.

At this time, Strachey "adopted some of the more liberal attitudes of the late 1960s: he dressed casually, wore his hair long, and although his manner was never outlandish his homosexuality became more overt" (Campbell-Kelly 1985, p. 36). A photograph of Strachey in 1969 is shown in Figure 6.6. Penrose also remembered that Strachey, although rarely directly involved in politics, was certainly progressive in his beliefs: at a dinner party in the late 1950s or early 1960s "he remonstrated that women were undervalued and underpaid in the society of the day. I remember the gathering of highly forward-thinking men reacting with some incredulity, asserting that such worries were surely now outdated" (Penrose 2000, p. 84). This liberal attitude, no doubt learnt from his suffragist mother Ray, had been visible even in his schoolmaster days, as Jackson (2000, pp. 73–4) recalls:

> He encouraged us to be sceptical of the school's adulation of athletic prowess, and to disdain the compulsory School Cadet Force, where we paraded each week in army uniforms and boots, carrying Lee–Enfield rifles made in 1916. He shocked our conformist prejudices with scan-

Figure 6.6: Christopher Strachey at the 1969 second NATO Software Engineering conference in Rome.

> dalous tales of his father Oliver and his great-uncle Lytton Strachey,[26] and other members of the Bloomsbury set.
>
> Everything he taught us may seem commonplace in this age of new maths, computer literacy, and social freedom. But nearly fifty years later I remember clearly how it seemed to me then as an adolescent boy. It was an enlightenment and an intellectual liberation.

In 1967, Strachey's "long-time assistant and coworker" (Scott 2000, p. 104) Joe Stoy joined the PRG. Stoy's contributions were mostly as a teacher and assistant; he said "I don't claim to be a great producer of new stuff in [denotational semantics]; more an expositor" (Stoy 2016b). Stoy's background was as a graduate student in physics at Balliol College, Oxford, where he began using the university's KDF-9 computer

[26] Note that Jackson is slightly mistaken about this relationship: in fact Lytton was Christopher's uncle.

for spectroscopy. Stoy had experience in computing having done a brief undergraduate course on the subject, and was using ALGOL 60 for his graduate work (Stoy 2016b). He got to know Joyce Clark, the wife of the university's computer engineer, through his spectroscopy work; she was a programmer at the PRG and invited him to a party thrown to celebrate the move to 45 Banbury Road. That party is where Stoy first met Strachey.

This period also marked the acquisition of the PRG's own computer, a CTL Modular One. That was to prove essential to the PRG's work, as before then, the computing laboratory under Fox was the gatekeeper to computer work and could be quite interfering. Stoy (2016b) remembers:

> To get permission to use the university computer you had to discuss your problem with a numerical analyst; it was essential, they had to sign off on it, because all programs would contain some numerical analysis, and it was important not to go wrong. For example, one of Fox's things was if you came up to him or to any of the team and said "I want to invert a matrix, please tell me the best way to invert a matrix," the usual reply was "No you don't. You're trying to do something like solving a set of algebraic equations, which would be much better and more accurately solved without actually producing the inverse of the matrix." So they would be giving that kind of advice.

Strachey and Stoy first worked together on developing ideas for an operating system for the Modular One; Stoy remembers he wrote a prototype in ALGOL 60 which impressed Strachey (Stoy 2016b). The first version, OS1, was coded in BCPL, imported from Richards' MIT compiler[27] (Richards 2000, p. 88). This was tested on the computing laboratory's KDF-9 prior to the delivery of the Modular One, and when that computer arrived in March 1969 it took only 45 hours to get the system up and running (Campbell-Kelly 1985, p. 36). The stages of the operating system were developed incrementally up to OS6, at which point reports were published in 1972 (Stoy and Strachey 1972a; Stoy and Strachey 1972b). McIlroy noted that the operating system had quite a few similarities to early UNIX: the file system in particular was exactly the same (McIlroy 2018).

[27] Stoy (2016b) remembers: "there was a stage where the punched cards of the BCPL compiler OCODE, the semi-compiled BCPL compiler, came across the Atlantic. They got misaddressed, so there was quite a delay in delivery, and Christopher said when we finally got them, they smelt rather salty."

The PRG OS was written entirely in the high-level BCPL, with an interpretative code developed by Strachey himself, something that pleased him greatly:

> Christopher used to claim proudly that he was the only person in the group that knew the Modular One machine code: everyone else used the interpretive code that he had written to make the machine look like a stack machine. 'Since nobody knows how fast the machine really is,' he would say, 'no one can object to a program on grounds of inefficiency.' (Barron 1975, p. 9)

Doug McIlroy, who has already been mentioned a few times, was a visitor at the PRG from September 1967 to June 1968. He got a leave of absence from Bell Labs, who continued to pay his salary on the condition that Oxford also paid something. He remembered that Strachey managed to get a pay of £3 per day: almost enough to live on (McIlroy 2018). Stoy (2016b) believed he had "wanted to learn about semantics by drinking from the source". During his time in Oxford, McIlroy worked on an idea that ultimately became the famous UNIX pipe, written in a note 'Co-routines: semantics in search of a syntax' (McIlroy 1968). He also did some work with Stoy on computer graphics, including a 3D projection for a book being written by a Wolfson fellow (McIlroy 2018). Interestingly, McIlroy does not remember spending any time looking at the PRG operating system, despite the professed similarities with UNIX once he got back.

A timeline, written by Strachey, of this period at the PRG, can be seen in Figure 6.7.

1969 marked the beginning of the most important part of Strachey's career, in his view, and the most important to the present work. This was his collaboration with Dana Scott.

6.4 Codifying the foundations

Dana S. Scott (b. 1932) is a logician whose undergraduate degree at Berkley taught him about Tarski's semantics of logic, as well as lambda calculus. Scott's doctoral supervisor was Church himself, at Princeton; and he received tutelage from Tarski as well. This combination of knowledge of both lambda calculus and universal algebra gave him an almost unique set of skills in the computing field, and gave him just the information he needed to work on the foundations of computing (Scott 2016). Scott's work with Michael Rabin in 1959 on automata theory won them the Turing

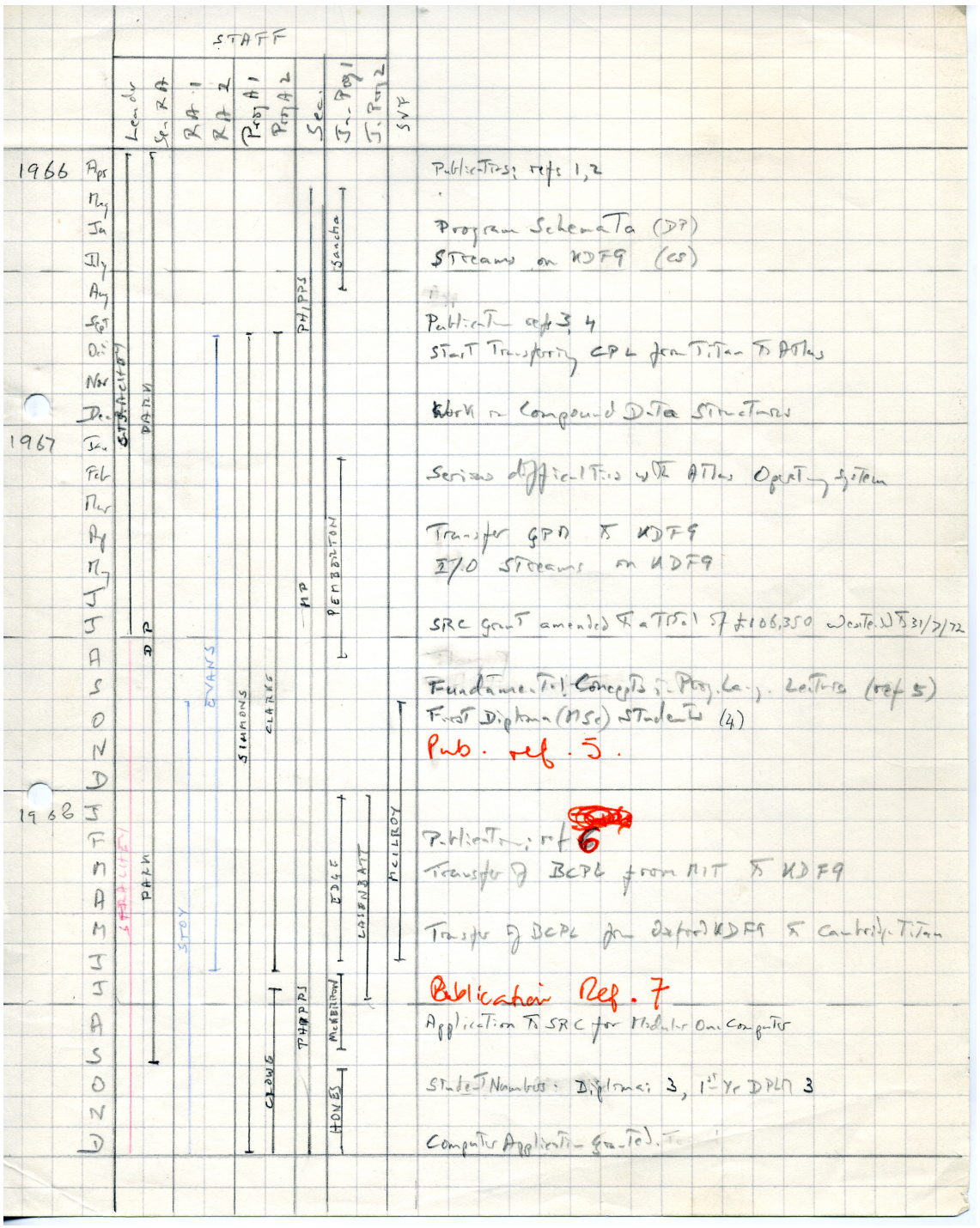


Figure 6.7: A chart of employees at the PRG, written in Strachey's distinctive hand. The right hand column marks notable events, such as grants awarded, and important project milestones.

Award in 1976; however, he moved away from this field because there seemed to him to be "excessive complexity" and no obvious applications (Scott 1977, p. 636).

Scott got his first experiences with computers as a student at Princeton and his introduction to ALGOL in the 1960s led him to begin thinking about programming languages, particularly the link between logic-computable functions in theory and computability of funcations in practice (Scott 2000). However, he had not much direct contact with using computers, writing in 1977 "I have no practical experience in present-day programming; by necessity I have had to confine myself to speculative programming, gaining what knowledge I could at second hand" (Scott 1977, p. 635).

This interest in programming led Scott to enter into collaboration with Jaco de Bakker during 1968 and 1969 at Mathematisch Centrum, his first sabbatical in Amsterdam. De Bakker had been a student under van Wijngaarden and his thesis was on the application to ALGOL 60 of van Wijngaarden's processor approach (de Bakker 1965). This work also contained a criticism of Strachey's early approach and indeed the use of lambda calculus generally, which is that the use of a function notation "conflicts with the dynamic structure of a program, which consists of a number of instructions executed successively". However, de Bakker did acknowledge that lambda calculus "can be used to describe the locality concept of ALGOL 60 in a way which is more elegant than in any other system we know of", though it struggles with assignment and jumps (de Bakker 1965, pp. 2–3). In contrast, de Bakker's approach modelled assignment very well, but had difficulty with locality and jumps (de Bakker 1965, p. 9).

An interesting aspect of de Bakker's work is that the processor and its metalanguage were defined by ALGOL 60 programs. The definition of ALGOL 60 itself was given in terms of the metalanguage and processor. This means, of course, that there is something of a loop: a reader could not come to the definition with no knowledge of ALGOL 60 and emerge understanding it. However, if the reader has some informal knowledge, they can understand the processor to an extent, read the description of ALGOL 60, bettering their knowledge of the language, whereupon a second reading of the processor would become more sound. Eventually, de Bakker hoped, the process would converge, and the reader would then understand ALGOL 60 very well (de Bakker 1965, pp. 7–8). This addresses in a rather unique way the problems levelled at McCarthy at FLDL of introducing a new language as the metalanguage and the requirement of understanding that too.

By the end of the 1960s, however, de Bakker was coming to realise the limitations of this approach, and he worked with Scott on what they called 'A Theory of Programs'. The approach built on Scott's work on automata theory, with the addition of input and output (Scott 2016). This was first presented as a seminar at the Vienna Laboratory in August 1969 (de Bakker and Scott 1969). As with many other pieces described in the present chapter, this was not formally published, eventually appearing in a Festschrift for de Bakker (Klop, Meijer, and Rutten 1989). The publication was not as a typescript, however: Scott's handwriting is beautifully clear enough that it was reprinted as a manuscript.

From his work with de Bakker, Scott was invited to be a founding member of IFIP's new Working Group 2.2, on Formal Language Description Languages, set up following the success of the conference with the same name , under the initial chairing of Tom Steel. Scott did not attend any of the meetings until the third, unlike Strachey, also a founding member (Zemanek 1967a), who had attended from the first meeting (Walk 1967a). Scott and Strachey met at the third meeting, held in April 1969 in Vienna. The two were both in a sub-group on language concepts, and Scott showed his interest in using more formal approaches (Walk 1969a, p. 14). Also at this meeting, Scott presented a long list of questions for the field, organised into categories (Walk 1969a, pp. 38–40). The list is summarised below:

Substance Is there something to discuss independent of language?
Semantics Is there a definition of computation?
Deduction Can the result of understanding programs be formalised?
Description Can the systems be communicated?
Design Can the languages be improved?

There were also many sub-questions in each category, showing Scott clearly engaged with the field and having a very thorough understanding of its concerns. Strachey took very detailed notes of this presentation, which can be seen in the archive (Box 287, E.40); he was clearly impressed with Scott's engagement.

Scott (2000, p. 103) later wrote that he was "struck by Strachey's striving to isolate clear-cut general principles" and found his approach "the most sympathetic of the members of the group". This particularly was in contrast with the approaches of the ALGOL 68 definition and the Vienna group (Walk 1969a, p. 2). Scott was interested in semantics as a way to think about computations in a more abstract way than via their implementations, and as a path to program verification, later saying "How

Figure 6.8: Dana S. Scott in 1970.

can you verify a program if you don't have any semantics for the program?" (Scott 1994). In September 1969, just before joining Strachey, Scott (in Walk 1969b, p. 19) said of semantics "The aim is to develop a theory for correctness, equivalence, and termination, for a suitably rich language involving assignment, recursive procedures, and call by value".

A photograph of Scott at this time can be seen in Figure 6.8.

Scott was invited to the Vienna Laboratory in August 1969 to help the group understand Floyd's work on flowcharts and assertions (Floyd 1967b). However, Scott spent the time talking instead about his work with de Bakker, which began to plant seeds in the Vienna Lab that later flowered into a different direction for their research (Astarte and Jones 2018, p. 109).

A letter from Strachey to Scott in September 1969 shows the tail end of a "stimulating" conversation (Strachey 1969). Scott had been proposing a method for language definition based on his work with de Bakker which Strachey did not particularly like. His criticism was that, unlike lambda calculus derivations, Scott's approach would likely struggle with the heavy use of functions and recursion that Strachey saw as critical. However, despite this—and as a result of the immediate gelling the two experienced—Strachey invited Scott to Oxford for the Michaelmas term of 1969 (October 1969–January 1970) and the letter shows the final arrangements for that: after the WG 2.2 meeting in Essex, the two would travel to Oxford together.

Work began immediately on Strachey's formal semantics ideas, a period of intensive activity that Scott found exhausting and satisfying in equal measures, as he later recalled:

> That term was one of feverish activity for me; indeed, for several days, I felt as though I had some kind of real brain fever. The collaboration with Strachey in those few weeks was one of the best experiences in my professional life.
> (Scott 1977, p. 637)

Joint seminars with all PRG members invited took place weekly on Wednesday afternoons, and could last many hours; Stoy (2016b) remembers one occasion in particular when an exhausted Scott melodramatically clasped his forehead and exclaimed "Oh, I'm so tired!". He added that the seminars often went on well into the evenings and ended in the pub.

Scott's main goal was to provide a sound mathematical foundation for Strachey's semantics by determining a way to define the domains for the functions which would represent commands. Initially, Scott thought there was no way to define a type-free lambda calculus model; as Strachey (1973a, pp. 6–7) pointed out, the issue was related to the well-known cardinality problem:

> If we allow atoms as well as λ-expressions every object must be either an atom or a function (as every λ-expression is considered to be a function). This leads to the defining equation.
>
> $$D = A + [D \to D]$$
>
> [...] If D has k members, the full set $D \to D$ has k^k members and, as Cantor's theorem shows, this is always greater than k provided k is greater than one.

This obviously breaks the equality, and there is a fundamental contradiction.

Initially, Scott (Scott 1969, republished in 1993) presented a typed lambda calculus model. The idea, as Stoy (2016b) describes, "made heavy use of explicit representation and derepresentation functions to map functions into the domain space", and was rather mathematically unsatisfying and awkward.

However, after a moment of sudden inspiration, Scott found an untyped solution:[28]

> Then, one Saturday morning in November of 1969, when lying on the bed in the guest room of the flat that I and my family had rented in Headington, an awful thought occurred to me. In setting up the higher types of continuous functionals, I had seen that each of the type domains had a *basis of finite elements* —in the sense that each object of each type is the least upper bound of the finite elements it contains. In passing from one type level to the level of continuous functions above it, the basis of finite elements usually became *more complicated*. But what if, I suddenly thought, there were a type D such that the basis of the function space $(D \to D)$ was *no more* complicated than the basis of D? Could we then have a type $D = (D \to D)$ in the sense of an isomorphism of partially ordered sets? In a fever I set about finding such a domain over the next few days, and that was how Domain Theory was born in my mind. After being so critical of the formal approach to λ-calculus, I would now have to eat my own words, because the λ-calculus could indeed have *mathematical* models.
> (Scott 2016, p. 13)

The partial ordering mentioned is based on the notion of information in a function, which is to say the size of its input or output domain. For example, the function $\lambda x \cdot 0$ (i.e. one which returns 0 no matter the input) has less information than $\lambda x \cdot x$ (the identity function) because the first has an output domain of 0 but the second has (assuming $\mathbb{N}$ input) $\mathbb{N}$ as its output, a considerably larger set. Scott published the results in a series of foundational papers, written initially as PRG monographs (Scott 1970; Scott 1971b; Scott 1971a, 1973).

Tennent (quoted in 1976, p. 442) later wrote of the idea:

> The generalization from computable to continuous functions is much like the generalization from algebraic to real numbers. In both cases one moves from a small but subtle set, determined by a certain kind of finite implicit representation, to a larger but structurally simpler set which can be constructed by limiting processes.

[28] All emphasis in this quotation is original.

Scott (2016, p. 14) also remarked that due to his knowledge of work by other logicians in related areas, he actually had all the pieces to put together the model for the lambda calculus in 1957, and that he deeply regretted it took him so long to make the connections. Indeed, before starting work with Strachey and formalising the domains, Scott had already worked out a method for finding fixed points of functions[29] (reported in Walk 1969b, pp. 34–5). However, this was based on a definition by abstract machine and did not have the pure mathematical basis of the solution Scott developed in Oxford.

The importance of Scott's work for denotational semantics is that it allowed the basis of the semantics to be purely formed from mathematics and logic. As Strachey had pointed out in the conclusions of 'Fundamental Concepts', using reduction rules as the semantics of lambda calculus meant an essentially interpretive base; however, Scott's logic and domains meant that the basis could instead be a set of induction principles. This Stoy (2016b) described as "what Christopher meant by the basis of a mathematical definition". Crucially, this basis provided necessary abstraction and enabled proofs to be made without resorting to induction over steps of interpretation—as well as solving foundational problems like the use of the Y combinator. These were important requirements for Strachey, who wrote:

> The mathematical entities concerned must be properly defined and the operations on them precisely specified. There is no room for discussing a 'state vector', for instance, without specifying its components completely;[30] we must not assume that we can apply a fixed point operator to functions until its existence has been proved or use Curry's 'paradoxical combinator', Y, as a fixed point operator at all until the validity of applying a function to itself has been demonstrated; structural inductions are unsatisfactory without some assurance that they do not involve circularity. All this may be commonplace in mathematical research papers but it is a rarity in computing.
> (Milne and Strachey 1974, p. 13)

Strachey was grateful for Scott's hard work, as it gave a good deal of respectability to his semantics:

[29] Bekič (republished in 1984) and Park (1969) had also—independently—developed methods for finding fixed points of mutually recursive functions. Interestingly, Bekič had developed his approach whilst in London working with Peter Landin (Bekič 1984b, p. XXI)

[30] This is clearly a specific reference to McCarthy's approach; it mirrors Strachey's comments to McCarthy at FLDL in which he declared he wanted to be able to treat the components of the store carefully; see Section 3.1.

> We have been at considerable pains to introduce and maintain a mathematically acceptable degree of rigour in our theory. The rigorous mathematical basis on which our theory is founded is due to the work of Dana Scott; without this work the present essay could not have been written. (Milne and Strachey 1974, p. 15)

Historian Mike Mahoney (2002) noted that the field of computing provides applications for certain areas of mathematics which had previously been considered of little practical use, and that Scott's use of lattice theory was a good example of this. Gordon (2000, p. 65) added that the synthesis of theory and practice shown in this work was of great interest to the computing community:

> This exhilarating and elegant combination of programming language theory (Strachey) and sophisticated mathematics (Scott) generated tremendous excitement and was seen as a major breakthrough and a very hot topic for research.

However, some criticised the complex technical nature of the mathematics. McCarthy clearly understood and engaged with the topic—as evidenced by his speculation on the applicability of Scott's ideas in his (1980) paper on the history and future of LISP—but did not hold much hope for the ultimate utility. Scott himself (2016) later acknowledged that the deep theoretical aspects of the work caused some problems: "In many ways I rather regret having cast the foundations of Domain Theory at that time in the form of lattice theory (and later DCPO theory). Some people liked it, and some people disparaged it as 'Scottery' and too abstract".

With the introduction of Scott's domains as the foundation, mathematical semantics had by 1970 reached a level at which it could be applied to the majority of programming language constructs successfully. The first paper on the grand unified scheme was released in 1971, with both Scott and Strachey as authors. This paper showed an obvious linking to mathematics right from the start, beginning by showing a 'semantic functions' for deriving numbers from numerals and noting that you could use the semantics to prove properties of operators, such as bitwise addition. This framing of the method perhaps showed the bite of Fox's (1961) comments about the practical uselessness of non-numerical computing still haunting Strachey. The rest of the present section explains the core concepts of Scott–Strachey semantics as of 1970.

Syntax of the language under definition is dealt with in a semi-abstract way: depending on the language's size, fragments of concrete syntax could be used for

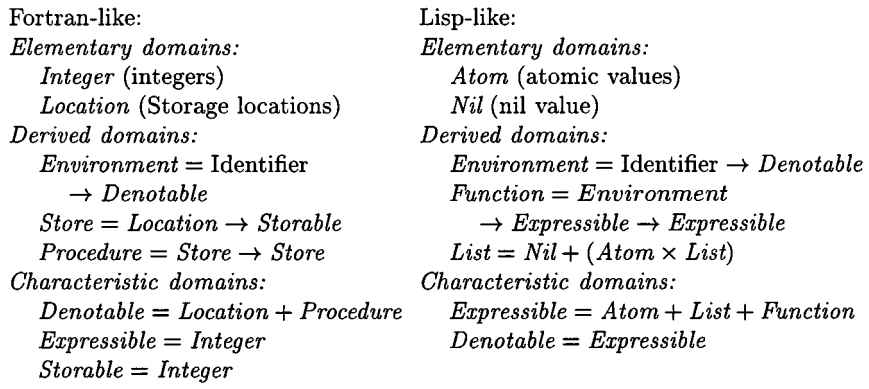


Figure 6.9: Semantics domains for two different types of language. Taken from Schmidt (2000, p. 92).

clarity of semantic phrases. The essential concept in use is 'annotated deduction trees': parses from programs tagged with labels that correspond to concrete syntax. This performs the same function as abstract syntax in the McCarthy and Vienna style but with more obvious naming. The method's lack of emphasis on syntax fits with "Strachey's first law of programming: 'Decide what you want to say before you worry about how you are going to say it' " (Barron 1975). All context-sensitive errors (such as mismatch of type between declaration and use) are caught in the semantics, and simply left undefined.

Syntactical and semantic domains of languages are identified early on in mathematical semantics definitions, and while it is the semantic equations of the language that show the meaning, a lot of the language's characteristics can be determined purely by examining the domains used. Strachey (1973a) wrote about this specifically, showing the domains of ALGOL 60 and PAL (a functional applicative language) and how different they appear. Schmidt (2000, p. 92) also gives an example of the domains of FORTRAN-like and LISP-like languages, as can be seen in Figure 6.9.

These clearly show that FORTRAN is a language for numerical computation, and LISP for symbolic lists. Strachey noted that working out domains' contents should be an early task for a language designer as they influence much of the language. Reynolds (in Jones et al. 2004) added, however, that if an aspect of the language is expressed by the semantic domains, it is basically assumed to be true; but the truth can only be determined through a huge case analysis of every part of the language. However, this is even more complicated in operational approaches, where properties of the state (corresponding with the semantics domains) relies on the steps of the

interpreting machine.

At its core, mathematical semantics is a mapping from programming language constructs (which are parts of the deduction trees) to the space of denotations. The denotations are, in their simplest form, functions from store to store, where a store is an association of locations with values. The store is typically represented as the domain S, members of which are σ.

A second argument to the semantic function is an environment, an association of identifiers with locations. This works together with the store to enable the sharing of variables across block locations, much like in the Vienna approach. Environments are members of the domain E and are represented by ρ. The splitting of environment and store is important; exactly how this idea came into mathematical semantics is not entirely clear. Scott (2000, p. 111) credited himself with the idea:

> [The separation of the store from the environment] was a suggestion of mine made in fall of 1969 when our joint work was just starting. The idea was common from logic and the semantics of the 'static' languages employed in the theory of models as understood by logicians. The distinction has become standard in semantical discussions.

However, the story is somewhat more complicated. 'Fundamental Concepts' (Strachey 1967) contains the ideas of both an environment for evaluating expressions, and a store for denoting commands, but the two were not clearly brought together in one semantic function. A formalisation of a very similar idea to the environment/store split was described in a mathematical semantics context by Park (1968). This was published in Edinburgh University's Machine Intelligence technical report series, but as Park had worked at the PRG until September 1968 it is likely the idea was largely formulated in Oxford. In this work, Park described a 'state' of two components, the first of which is a mapping from the 'legal expressions' existing at that point in the program into a class of locations, and the second being a mapping from this class of locations into a class of objects. This is remarkably similar to the Scott–Strachey approach, although unified in one tuple rather than split into two objects.

Tennent and Ghica (2000, p. 121) also noted that the approach was essentially the same as that of the Vienna group and the designers of ALGOL 68, although all seemed to be independently developed. In 1969, following discussion at WG 2.2 on models of storage, Walk (1969a, p. 22) argued that the Vienna method and Strachey's were conceptually the same: "i.e. there is a functional relationship between

names (addresses), values, and the store". Lucas, however, wanted to know why environment could not simply be part of the state, as it had in ULD; Scott explained that the advantage of keeping it apart meant that it could be more easily separated in proofs (in Walk 1970, p. 33).

Strachey (1973a, p. 11) explained the concepts of store and environment:

> The two functions ρ and σ together with their associated domains D and V go a long way to characterising a programming language. There is a fundamental difference between these two functions which is the source of many of the confusions and difficulties both about programming languages and also about operating systems. This is that while the environment ρ behaves in a typically 'mathematical' way—several environments can exist at the same point in a program, and on leaving one environment it is often possible to go back to a previous one—the machine state σ which includes the contents function for the store, behaves in a typically 'operational' way. The state transformation produced by obeying a command is essentially irreversible and it is, by the nature of the computers we use, impossible to have more than one version of σ available at any one time. It is this contrast between the static, nesting, permanent environment, ρ, and the dynamic irreversibly changing machine state, σ, which makes programming languages so much more complicated than conventional mathematics in which the assignment statement, and hence the need for σ, is absent.

A key property for mathematical semantics is that it is, as much as possible, 'homomorphic': meaning of compound constructs is made from composing the meaning of constituent parts. This means compositionality is preserved and sequencing can be handled by functional composition, which was first mentioned by Strachey in 'Fundamental Concepts' (Strachey 1967, p. 25). The importance of this is explained by Schmidt (2000, p. 90) who notes that this makes the definitions *inductive*: proofs can be made by induction on the subcomponents. This then mirrors the way that the syntax is defined with BNF. Schmidt gives an example:

$$\mathcal{A}[\![C_0; C_1]\!] = \lambda\rho \cdot \lambda\sigma \cdot (\mathcal{A}[\![C_1]\!]\rho)(\mathcal{A}[\![C_0]\!]\rho\sigma)$$

The meaning of two commands in sequence (C_0 and C_1) is formed by the functional composition of their denotations. This little equation also highlights some other important properties, such as showing that there is associativity of command sequencing, that the environment is unchanged by commands, and that the store produced by the first command becomes the store used in the second command.

The denotations used, as has been seen, are unnamed functions defined by λ expressions. As Strachey had said many times (for example in van Wijngaarden 1966b, p. 23), the advantage of using funtions as the base is that they are well-known mathematical objects with well-known properties. This makes reasoning about them more straightforward and tractable than in operational approaches like the Vienna Group's, where inductive reasoning had to be performed over the steps of the complicated interpreting machine.

Notational convention in Scott–Strachey semantics uses capital Roman letters for domains of interest, calligraphic capitals for semantic functions, and lower case, often Greek, letters for identifiers and function arguments. Fragments of deduction trees corresponding to program parts are enclosed within 'Strachey' brackets $[\![\,]\!]$. The semantic functions take multiple arguments, but rather than in a tuple these are Curried (although they could just as easily be bound in with a lambda on the right-hand side).

This means that instead of a function being defined as $\mathcal{A} : B \times C \to D$, it is written $\mathcal{A} : B \to C \to D$. This notational choice allows easier recognition of the homomorphism at play; Stoy argued that it also allows varying levels of detail to be supplied for a different meaning (adapted from Stoy 1977, pp. 253–4):

$\mathcal{A}[\![t]\!]$ is the meaning of a command by itself;
$\mathcal{A}[\![t]\!]\rho$ instantiates the variables by adding in an environment, forming a closure;
$\mathcal{A}[\![t]\!]\rho\sigma$ is a particular execution of the command with a particular store.

Another crucial element to mathematical semantics is the use of fixed points to derive representations of loops and recursion in programs.

Consider the following loop:

$$\textbf{while } x > 0 \textbf{ do } x := x - 1$$

Intuitively, it is clear to see that this terminates with $x = 0$ for $x > 0$ and will not terminate for $x < 0$. In each iteration of the loop, the value of x decreases, and at the limit of any value of x a graph of the function finishes at 0. Through this limit construction one can determine the fixed point of the function.

The storing of command denotations in the store means that recursive definitions create loops in the abstract representation, as Gordon (2000, p. 65) explained:

> The technique of 'tying a knot' with pointers [L-values] was then a standard way to implement recursion and could be represented by constructing circular loops of pointers in a store. This technique raised the question of whether such loops could be characterised as least fixed-points.[31]

The fixed points of the recursive lambda expressions used to define loops and recursion represent potential denotations of the commands. Scott's lattice theory for the domains of these functions imposes a partial ordering for which there is always a minimal solution. This then represents the least fixed point of the loop, which is the desired representation. Mathematical semantics definitions utilising fixed points tended to include a *fix* function which can always find the fixed point, and its existence is made possible by the continuity property of the domains. That there are firm mathematical proofs of this property gives the least fixed point method surety, and allows its replacement of the 'paradoxical' Y combinator. Schmidt (2000, p. 91) argued that this method of defining loops inductively also forces language designers to confront the mathematics of iteration.

Scott (2000, p. 112) explained some of the historical development of the fixed point operators and their related induction rules, starting with a quotation from a historical reflection written by Strachey (in Milne and Strachey 1974, p. 29):

> The relevant theorems about fixed points, due to Knaster and Tarski, had little impact on computing [as distinct from recursive function theory] until the work of David Park and Scott. Landin in his 1964 paper [(Landin 1964)] made use of the 'paradoxical combinator', Y, discussed by Curry and Feys. In λ-calculus Y has the formal properties of a fixed-point operator, the definition of which requires the application of a function to itself. After Scott had completed his theory of reflexive domains, Park showed that this operator could be identified with the [minimal fixed-point function]. The paradoxical combinator was extensively used in the early stages of mathematical semantics as a minimal fixpoint operator without any satisfactory mathematical justification; fortunately, subsequent events showed that these uses were in fact substantially correct. The development of induction rules for mathematical semantics was started by McCarthy, who introduced 'recursion induc-

[31] Later work showed the similarity of this approach to LISP's implementation of recursion using dynamic binding. Wadsworth (1976) demonstrated the connection between LISP functions and fixed points, and Gordon (1975) showed that LISP's *eval* function was equivalent to mathematical semantics. This is yet another example of the extraordinarily lasting appeal of McCarthy's language, and its continuing relevance to functional programming: it remained, more than 15 years after its appearance, as a standard by which the theoretical value of semantic approaches could be judged.

> tion'. This rule turned out to be inadequate; more satisfactory rules were discussed by de Bakker and Scott.

Scott continued:

> The point here is that McCarthy's domains of computation were too limited, and the rôle of continuity of functions and inclusivity of predicates was not apparent. When the better domains were introduced, better induction rules were required.

Another important contribution of the domain theory is that it allowed the treatment of higher-order functions. This was necessitated simply by the storing of functions ($S \rightarrow S$) within elements of S, but also allowed functions which represent higher-order procedures to be properly denoted, although their representations became increasingly complicated. Defining programming languages also becomes complicated when more complex language features like recursively defined procedures with side-effects must be considered. These make the denotations used in the store more convoluted; work on this was performed later by Milne, Reynolds, and the latter's student Frank Oles. Such considerations are increasingly technical, and the paper by Tennent and Ghica (2000) provides a sufficient explanation for the interested reader.

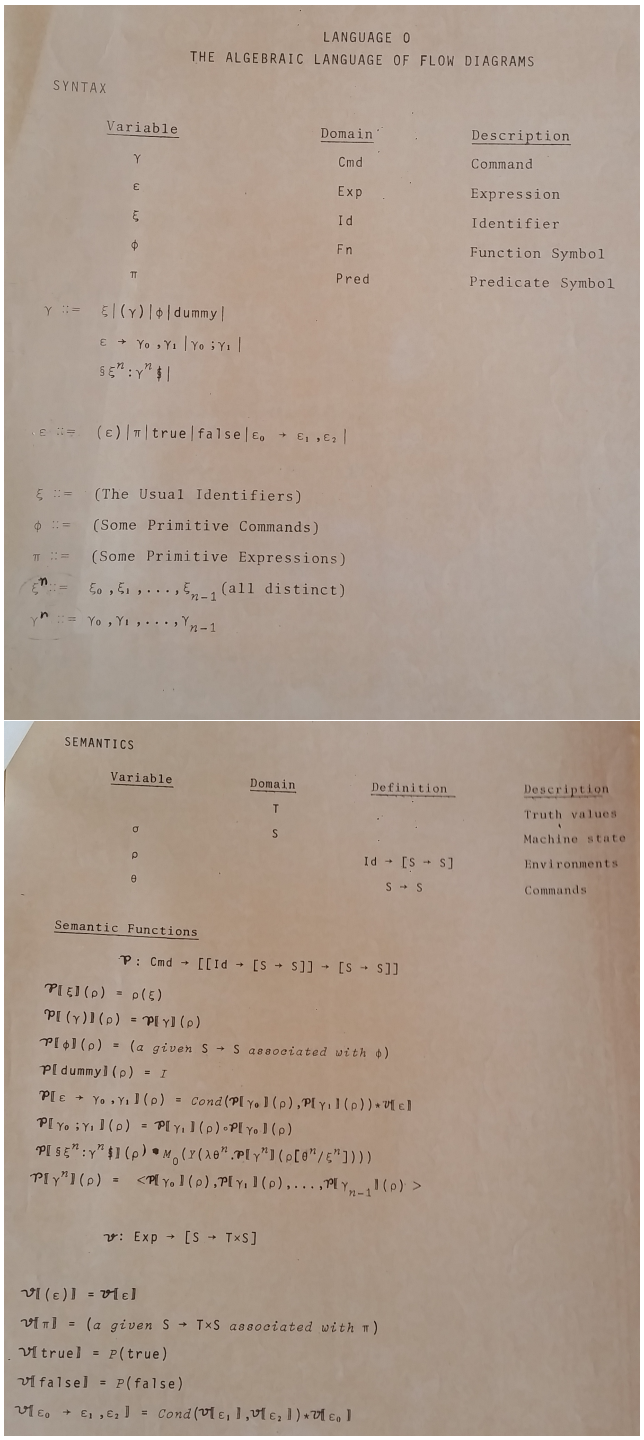
A more detailed technical explanation of the mathematical semantics approach is beyond the scope of the current work, but is explored in another paper by the current author, alongside comparisons with other semantic approaches (Jones and Astarte 2018). Further technical details can be found in Stoy's (1977) textbook and a shorter summary paper (Stoy 1980). Tennent (1976) also wrote a good technical overview.

Strachey loved to demonstrate his approach with very compact language descriptions, often fitting on one piece of paper. An example here is taken from the archive and shown in Figure 6.10.

Hoare (1994), however, indirectly criticised this:

> I'm constantly trying to absolutely emphasise the role of proving the statements as you go along piece by piece. Never present a semantics as a page of axioms, definitions or deductions. Always try to prove on the basis of something else.

The period from Scott's first visit in 1969 to his second in Spring 1971 was one of the busiest in Strachey's time at the PRG and was when the really important



LANGUAGE 0
THE ALGEBRAIC LANGUAGE OF FLOW DIAGRAMS

SYNTAX

Variable	Domain	Description
γ	Cmd	Command
ε	Exp	Expression
ξ	Id	Identifier
ϕ	Fn	Function Symbol
π	Pred	Predicate Symbol

$\gamma ::= \xi \mid (\gamma) \mid \phi \mid \text{dummy} \mid$
$\varepsilon \to \gamma_0, \gamma_1 \mid \gamma_0 ; \gamma_1 \mid$
$\S \xi^n : \gamma^n \$ \mid$

$\varepsilon ::= (\varepsilon) \mid \pi \mid \text{true} \mid \text{false} \mid \varepsilon_0 \to \varepsilon_1, \varepsilon_2 \mid$

$\xi ::=$ (The Usual Identifiers)
$\phi ::=$ (Some Primitive Commands)
$\pi ::=$ (Some Primitive Expressions)
$\xi^n ::= \xi_0, \xi_1, \ldots, \xi_{n-1}$ (all distinct)
$\gamma^n ::= \gamma_0, \gamma_1, \ldots, \gamma_{n-1}$

SEMANTICS

Variable	Domain	Definition	Description
	T		Truth values
σ	S		Machine state
ρ		Id → [S → S]	Environments
θ		S → S	Commands

Semantic Functions

$\mathcal{P}: \text{Cmd} \to [[\text{Id} \to [S \to S]] \to [S \to S]]$

$\mathcal{P}[\![\xi]\!](\rho) = \rho(\xi)$
$\mathcal{P}[\![(\gamma)]\!](\rho) = \mathcal{P}[\![\gamma]\!](\rho)$
$\mathcal{P}[\![\phi]\!](\rho) = (a\ given\ S \to S\ associated\ with\ \phi)$
$\mathcal{P}[\![\text{dummy}]\!](\rho) = I$
$\mathcal{P}[\![\varepsilon \to \gamma_0, \gamma_1]\!](\rho) = Cond(\mathcal{P}[\![\gamma_0]\!](\rho), \mathcal{P}[\![\gamma_1]\!](\rho)) * \mathcal{V}[\![\varepsilon]\!]$
$\mathcal{P}[\![\gamma_0 ; \gamma_1]\!](\rho) = \mathcal{P}[\![\gamma_1]\!](\rho) \circ \mathcal{P}[\![\gamma_0]\!](\rho)$
$\mathcal{P}[\![\S \xi^n : \gamma^n \$]\!](\rho) = M_0(Y(\lambda\theta^n . \mathcal{P}[\![\gamma^n]\!](\rho[\theta^n / \xi^n])))$
$\mathcal{P}[\![\gamma^n]\!](\rho) = \langle \mathcal{P}[\![\gamma_0]\!](\rho), \mathcal{P}[\![\gamma_1]\!](\rho), \ldots, \mathcal{P}[\![\gamma_{n-1}]\!](\rho) \rangle$

$\mathcal{V}: \text{Exp} \to [S \to T \times S]$

$\mathcal{V}[\![(\varepsilon)]\!] = \mathcal{V}[\![\varepsilon]\!]$
$\mathcal{V}[\![\pi]\!] = (a\ given\ S \to T \times S\ associated\ with\ \pi)$
$\mathcal{V}[\![\text{true}]\!] = P(\text{true})$
$\mathcal{V}[\![\text{false}]\!] = P(\text{false})$
$\mathcal{V}[\![\varepsilon_0 \to \varepsilon_1, \varepsilon_2]\!] = Cond(\mathcal{V}[\![\varepsilon_1]\!], \mathcal{V}[\![\varepsilon_2]\!]) * \mathcal{V}[\![\varepsilon_0]\!]$

Figure 6.10: Strachey's example semantics of Language 0, the algebraic language of flow diagrams. The compact nature of the notation was something that clearly pleased Strachey, who put out a good many of these two-page semantics. Taken from the Strachey Archive, Box 275, C.225.

foundational work in mathematical semantics was laid down. Strachey's timeline for this period can be seen in Figure 6.11.

6.5 Developing the idea

After Michaelmas Term 1969, Scott returned to Princeton, spending another sabbatical term at Oxford in Spring 1971. In 1972, he accepted the new chair of Mathematical Logic at Oxford; Strachey, too, had received a promotion, becoming Professor of Computation in 1971. Unfortunately, Scott and Strachey both (in Scott's words) "had so many obligations and duties as new professors at Oxford that [their] joint work could never again be so concentrated" as it was in 1969 (Scott 2000, p. 112). Scott did, however continue to work on the logical foundations, refining his models of lambda calculus.

With these foundations now firmed by Scott, Strachey was very excited by the developments. What had been, before Scott, a good idea with some shaky places was now a rich, deep, and powerful system for language definition and investigation. Strachey reported this to the SRC in his annual report in early 1970:

> However, although [by 1969] some of the problems were satisfactorily resolved and a good deal of experience was gained in coping with the notational problems (which are very considerable), the results lacked a firm mathematical basis and did not form a convincing whole. This position has now completely changed as the result of a visit from Professor Dana Scott which started in September, 1969. Strictly speaking this work should be reported in July 1970, as part of the year 1969/70, but the work is so important and the impact it has had upon the Programming Research Group has been so great that I have preferred to report it immediately ...
>
> The work of applying these ideas in detail to the formal semantic theory is only just beginning and there are some areas, particularly those concerned with generalised jumps, which remain obscure. But I now feel convinced that we have the proper mathematical basis for a satisfactory theory of computing and I am looking forward to its development in the next year or two.
> (Strachey, quoted in Campbell-Kelly 1985, p. 37)

Various developments and refinements of the mathematical semantics idea continued to be created, such as Strachey's piece on the 'Abstract model for storage' (Strachey 1971c). Yet another unpublished work, this was an explication of the properties and

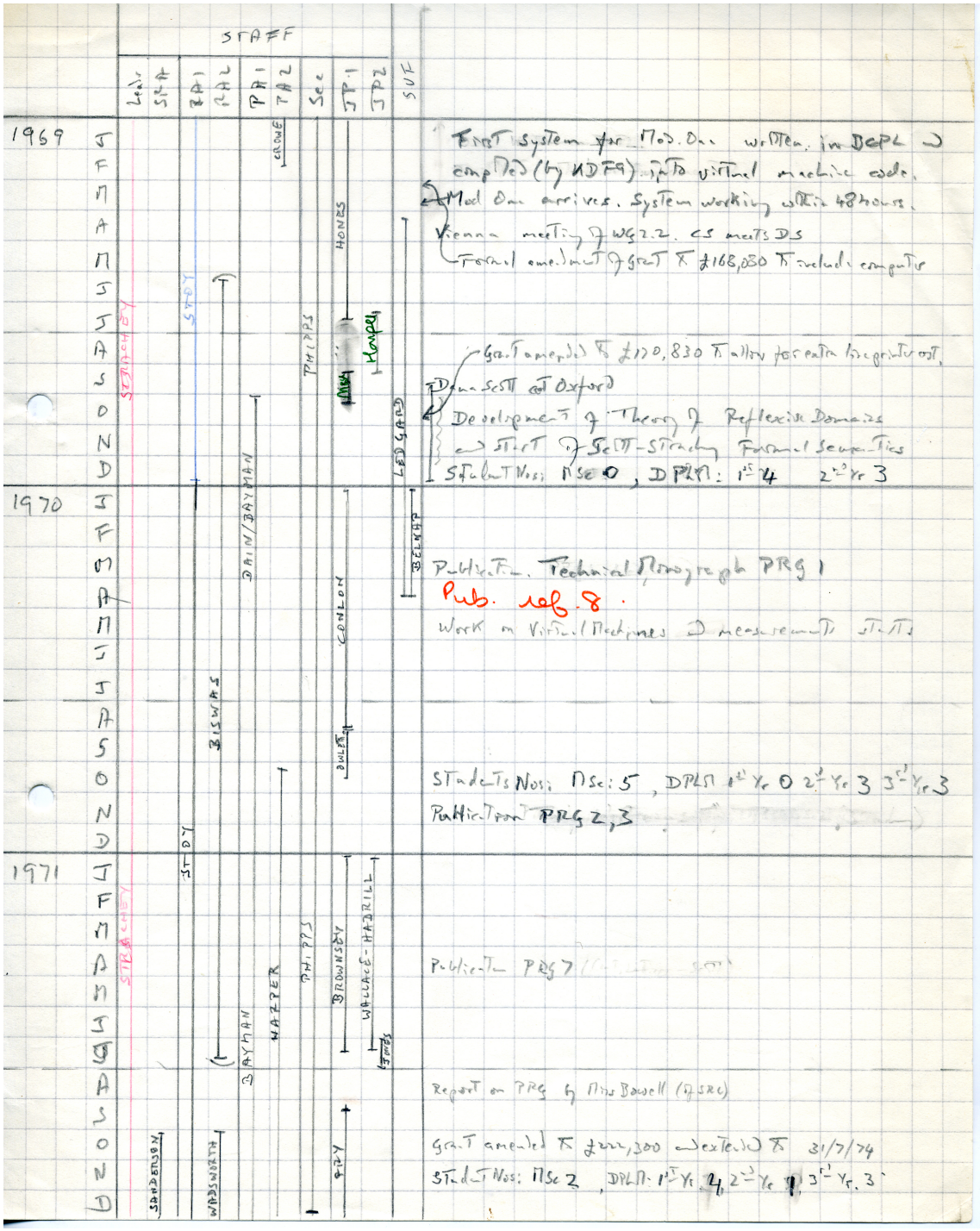


Figure 6.11: A chart of employees at the PRG from 1960 to 1971. The right hand column marks notable events.

basic functions required for the store. The three major steps forward for the method were all driven by PRG graduate students and are described below.

6.5.1 Wadsworth and continuations

The biggest extension to mathematical semantics, after Scott's domain theory, was the notion of continuations: a way to model remaining computation using functions that could be used to handle jumps, among other things. Variations on this idea were constructed and published by many diverse researchers, as Reynolds reported in a paper (Reynolds 1993) and expanded upon in a 2004 talk at the Science Museum (Reynolds 2004). These inventors include Landin, with the J operator discussed in Section 3.2, and van Wijngaarden's syntactic transformation of go tos into procedures (see Section 3.3). Van Wijngaarden's work did not make an explicit reference to this idea as a named concept, however, and those present at his talk, such as McIlroy, did not spot the power and flexibility of the concept at the time (McIlroy 2018). Mazurkiewicz (1971) worked out the core of the continuations idea with his 'tail functions' which, while not as powerful as later formulations, were the first time the concept had been given an explicit name. Lockwood Morris (1970) described 'dump functions'[32] which also performed the same function, but these were not widely known at the time.

For the denotational semantics story, the important contributor is Christopher P. Wadsworth, who was a doctoral student at the PRG before joining as a research assistant in 1971. His doctoral work was on graphic models of lambda calculus that allowed a better kind of normal order reduction (Wadsworth 1971). Wadsworth and Strachey had been trying to develop a way to handle jumps even before Scott arrived (Wadsworth 2000, p. 131). The key issue was providing a homomorphic model of a go to statement. The content of such a command is just a label and there is no obvious way in which its meaning can be contained in something derived from that label. In the period from 1968 to 1970, the pair tried a number of approaches, many of which can be seen in the Strachey archive as handwritten notes (Box 275, C.229). Some of these worked mathematically but were unacceptably convoluted:

> One approach will give a flavour of the elaborate nature. This involved distinguishing three kinds of jumps—hops (local to a block), skips (out of

[32]The idea was described in a talk at Queen Mary called 'The next 700 Formal Language Descriptions', a nod to Landin.

> blocks or procedures) and (full) jumps (possibly back into earlier blocks or procedures). The original program was thus transformed to mark up the three kinds of jumps. After a good deal of trial and error, mostly to handle the subtle interaction with changes of environment, we found definitions that appeared to give a correct treatment, but the complexity seemed to us greater than acceptable for making effective use through such descriptions, and altogether too intricate compared to the rest of the descriptions.
> (Wadsworth 2000, p. 131)

Precisely who developed the idea that ended up working best is unsure; Wadsworth (2000, p. 132) reports:

> Then in October 1970 Strachey showed me a paper 'Proving algorithms by tail functions' by Mazurkiewicz (Mazurkiewicz 1971) which he had obtained from an IFIP WG2.2 meeting. Just the phrase 'tail functions' in the title was enough—given the experience of our earlier struggles—for the ideas to click into place! The (meaning of the) 'rest of the program' was needed as an argument to the semantic functions—just so those constructs that did not use it, like jumps, could throw it anyway. The term 'continuation' was coined as capturing the essence of this extra argument (though I often wished to have a shorter word!) and the rest, as they say, is history.

However, the ideas were clearly circulating a little earlier than that. In a handwritten note in the archive, dated April 1970, Strachey can be seen working out an idea of cutting programs into 'program segments' and manipulating those. The important passage runs as follows:

> If γ [a program segment] is general (i.e. may have an abnormal exit) it must be treated as terminal. Thus its normal exit must be made terminal in the same way as $\Sigma \to \tau_0$, γ has to be changed to $\Sigma \to \tau_0, \tau_1$. This is done by supplying an extra parameter which is the normal successor.
> (Strachey 1970c)

The precise meaning of the Greek letters is unimportant in this context; the point is that there is the idea of an extra argument to the semantic function to capture the command's successor which can be manipulated if a jump is encountered. Otherwise, it represents the result of the rest of the computation. This is the core idea of the continuation.

Stoy gives an illustrative example of the continuation as it was canonically defined:

$$\mathcal{A}[\![\Gamma_1;\Gamma_2]\!]\rho\theta = \mathcal{A}[\![\Gamma_1]\!]\rho\{\mathcal{A}[\![\Gamma_2]\!]\theta\}$$

Essentially, Γ_1's continuation is the denotation of Γ_2, and the continuation for the composition is the continuation of Γ_2. Thus continuations preserve compositionality. This equation also shows that the environment ρ is unchanged in this process. Commands are supplied with continuations simply so they have the option of being ignored should an abnormal termination occur.

Immediately after the Mazurkiewicz inspiration, Wadsworth (1970, 1972) began to write notes explaining the idea.[33]

Neither of these were published, but of course circulated informally, and were taken forward at the PRG and by Reynolds at Syracuse (Wadsworth 2000, p. 132). A published version eventually appeared for a miniature language, authored by Strachey (1973b), and towards the end of the year, as Wadsworth prepared for a move to Syracuse, a PRG monograph was written jointly (Strachey and Wadsworth 1974). This was the key reference for continuations and with this the denotational semantics approach was able to handle most aspects of programming languages.

Wadsworth (2000, p. 132) notes that he tried to get more complicated ideas into this paper to show that continuations were powerful enough to handle complex language techniques, but Strachey preferred to keep it simple—especially as further delay in publication would be undesirable. Wadsworth adds that this simplicity in technical pieces was typical of Strachey:

> Strachey had an acute sense of when something was 'right'—generally when it was simple enough and elegant enough that it could be seen intuitively to be right—and he abhorred overelaboration or contrived methods that 'sort of worked'. A favourite motto of his—I am not sure if he coined it himself or was using it from elsewhere—was "You can push a pea up a mountain with your nose if you really want to, but that does not mean that it is a good way of getting it there". For me, this was a kind of 'Strachey test'.

Stoy's meaning stack can now be developed as follows (adapted from Stoy 1977, pp. 253–4):

$\mathcal{A}[\![t]\!]$ is the meaning of a command by itself;
$\mathcal{A}[\![t]\!]\rho$ instantiates the variables by adding in an environment forming a closure;

[33] A manuscript of the first can be seen in the Strachey archive, Box 275, C.238.

$\mathcal{A}[\![t]\!]\rho\theta$ adds a continuation, making the command 'ready to go', and is the denotation of a label;
$\mathcal{A}[\![t]\!]\rho\theta\sigma$ is a particular execution of the command with a particular store.

Tennent (1976, p. 447) described the continuation method of interpretation for expressions and commands as 'prophetic', since "they must specify not merely the local result or effect of a construct, but its contribution to the final result of a complete program or process execution". Jones, who, along with other members of the Vienna Group which had reconvened to write another description of PL/I at this time, was not keen on the continuations approach, later terming it "too powerful" simply for the task of modelling abnormal termination (Astarte and Jones 2018, p. 121). Jones worked out an alternative mechanism which used combinators; more is discussed of this in Section 7.3. In his final published piece, Landin, reflecting on continuations and their various guises, seemed to hold a similar view about his J operator: "J systematised three 'enhancements', i.e., corruptions, of strict lambda (viewed operationally), that helped it to model goto's. It might be said that they were so powerful that they could hardly help not" (Landin 1997, p. 1-2).

6.5.2 Mosses and a formalised metalanguage

Another interesting extension to mathematical semantics—although not as essential to the method as continuations—was provided by Peter D. Mosses, another graduate student at the PRG. Mosses developed a formalisation for the denotational semantics metalanguage, worked on a system for deriving compilers from semantic descriptions, and also wrote a definition of ALGOL 60 in the Scott–Strachey style.

Mosses had studied mathematics at Trinity College, Oxford, the same as Wadsworth, although three years later. Between his second and third years, Mosses attended a summer school at the PRG, which entailed one week learning ALGOL 60, one studying numerical computation with Fox, and a third on symbolic computation under Strachey.[34] Whilst in his third year of study, Mosses embarked on a project with a few classmates to write an ALGOL 60 compiler for the PRG's Modular One computer; this was ultimately unsuccessful, but Mosses learnt BCPL and his contact with the PRG grew. Strachey encouraged Mosses to attend Scott's seminars on domain theory, ongoing at that point in 1969.

[34] Much material here on Mosses comes from an informal interview with the current author, in June 2016.

After completing his first degree, Mosses took job interviews at ICL and IBM Hursley, receiving a job offer from IBM, but he decided instead to enrol on an MSc in Mathematics at Trinity College, Oxford, in 1970. This was essentially a computing conversion course delivered by the Programming Research Group to lead students into further research on computation. For his Master's thesis, Mosses worked on a grammar and parser for semantic equations that became the basis of his formalised metalanguage.

Mosses' DPhil thesis, supervised by Strachey and examined by Burstall, took the idea and extended it to define the semantics of the metalanguage, which Mosses called Mathematical Semantics Language, MSL. MSL itself was presented in a paper at a Mathematical Foundations of Computer Science symposium in June 1974 (Mosses 1975b). Prior to Mosses' formalisation, the metalanguage was rather informal and imprecise, something of which Strachey was aware, writing that it was "almost nothing more than a notation" (Milne and Strachey 1974, p. 16). A fully formalised metalanguage allowed Mosses to design a system that could generate prototype compilers directly from a denotational semantic description. This was called the Semantics Implementation System, SIS (Mosses 1975a). It is worth mentioning that the hope for automatically generated compilers was recognised by others, such as Tennent (1976), but Mosses made the most progress on actually creating such a system. SIS was written for the Modular One in BCPL and used lambda calculus with call-by-need semantics.

The process of SIS was based on that of a computer system, and described in terms of a series of abstract objects transformed by functions. A program text is transformed into parse-trees by a parser, meaning is given by a semantic function, and implementation creates a function from input to output. The mechanisation of these objects was given by using a simulation in a language called LAMB based on Scott's LAMBDA language, which was itself a syntactically sugared version of pure lambda calculus. This was put together in SIS by creating the LAMB code of the components from high-level descriptions: the parser was written in a BNF grammar, and the other components in MSL. Mosses argued that MSL provides advantages over semantic equations when defining a language, although he acknowledged that for descriptions and discussions the compactness of the latter might be an advantage. He also believed that MSL would be helpful to a novice to semantics with a computing background as it looks more like a programming language (and can even be given an operational interpretation).

Strachey was very excited by this project, writing in a grant application (quoted in Mosses 2016) "Its use forces language designers to think clearly about the semantics of their language in the same sort of way as the introduction of BNF made language designers think clearly about syntax" and "I have high hopes that it will be possible to prove the validity of the whole system by a demonstrable mathematical argument". However, the system was very slow to run, and its object programs equally inefficient. Development on SIS did not finish until 1979, at which time Mosses had moved to Aarhus (Mosses 2016).

As a side project during his doctoral work, Mosses wrote a mathematical semantics of ALGOL 60, using MSL.[35] It is interesting to consider why ALGOL 60 was chosen for definition given Strachey's previous work with Wilkes criticising the language. One clue comes from the acknowledgements in the preface to Mosses' description:

> The original inspiration for this report came from reading [the Allen, Chapman, and Jones (1972) ALGOL description][36] and [Peter Landin's 'Correspondence' descriptions (Landin 1965a,b)], as it was felt that a shorter and less algorithmic description of ALGOL 60 could be formulated in the Scott–Strachey semantics.
> (Mosses 1974, p. 4)

Strachey also wrote that writing the full semantics of ALGOL 60 showed problems in the language, just as many others, especially Landin, had argued:

> The attempt to apply this method to existing languages shows that even such languages as ALGOL 60 contain a number of confusions or unfortunate choices which greatly lengthen their standard semantic descriptions and complicate their compilers. The use of standard semantics should help language designers to avoid these dangers at least as much as the introduction of BNF clarified the syntax.
> (Milne and Strachey 1974, p. 22)

ALGOL 60 was clearly becoming seen as the standard for demonstration of semantic descriptions; Mosses, like the Vienna Lab and Landin, used it as a way to indicate the advantages of his style. Also, at this time, the continuations method for defining

[35] Stoy (2016b) argued that this was the practical complement to Mosses' theoretical work, and continued to emphasise Strachey and the PRG's desire to span both aspects of computing.

[36] This description was written by some researchers at the IBM Hursley laboratory, and is described further in Section 7.3. In essence, it can be seen a stepping stone between the Vienna large ULD definitions, and their later work with denotational semantics.

jumps had been finished and it was useful to show their application to a full-sized language. Mosses remembers[37] that CPL and Pascal were also considered, but CPL was too large and poorly defined, and Pascal's syntax definition was in a diagrammatic railroad style that would have to be translated first, where ALGOL 60 already had a BNF grammar written.

Mosses did originally intend for the ALGOL 60 definition to be run on SIS, but this was never actually achieved. Nevertheless, the description remains the most ambitious and complete full-language description produced by the PRG. It uses the standard approach of using one-letter variables, but the formalised MSL notation is considerably denser to read, and a significant commentary section is required to decode the text of the description, which offers little intuition to the reader. Wadsworth (1974) wrote of it in a letter to Strachey "I must admit I still feel a little surprised it's as long as it is—I guess ALGOL 60's just not nearly as 'well-behaved' as one tends to think it is".

6.5.3 Milne and the Adams Essay

The final chapter in Strachey's involvement with semantics came in the middle of the 1970s. Together with another graduate student, Robert E. Milne, he wrote a very lengthy essay on his semantic approach, intended for submission to the Adams Prize at Cambridge University.

Campbell-Kelly reports that it was a great ambition of Strachey's to be elected a Fellow of the Royal Society, especially towards the end of his life (Campbell-Kelly 1985, pp. 38–9). Strachey wrote to a number of contacts on this subject, who provided advice and guidance. One of these was Lord Halsbury, Strachey's long-time friend and champion from his NRDC days, to whom Strachey provided a list of people who could be relied upon to support his election (Strachey 1971b). Another was James Hardy Wilkinson, already a Fellow, who told Strachey that it was difficult for computer scientists to be elected, especially if they had relatively few publications, as Strachey did (Wilkinson 1972). Wilkinson suggested that Strachey submit an essay for Cambridge's Adams Prize in Mathematics for 1973–74 as a way to remedy this; winning the prestigious award would go some way towards showing the Royal Society that Strachey was worthy of election.

Strachey followed this advice, co-authoring with Milne, a Cambridge doctoral stu-

[37] In the same private conversation, June 2016.

dent with whom he had been working.[38] Their work together included a method for using continuations to model parallelism, which worked but was unsatisfactory as it could not be used for many desirable proofs, and even those which could be achieved were complex (Milne 1972).

According to Scott (2016, p. 14), he himself might have been a third author, had there not been a requirement that all authors be holders of degrees from Cambridge. Indeed, the Strachey archive (Box 275, C.224) contains two overviews for the structure of a book entitled *A Mathematical Theory of Computation*, one handwritten by Scott and a second handwritten by Strachey. There are interesting differences between the two: Scott started with a part on the construction of domains before moving on to the applications in formalising languages, where Strachey began with an informal introduction to languages before introducing the mathematics of domains. Scott (2000, p. 112–3) also suggested that part of Strachey's motivation for writing the Adams Essay could have been to demonstrate his worth to Cambridge: his undergraduate degree was undistinguished, and his work there as a researcher on CPL had not been a great success. Finally, Strachey had been approached by Academic Press in 1973 and invited to write a book on the theory of computation; his response was tentative enthusiasm, and he thought the Adams Essay could form the core of such a book, which would be authored instead with Scott (Strachey 1973d).

The resultant essay submitted to Cambridge was called *A Theory of Programming Language Semantics* (Milne and Strachey 1974), and was intended as a comprehensive account of the fundamental concepts in programming languages and how they could be modelled in a denotational style. Its submission came just before the deadline with some sections still incomplete (Campbell-Kelly 1985, p. 39). The essay was illustrated with the full description of a large purposely invented language, Sal, and a method for giving implementations derived from formal semantics, complete with proofs of equivalence and correctness.

Despite its incompleteness, the essay was very lengthy, running over six hundred pages. The conclusion of the introduction makes a plea for forgiveness of this:

> A superficial glance will show that our essay is long and our notation elaborate. The basic reason for this unwelcome fact is that programming languages are themselves large and complex objects which introduce many subtle and rather unfamiliar concepts. We have preferred to

[38] Milne's degree was awarded by Cambridge, but he spent the majority of his time at the Programming Research Group in Oxford, and Strachey was his main supervisor (Milne 2016).

> use a large language (Sal) as our example rather than the trivially small languages which have been used by most authors when discussing semantic questions, in order to disarm the criticism that formal semantics can only deal with unrealistically simple solutions.
> (Milne and Strachey 1974, p. 22, also quoted in Campbell-Kelly 1985, p. 39)

The task of writing the essay was very difficult; according to Stoy (2016b) "it was absorbing Christopher's time very completely—all waking hours". Stoy believed that Milne was producing a great deal of mathematics with which Strachey had to strive to keep abreast, all the while incorporating sufficient prose into the essay. Strachey "wanted that the wood shouldn't be lost in the trees, that the essay wouldn't just be lemmas and theorems". Strachey himself wrote in March 1974 that he was "just keeping [his] head above water", turned down many invitations for engagements, and stopped attending WG 2.2 meetings at this time (Strachey 1974a). Strachey took a few weeks' rest in early 1975, returning to begin revising the essay into a book (Campbell-Kelly 1985, p. 39). However, in Spring 1975, Strachey contracted a disease first identified as jaundice, made an apparent recovery, and then died suddenly of infectious hepatitis on 18 May 1975, shortly after hearing that the submission for the Adams Prize had not been successful (Scott 2016, p. 14).

A timeline of Strachey's last period at the PRG can be seen in Figure 6.12.

The essay was not published in its co-authored form, although well-photocopied drafts remain. Instead, Milne rewrote it after Strachey's death, unable to easily reconcile the parts written by Strachey with the new parts he was writing himself (Milne and Strachey 1976, p. 9). The resultant book (Milne and Strachey 1976) is very different to the original draft due to the different writing styles of the two men. It is immediately clear from reading the draft that large portions of the early chapters are written in Strachey's voice; Milne (2016) later noted that he removed the personal and historical remarks. Scott (2000, p. 104) speculated that it might not have been possible for Strachey to have written the piece anyway:

> To me, in any case, it is doubtful whether Strachey could have written a satisfactory beginning text himself. [...] What he could have done, and did do on several occasions, was to produce shorter pieces which would, with great clarity, put forward what were the basic ideas and important problems, since his unique experience gave him special insight. True, he had his prejudices and hobby-horses, but many of us could, and did, forgive him everything for the sake of his excellent ideas. This is not

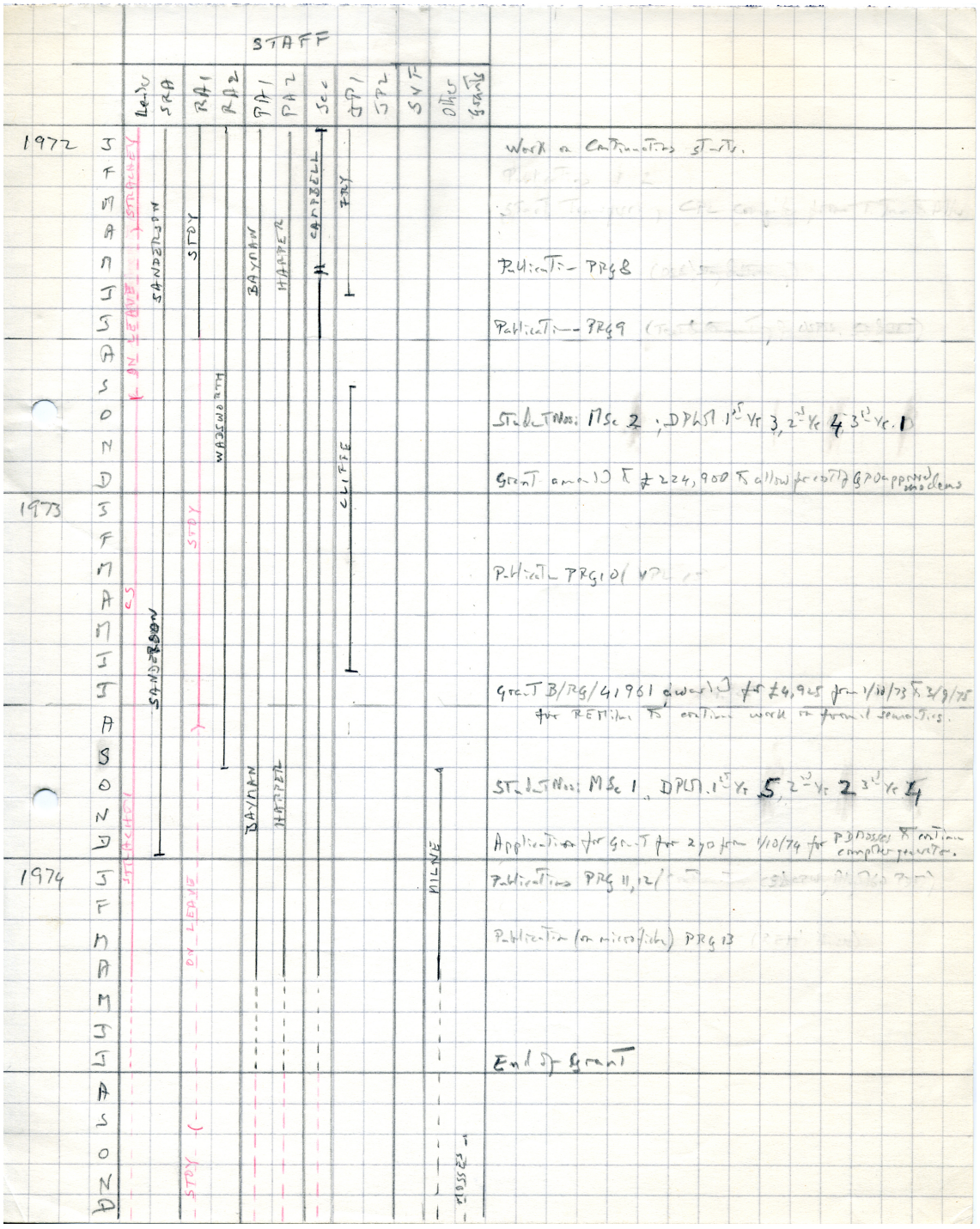


Figure 6.12: A chart of employees at the PRG from 1972 to 1974. The right hand column marks notable events.

> meant to imply that he was incapable of sustained work, since that is far from the truth, but academic writing was a considerable chore for him and in a certain sense was a waste of his time.

A strong reason for Milne's need to rewrite was his own substantive contribution. This was especially on the topic of the requirement for different levels of semantic description, perhaps in different styles, for different users. Milne (2000, p. 78) later summarised these:

> - A language designer may gain insight from conceptualising the problem using stores, environments and continuations. The expression of this insight to users of the language may take a verbal form or sometimes even a graphical form, but it can be both fairly exact and easily understood if it uses these concepts.
> - A program designer may prefer to think in terms more directly related to the surface structure of the program. In this case results about program constructs (algebraic equivalences and logical assertions) may be useful. [...]
> - An implementation designer may prefer to think in terms more directly related to the main components of the implementation. [...] What is needed is something more concrete than stores, environments and continuations but having the same merit of exposing essential distinctions without obscurity.

To this end, Milne's book included the full definition of Sal, a language with enough diverse features to show that the denotational approach could cover any language. Proofs were provided of "several algebraic equivalences and of two implementations (one 'interpreted' and one 'compiled')" (Milne 2000, p. 79). Establishing appropriate equivalence relations between the implementations allowed Milne to show that the different implementations produced correct results. This was essentially the same approach as McCarthy had taken a decade earlier; although Milne did cite McCarthy, he did not give a great deal of credit to the earlier researcher. While these ideas had been present in the joint draft in a preliminary form, they were significantly expanded in Milne's book. Milne and Strachey (1974, p. 17) had noted that the 'store semantics', one of the implementations, took some inspiration from Landin: "some of the arguments taken by the semantic functions are recognizably similar to the components of the SECDM [sic] machine".

Given that Milne showed that more implementation-like, or 'operational', models of languages could be formulated and usefully manipulated, the question may arise

as to why bother with the denotational parts at all. The answer to this mirrors the long-held beliefs in the mathematical semantics community about the limitations of operational models:

> Naturally we could regard operational valuations as "canonical" ones from which denotational valuations would be derived. There are three reasons why we choose not to do this. Firstly, operational values obscure the essence of programs by forcing their users to think in terms of intricate mechanical details rather than concepts. Nextly, operational valuations are extremely inconvenient to apply in proofs about programs, because they let one function be represented by several distinct parts of programs or pointers to programs. Lastly, operational valuations are often best related to one another by introducing denotational valuations as intermediaries.
> (Milne and Strachey 1976, p. 13)

However, Turner (2009, p. 6) argued against this point in a more philosophical sense. The operational approach was described by denotational semantics practitioners like Stoy as non-mathematical because it does not reduce to a set of mathematical objects as denotational semantics does: it is concerned with symbol manipulation or machine operations. However, standard mathematical systems are syntactic and concrete, argues Turner, but still refer to abstract mathematical entities. If, as the denotational semantics community posits, the same is to be said of computation, does that make programming languages discovered, rather than invented? Not quite, says Turner: instead, there is an abstract theory of operations from which programming language designers select constructs.

Milne's book is long and complicated, running over eight hundred and fifty pages. This is partly a result of containing three full descriptions of the same large language as well as plenty of exposition, and partly because the mathematics is detail and complex, with a dense notation. Milne acknowledged this:

> Those who are daunted by the prospect of reading this book should reflect on the feelings of Mollie Phipps, who typed most of it. [...] detailed suggestions about the entire book were made by Claire Milne.[39] In fact one might say of this book that without Dana Scott it would not have been written, without Mollie Phipps it would not have been typed, and without Claire Milne it would not have been read.
> (Milne and Strachey 1976, p. 9)

[39] Robert was Claire's husband.

Reminiscing about the book, (Milne 2000, p. 81) remembered that the actual task of writing was particularly difficult:[40]

> Even the book relied on handwriting: each page was overtyped several times, with different fonts fitted to a device called a "typewriter" (a keyboard mechanically linked to a memoryless character-by-character printer), before up to fifty script characters were written in by hand (my wife's or my own).

The lengthy experience of two or three years working solidly on the book meant that Milne returned to the topic only intermittently in the next forty years (Milne 2016). Nevertheless, it remains an excellent if dense reference on the deep theory of the relationship between language descriptions and implementations. Scott (1977, p. 635) spoke highly of it, particularly the thoroughness of mathematics used all the way through the implementation levels:

> What is important about the book is that it pushes the discussion of a complex language through from the beginning to the end. Some may find the presentation too rigorous, but the point is that the semantics of the book is not mere speculation but the real thing, it is the product of serious and informed thought; thus, one has the detailed evidence to decide whether the approach is going to be fruitful. Milne has organized the exposition so one can grasp the language on many levels down to the final compiler.

Not long after Milne's rewrite was published, another book on the subject was presented, written by Stoy (1977). This was much more like a textbook, and serves as the standard reference for introduction to the topic. The material was first developed during a long period lecturing on the semantic method at the PRG and then during a sabbatical year at MIT from 1973 to 1974. Stoy had been invited by Bob Farno to teach on mathematical semantics, and this material was put together with some of Milne's results for implementations to form the book (Stoy 2016b).

Stoy's book was called *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*, showing that by this time, the name 'denotational' was in use. The term is used because the emphasis of the approach is on defining which values are *denoted* by various parts of programs (Stoy 1977, p. 13). The introduction

[40] Jones remembers (personal communication, 2016) that Stoy had his own particular method of dictating mathematical expressions to Phipps: one noteworthy aspect is that he would pronounce 'bra' for an opening parenthesis and 'ket' for its closing counterpart.

to the book was written by Scott and provides some historical remarks. The rest of the book proceeds to look at motivations for studying formal semantics and outlines the three major approaches (operational, denotational, and axiomatic) before going on with details of the denotational approach. Stoy starts simply with numerals, moving on to functions and lambda calculus, before introducing domain theory and applying it to give semantics to increasingly complex languages. These start with lambda calculus, then flow diagrams, before introducing states, environments, jumps, and finally assignment. Increasing complexity in this way suggests an audience more of mathematicians and logicians than programmers, the latter of whom would likely understand assignment a lot better than lambda calculus. The final chapter of the book looks at 'non-standard semantics' and different implementation levels, and is drawn from Milne's work.

Discussing the use of different kinds of semantics, Stoy argued that it might seem at first glance that operational semantics is the best for a language implementer, denotational for the designer, and axiomatic for the programmer. The meaning of programs is not so clear in operational approaches, he said, because such descriptions are too much like "cranking a handle" to produce a value, hiding the meaning of the program as a whole. However, this judgement of operational semantics is somewhat reductionist, as such interpretive descriptions also demonstrate all the intermediate states in a way more natural to a programmer than the lengthy sequence of functional compositions used in denotational semantics.

Stoy argues that in axiomatic approaches, it is difficult to determine when the axioms hold, and for a language designer or implementer whether their language really does satisfy all the axioms.[41] An advantage of the axiomatic approach is that it is largely written in very similar notation to programs, which is significant as it may increase the ease with which a programmer can reason about their programs.

Ultimately, Stoy argued that denotational semantics was the best core semantics, as definitions can be translated between the different approaches. Axioms can be derived from a denotational model, bringing the advantage that there is a lot more

[41] Milne and Strachey (1974, p. 20) also made the same criticisms, in rather more stark terms:

> The techniques for proving properties of particular programs introduced by Hoare under the rather unjustified title of the 'axiomatic' method are considered promising by some; the 'axioms' of Hoare should be theorems in our semantic theory, but they are generally incorrect in their original forms, since implicit in them are several complex and boring restrictions on when they can be applied.

supplementary information about applicability and consistency.[42] Also, operational machines can be refined from denotational definitions, using Milne's methods. This notion of translatability was called the 'semantic bridge', as Stoy (2016b) explains:

> Christopher had the notion, which I set out in the final chapter of my book [...] of the 'semantic bridge'. He saw denotational semantics as the keystone, the arch at the top of the bridge. If you had defined a language denotationally, then you could use that definition on one side of the bridge to show the correctness of an implementation, which would have an operational semantics as its basis, and on the other hand, you could show the truth—you show that what we might call the 'proof rules', the axiomatic approach, that these were actually theorems based on the denotational definition. He saw the denotational semantics as the connection both to the operational semantics on one side and the axiomatic semantics on the other.

This focus on the primacy of denotational semantics should not be surprising given the book's author; indeed Stoy (1977, p. 23) acknowledged "We have, of course, been making a case for the denotational point of view, which is the principal subject of this book. Apologia for the other methods may be found elsewhere". Stoy (1981) also wrote a paper on proving the equivalence of methods of language definition, using an interpreter and a denotational semantics.

Scott (1994) noted in 1994 that denotational semantics worked very well for particular kinds of language, but struggled with others—many of which, unfortunately, became the popular kind.

> [Denotational semantics] has been rather concentrated on what you might call functional languages, whereas a lot of the interest and use of computer languages now is much more concerned with interactive languages and distributed computing, parallel computing, communicating processes of all kinds. And so there's an awful lot to do about communication protocols and transmission that makes a lot of difficult problems of interpretation. So by far the last word on semantics hasn't been said, because there's still a lot of phenomena that need explanation.

[42] A PRG doctoral student, George Ligler (1975) wrote a piece, published in ACM's second *Principles of Programming Languages* conference, which showed how this derivation could be made.

6.6 Reception and expansion

John C. Reynolds, an American computer scientist, had been present in Oxford in Autumn 1969 during the foundational period in denotational semantics, and had attended Scott's lectures (Scott 1977). This began Reynolds' interest in denotational semantics and led him to start considering thoughts about programming languages and how to improve them (Reynolds, in Jones et al. 2004). Much like Landin had with ISWIM, Reynolds put his ideas into a language called Gedanken (German for 'thoughts'), which was a simple typeless language permitting functional data structures and coroutines (Reynolds 1970). Reynolds' work in the early 1970s looked at using interpreters for higher-order languages (Reynolds 1972) and this grew into denotational work, by understanding what a programming language that combined higher-order procedures with assignment would be like. Reynolds proposed an alternative semantics of ALGOL he felt was more true to the spirit of the language, which made it clear that commands "don't change the shape of the state" (Reynolds 1981). However, his main interest was in the applications of the theory of denotational semantics, such as typing (Reynolds 1974b) and continuations (Reynolds 1974a), rather than the description of programming languages *per se*.

Reynolds contributed theoretical suggestions to Scott as well, including the use of disjoint sum domains, an important construction for typing, as appreciated by Scott (1972) in a letter to Robin Milner. Strachey was also very positive about Reynolds, writing "his work in this field is unquestionably outstanding, and he has shown a very clear insight into the conceptual problems involved" (Strachey 1972a).

Denotational semantics was greeted at first with some level of suspicion by the computing community as a whole. Common arguments, especially from the Vienna group, criticised its mathematical complexity. The Group was, however, clearly able to understand the concepts, as the following quotation from Walk (2002, p. 82) shows:

> It avoids the concept of computational steps and associates meaning to a language term in the form of a mathematical object, where the meaning of a composite term in general is built up from the meanings of the components of the term. Denotational semantics offers elegant explications for certain language concepts (e.g. procedures), less elegant ones for others (e.g. goto's), and in many cases makes it easier to construct mathematical proofs.

Lucas (1978) also wrote a summary of the denotational idea; by 1978, the Vienna Group had been working on their own approach to modelling programming languages in a similar style. He explained the most simple language constructs and how they are handled denotationally: assignment, sequencing, loops. Lucas criticised the already growing complexity with such simple language features, suggesting this made the method impractical for the majority of actual programmers: "it seems unrealistic and in fact unnecessary to require that each compiler writer be fluent in modern algebra" (Lucas 1978, p. 13). He compared the situation to foundations of mathematics and applied mathematics: "One would expect that the foundations are used to justify, once and for all, useful practical methods which in turn can be applied directly by the practitioner (not everybody who applies Fourier Analysis needs to be a specialist in the foundations of mathematics)".

At the fifth meeting of WG 2.2 in 1970, Lucas noted that there was some similarity in the way that denotational semantics and VDL modelled block and procedure structure. Somewhat petulantly, perhaps given the cool reception that the ULD definitions had experienced, he commented (in Walk 1970, p. 38) "What is really the difference to what we did? We have a representation of [the denotations for blocks and procedures] in store, which gives the possibility to throw them away". Scott explained that the cleaner separation of store and environment in denotational semantics made it much easier to prove when the store changes (and that the environment cannot). McCarthy was keen to ask about how denotational semantics could be used for the checking of compiler correctness, and Scott presented the diagram shown in Figure 6.13 (from Walk 1970, p. 47). He explained that you could show by methods of finite approximation that the diagram was symmetrical: the 'Values' would be equivalent.

Lucas (1978, p. 15) went on to criticise the general applicability of denotational semantics, noting that it was not able to handle parallelism because "With parallel processing, it seems referential transparency cannot be achieved as long as denotations of statements merely reflect the initial state/result state relation". Lucas also argued that the continuations method for handling jumps "[does] not closely correspond to the intuitive concept of the construct".

Dijkstra (1974b) too, had some complaints about the method, expressed in a letter to Bekič. Dijkstra largely focused on the huge mathematical mechanisms required which compared unfavourably, in his view, to the more pared-down axiomatic semantics.

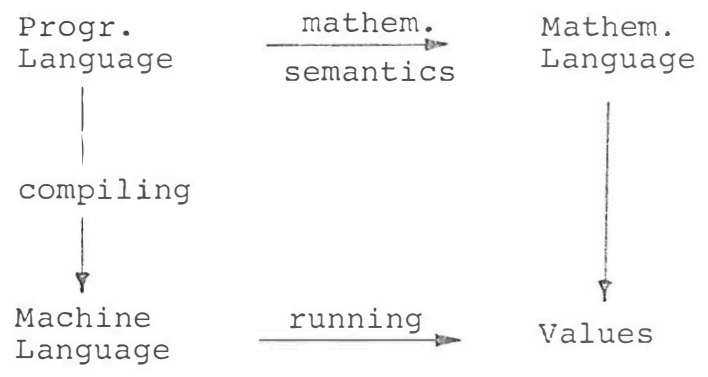


Figure 6.13: A diagram showing the utility of denotational semantics in determining compiler correctness, as presented by Scott to WG 2.2.

> It then came towards me as a view of computing that can be appreciated for its consistency, for its conceptual unification, but not without a few drawbacks, which I am beginning to regard as serious. The first one is the size of the mathematical machinery involved, compared to which the propositional calculus –and for an axiomatic definition now nothing more seems needed– is negligible. [...] The second drawback is that defining the semantics as a minimal fixpoint is a very indirect one –some people would even call it clumsy–: first one needs functional analysis in order to introduce the concept of a fixpoint, and after that lattice theory to formulate that it is not just any fixpoint, but among the fixpoints the minimal one.

Despite some negative reactions, denotational semantics came to be regarded as very important in the field of programming language description. Indeed, WG 2.2 delayed its fifth meeting from April to September 1970 partially in order to give the group time to digest the work of Scott and Strachey (Steel 1970). The ideas spread in popularity, in large part due to the paper of the Canadian Tennent (1976) which brought the ideas to the Americas. This paper provided an excellent overview of the approach as well as defining Reynolds' Gedanken language. It was one of the first publications to use the word 'denotational' and likely contributed to the popularisation of that term. Tennent (1977) also wrote a paper showing how the use of denotational semantics could help with language design, applying these principles to Pascal, in much the same way that Landin had written about using his semantics principles in language design.

By the first *History of Programming Languages* conference in 1981, McCarthy (1981, p. 183) was referring frequently to the hope of mathematical semantics for LISP: "LISP will become obsolete when someone makes a more comprehensive language

that dominates LISP practically, and also gives a clear mathematical semantics to a more comprehensive set of features". This shows how much the ideas had penetrated into computing: denotational semantics was being seen as a standard.

However, there was one area in which denotational semantics was struggling: modelling non-determinism. Handling multiple possible outcomes breaks up the homomorphism principle just as much as jumps do, and allowing commands to interleave breaks the flow of composition of commands' denotations. Milne did some work on using continuations to handle parallelism, but, as mentioned, it was unsatisfactory for proofs. Reynolds also experimented later in using continuations to handle both concurrency and jumps in his separation logic, but they could not be used for both at the same time (Reynolds 2004). Eventually, Gordon Plotkin formulated the power domain approach, which worked, but was described as 'nasty' by even denotational devotees such as Mosses due to its immense mathematical complexity.[43]

Another difficulty was non-terminating programs, such as operating systems, for which the concepts of initial and final state are not easy to determine. John Laski (1974) pointed this out in a letter to Strachey; Strachey's reply was that it could be done in a more operational style, and the connection with a denotational semantics could then be established using Milne's methods (Strachey 1974b). However, this method was clearly not ideal. (Dijkstra 1974b) also commented on the inability of denotational semantics to handle non-termination, noting that a talk by Reynolds had attempted to address the problem:

> John Reynolds started in Munich also with the operating systems and the airline reservation systems and after that two minute motivation...[44] he talked for the remainder of more than an hour about the logical difficulties of dealing with real numbers! It was a very nice and illuminating talk (at which I rejoiced that real numbers are no longer my problem!), but by the time it is suggested that one needs real number theory for dealing with operating systems and airline reservation systems, some hilarious mistake has been made.

Whether or not denotational semantics ultimately achieved the penetration for which Strachey had hoped, his influence on computing is undeniable. Michie (1971, quoted in Campbell-Kelly 1985, p. 40) wrote:

[43] See more in Section 7.4.

[44] Ellipsis present in Dijkstra's original letter.

> Today an "invisible college" of programming theory exists throughout the Universities of Britain. Almost every member of this "college" was guided along the path at some stage by Strachey's direct influence. Developments of theory may in the long run prove decisive in helping to clear the hurdles of software engineering which still lie ahead.

Schmidt (2000) agreed with Michie's predictions, and wrote that Strachey's works influenced many important trends in theoretical computing, including type theory, structural approaches to semantics—especially SOS (see Section 7.4)—and semantic calculi such as Milner's CCS.

By the time of Strachey's death, the PRG had become a successful and respected department. Hoare (2000, p. 71) remembers that he came to Oxford in 1977 to lead the group partly because he recognised how important it was to learn about denotational semantics. He also quoted a piece written by Strachey showing the latter's commitment to keeping in touch with both theory and practice:

> When I arrived in Oxford, and sat in Strachey's office on his chair and at his desk, I found in the top right hand drawer a brief report which he had written on his research. It contained the following memorable declaration:
>
> > It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen.

Hoare continued to explain that the same principles guided him, and the PRG kept flourishing along these lines.

In conclusion, Strachey was extremely influential in computing, although not this was not always outwardly obvious. His ideas on programming languages that came from CPL went on to influence C, ALGOL 68, and many other widely-used languages; functional programming, in particular, owes a great debt to Strachey and

his ideas.[45] Denotational semantics had a great impact on the whole field of language description, and ideas from it remain popular today. Furthermore, reactions to its complexity or certain other features sparked other valuable and important approaches. Some of these are discussed in the following chapter.

[45]For example, David Turner completed his doctorate at the PRG trying to create an implementation for Strachey's PAL language. It was too hard, but the ideas ultimately became the famous Miranda.

CHAPTER 7

Related developments

The main bulk of the present account is focused on the works of the IBM Laboratory Vienna and the Oxford Programming Research Group, as given in the previous two chapters. However, there are other stories worth telling because they grow out of those developments, provide interesting comparisons, or serve to flesh out the entire piece in some way. Section 7.1 describes what might be considered a dead end in both language definition and design, due to a complex description technique. Section 7.2 provides extra details on a key meeting arena where the material in the previous chapters was presented, discussed, and perhaps refined. Section 7.3 explores developments in Vienna that show the cross-influence of the groups discussed in earlier chapters. Section 7.4 outlines the development of a description technique intended as a reaction to a lot of the complexity of earlier approaches—one which still survives today as an important and usable method. A timeline of the major events detailed in this chapter can be found in Figure 7.1.

7.1 ALGOL 68 and two-level grammars

Following the success of ALGOL 60 and its various reports, as described in Section 2.3, a clear desire emerged amongst the ALGOL devotees to provide a successor language. The ultimate product, ALGOL 68, was ambitious in the scope of the language, and the description technique was a powerful tool that grew out of the problems of specifying context-sensitive description. However, the language was a

Figure 7.1: Some important events in the stories of ALGOL 68, IFIP WG 2.2, VDM, and SOS.

Early 1964	WG 2.1 begins working on 'Future ALGOL'
May 1965	TC-2 accepts a proposal to form a Working Group on formal language description
Oct 1965	Van Wijngaarden's MR76 proposal introduces two-level grammars and orthogonal design
Sep 1967	First meeting of WG 2.2
1968–1969	Bekič spends a year working with Landin in London
Feb 1968	MR93, a draft for ALGOL 68, is circulated to WG 2.1
Dec 1968	Dissenters resign from WG 2.1 and sign a 'Minority Report'
Dec 1968	ALGOL 68 is approved as an IFIP publication, codifying two-level grammars and introducing context conditions
Apr 1969	Hoare presents axiomatic semantics at a WG 2.2 meeting
1969	Burstall uses relations in a description of ALGOL 60
Early 1970s	IBM begins working on Future System (FS)
Mar 1970	Henhapl and Jones' 'exit' mechanism models jumps in VDL
Early 1971	WG 2.2 dissolves, partly due to a lack of purpose
1971	Milner develops LCF at Stanford
1972	Plotkin visits Syracuse, spending time with Reynolds
Aug 1972	Allen, Chapman, and Jones write a 'functional' description of ALGOL 60
Early 1973	The Vienna Lab begins developing a PL/I compiler for FS
1973	Revival of WG 2.2 under Neuhold with broader remit
1973	Gordon's PhD thesis introduces configurations
Dec 1974	The Vienna Lab writes a VDM formal description of PL/I
Feb 1975	FS is cancelled and the VDM work is nearly lost
1975–1976	Plotkin works out power domains
1978	A book on VDM is published, edited by Bjørner and Jones
1981	Plotkin's 'Aarhus notes' lay out SOS

mixed success at best, and this section explores the background to ALGOL 68 and considers the presentation method's successes and problems.

7.1.1 After ALGOL 60

Ownership of ALGOL was transferred to IFIP Working Group 2.1 in 1962 with van der Poel as the chair. After a short period working on an IFIP subset and I/O procedures, the group's focus shifted to successor languages. Progress began in early 1964 under the title 'Future ALGOL' and the topic was the main focus of the two meetings that year (Utman 1964a,c). At this time, WG 2.1 had a large list of members and attendees representing the great and good in programming languages; Figure 7.2 shows a collection of these. The group included academics and industrial workers; typical expertise included compilers, language development, systems, and also formal language description techniques. The nature of the group was politely argumentative and it had a reputation for backhanded jokes:

> A characteristic trait of the mood and spirit of WG 2.1 was the famous extension of the voting possibilities [...] from yes, no and abstention to a fourth choice: *I did not understand the question*, the semantics of which was essentially that the voting member for tactical reasons pretended not to understand the subject of the vote.
> (Zemanek 1981, p. 14)

The implication of this was that the question was too poorly worded. Another example of this awareness of the group's own "sometimes silly behaviour" is visible in the "working rules" proposed by Ingerman[1] (in van der Poel 1986, pp. 374–5):

1. Whenever a point shows a danger of being clear, it shall be referred to a committee.
2. A subcommittee shall prepare two or more contradictory clarifications of the point referred to it.
3. These clarifications shall be reported by the individual members of the committee, no committee report having seen[2] achievable.
4. The clarifications shall be discussed until one of them is in danger of being understood, at which point return to 1.

[1] According to van der Poel (1986, p. 378) Ingerman was known to be "a maker of pun and fun"; another example witticism is "This language fills a much needed gap".

[2] This apparent typographical error (the correct word is likely 'been') is copied unaltered from van der Poel's text.

TABLE 3 WG2.1 members active in the original design of ALGOL 68

Fritz Bauer	Techn. Hochschule Munich
Edsger Dijkstra†	Tech. University Eindhoven
Fraser Duncan†	National Physical Lab., UK
Tony Hoare†	Elliot Automation, UK
P. Z. Ingerman	RCA, Camden NJ
John McCarthy	Stanford University, CA
J. N. Merner	General Electric, Phoenix Arizona
Peter Naur‡	A/S Regnecentralen, Copenhagen
Manfred Paul	Techn. Hochschule Munich
Willem van der Poel	Techn. Hogeschool Delft, Chairman of WG2.1
Klaus Samelson	Techn. Hochschule Munich
Gerhard Seegmüller†	Techn. Hochschule Munich
Aad van Wijngaarden	Mathematisch Centrum, Amsterdam
Mike Woodger†	National Physical Lab., UK
Jan Garwick†	Oak Ridge National Lab., Tennessee
Brian Randell†	IBM, Yorktown Heights NY
Niklaus Wirth‡	Stanford University, CA
Peter Landin	Univac, New York NY
Hans Bekic	IBM Lab., Vienna
Doug Ross	M.I.T., Cambridge Mass
W. M. Turski†	Academy of Sciences, Warsaw, Secretary of WG2.1
Barry Mailloux	Mathematisch Centrum, Amsterdam
John Peck	University of Calgary
Nobuo Yoneda	University of Tokyo
Gerhard Goos	Techn. Hochschule Munich
Kees Koster	Mathematisch Centrum, Amsterdam
Charles Lindsey	University of Manchester, UK
Michel Sintzoff	MBLE, Brussels

Figure 7.2: Members of IFIP WG 2.1, as recorded in Lindsey's (1993, p. 7) history of ALGOL 68.

At the meeting in Baden-bei-Wien in 1964, two new versions of ALGOL were proposed, and given the code names 'ALGOL X' and 'ALGOL Y' (Utman 1964a, p. 12). Once called ALGOL 65 and ALGOL 70, the new names came to avoid committing to any particular date of publication. ALGOL X was to be an iterative development of ALGOL 60 and ALGOL Y a radically redesigned and improved language. In fact, there was even some thought that ALGOL Y would be a "*metaprogramming* language: a language for (effectively) defining programming languages" (Meertens 2016). Van Wijngaarden's (1962) Generalised ALGOL was taken as the starting point for both languages, and van Wijngaarden himself was confident that deciding to develop ALGOL X along those lines would be quick—"Within a year" (Utman 1964a, p. 12).

7.1.2 Towards ALGOL X

Work began in earnest in May 1965 at the Princeton meeting (Woodger 1965). As well as van Wijngaarden's approach, there was significant interest in the EULER language (Wirth and Weber 1966a,b), particularly its robust approach to typing. Wirth, Hoare, Seegmüller, and van Wijngaarden were charged to provide draft languages for the next meeting, to be held in St. Pierre de Chartreuse, in October 1965 (Randell 1965). Wirth's (1965) proposal was written over the summer while

on sabbatical at Mathematisch Centrum in Amsterdam, and Hoare made some suggestions to improve the concept, resulting in a joint proposal (Wirth and Hoare 1966) distributed significantly in advance of the St. Pierre de Chartreuse meeting. In contrast, van Wijngaarden's (1965) 'Orthogonal design and the description of a formal language' (commonly referred to as 'MR76', its publication number) was only circulated a week before the meeting. These were the two main proposals; according to Lindsey (1993, p. 8), Seegmüller's contribution was not a serious contender. Generally, the response at the meeting was that Hoare and Wirth's contribution contained good language concepts but was too informal; MR76 was desirably formal but the language was insufficiently advanced (Meertens 2016).

MR76 introduced some concepts which would go on to be very important in the upcoming ALGOL 68 story. One was a 'two-level grammar', which van Wijngaarden (1965, covering letter) liked because it "enables the design of a language to carry much more information in the syntax than is normally carried". The other was the 'orthogonal design' of the report's title: a small number of relatively simple language concepts were described, which could then be combined in any way to form a program. Of this, van Wijngaarden (1965, covering letter) wrote "I should like to see the definition of a language as a Cartesian product of its concepts". This notion was an extension of his 'generalist' attitude towards ALGOL 60, as discussed in Section 3.3: restrictions should not be made to a language that suppress the equal treatment of all its concepts. Turski (1981, p. 425) added later:

> It is hard not to admire the simplicity and completeness of this approach: if carried through, it guarantees the absence of annoying exceptions that plagued programming languages of that epoch. It also guarantees that no ad-hoc feature (concept) could be *easily* added to the language - the combinatorial complexity of consequences of such a move had to be followed through first.

This focus on the orthogonality concept and the use of two-level grammars are difficult to disentangle. The grammars can be seen as the extension of van Wijngaarden's strongly symbolic approach in his previous work: in a sense he was avoiding the question of applying an abstract semantics through a very powerful syntactic generator. Orthogonality then allowed for any syntactically-valid combination of semantics to be semantically useful, which skipped a lot of the dynamic error checking required in more conventional and restrictive languages. A potential germ of the 'two-level' concept for the grammars is the comment made by Hoare

on van Wijngaarden's *Formal Language Description Languages* paper (van Wijngaarden 1966b).[3] Hoare asked why the rules for the preprocessor of that approach could not be part of the ruleset carried by the processor, which van Wijngaarden acknowledged was possible. The closer mixing of these two concepts could well have been part of the inspiration for the two-level grammars.

While the metalanguage presented in MR76 was innovative and interesting to the members of WG 2.1, the language defined was less impressive; Lindsey (1993, pp. 8–9) later wrote it was "nothing special", and that the method of presentation made it "exceedingly hard to read (no examples, no pragmatics, and only 26 one-letter metanotations of not much mnemonic significance)". Nevertheless, WG 2.1 adopted the style, perhaps because it was the only proposal on the table which showed strong hints of a fully mature approach. The group was badly in need of something that looked like a clear path forwards; van der Poel (writing in 1986, p. 374) had criticised the group's lack of direction earlier that year, as there had been far too much argument over what he felt were trivial matters: "When we go on with fighting over details first and when the main lines of the issue of ALGOL X are not fixed in principle, then ALGOL X will ways, like the camel, be a horse designed by a committee".

At WG 2.1's sixth meeting, in October 1965, it was decided that Hoare, Wirth, Seegmüller, and van Wijngaarden should form a subcommittee to prepare a draft report. This should contain the language ideas discussed and agreed upon by the whole group, and be written in the formalism of van Wijngaarden (Turski 1981, p. 420). At the end of the meeting, the subcommittee announced that van Wijngaarden would write the draft and the other three would consider it and make appropriate suggestions, before circulating the document to all of WG 2.1 well in advance of the next meeting. Unfortunately, van Wijngaarden had significantly underestimated the time it would take for him to finish the draft, and the next meeting was delayed by six months (Lindsey 1993, p. 9). The delays annoyed Hoare and Wirth, who were now also having major disagreements with van Wijngaarden over the formalism used to define the language, as well as certain aspects of its content including the parameter mechanism. The two withdrew from van Wijngaarden's efforts and took their own ideas into a new ALGOL offshoot, which they called ALGOL W (Peláez Valdez 1988, p. 194). The promised draft report eventually appeared in October 1966 with van Wijngaarden as the only WG 2.1 author: he was now assisted by Barry Mailloux, his PhD student (van Wijngaarden and Mailloux 1966).

[3] See Section 3.3.

7.1.3 ALGOL 68 appears

The author list expanded further by the production of the final draft report presented to WG 2.1; now featured were John Peck, on sabbatical at Mathematisch Centrum from Calgary, and Charles Lindsey (van Wijngaarden et al. 1968). This document, known commonly as MR93 'Draft Report on the Algorithmic Language ALGOL 68', circulated in February 1968 to a response of "shock, horror and dissent" within Working Group 2.1 (Lindsey 1993, p. 11). In May 1968, a colloquium was held in Zurich to celebrate ten years of ALGOL, and MR93 came under fire. According to a report in the ALGOL Bulletin (quoted in Lindsey 1993, p. 12), the draft report was "attacked for its alleged obscurity, complexity, inadequacy, and length, and defended by its authors for its alleged clarity, simplicity, generality, and conciseness". At that meeting, Naur (1968, p. 60) in particular spoke in very critical terms:

> [MR93] sets out to provide the ultimate in formality of description, a point where Algol 60 was strong enough. In doing so, MR93 sets a new record of lack of appeal to human readers. In fact, it makes an attempt to create, not only its own special terminology, but a linguistic universe wholly of its own, and requires, to quote Mike Woodger, that the reader will have "his normal reading instincts .. thoroughly suppressed".

Naur laid the blame at the door of IFIP, its bureaucracy, and desire to create a "monument".

The draft report on ALGOL 68 caused a huge rift in WG 2.1. The group had previously been, according to Zemanek (1981, p. 15), a "crew of old friends and enemies who enjoyed meeting and fighting", but these fights turned acrimonious. An important point of contention was whether the language should be called 'ALGOL' and receive the blessings of IFIP, with the implication that all of WG 2.1 had contributed to and was satisfied with the language. Publication as an official IFIP language would have granted van Wijngaarden's production much recognition and considerably increased its likelihood of uptake. Many members of the working group were concerned the language was not ready for this due to its obscure presentation, so van Wijngaarden arranging to give an invited lecture at IFIP Congress in Edinburgh in August 1968 and calling the language 'ALGOL 68' caused quite some upset (Randell 2016). Zemanek, in his role as chair of TC-2, felt this as a very uncomfortable time; although he did not like ALGOL 68 himself (Neuhold 2016), he was aware that a turning point would come very soon for WG 2.1. He laid out

an ultimatum to the group in October 1968 making it very clear they would soon have to come to a decision over whether to recommend the language be published by IFIP (Zemanek 1981, p. 20).

It should be noted that not every member of WG 2.1 was opposed to the sponsoring of van Wijngaarden's language. Of course the team working on creating the language was positive, but some members had more balanced views. Landin (1968) was able to follow the method of description and wrote an introductory piece criticising certain aspects and praising others. Bekič (1968), although sharing concerns with the critics, thought the group should have given the work more of their attention. He particularly deplored that the critics had only spoken up upon publication of the full draft report as though its content and appearance was a surprise to them—despite having had ample opportunity to view and comment on drafts.

Duncan (quoted in van der Poel 1986, p. 384) made a balanced assessment when he wrote to van der Poel, the chair of WG 2.1:

> I think it is no exaggeration to say that a widespread opinion is that the document itself is extremely difficult to begin to understand (and unnecessarily so), but that inside it there may well be a good language trying to get out. *Maar niemand wil een kat in de zak kopen* [But nobody wants to buy a cat in a bag].[4]

By July 1968, and the Berwick meeting of WG 2.1, just before the IFIP Congress at which van Wijngaarden was due to present the language, politics and jockeying had overtaken the entire agenda, according to Lindsey (1993, p. 14). This ultimately led to a minority report (Dijkstra et al. 1969) being issued to TC-2 alongside MR93, with the intention it be published alongside the language if indeed it did become an IFIP publication. The text of the report essentially indicated that some members of WG 2.1 did not like the language and were not endorsing it—although in somewhat sharper terms, as would be expected from a piece originally drafted by Dijkstra. In December 1968, at the Munich meeting of WG 2.1, the report was read out by Randell who remembered "Somebody afterwards said 'You sounded as though you were giving an oration at a funeral.' You know, it was that tough" (Randell 2016).

The authors of this minority report resigned from WG 2.1, and some sent very strong resignation letters to van der Poel. Garwick (1969), for example, wrote "Under no circumstances must my name be used in connection with ALGOL 68 in such

[4]Translated from Dutch by the present author.

a form that it can be interpreted as my support of this effort". Duncan (1969) modified his official resignation form so that it no longer read "IFIP WG 2.1 on ALGOL" but "IFIP WG 2.1 on ALGOL 68". Ingerman (1969) even included a poem in accompaniment:

> If by chance your eye offend you,
> Pluck it out, lad, and be sound:
> 'Twill hurt, but here are salves to friend you,
> And many a balsam grows on the ground.
>
> And if your hand or foot offend you,
> Cut it off, lad, and be whole;
> But play the man, stand up and end you,
> When your sickness is your soul.
>
> A. E. Housman

The WG 2.1 leavers formed a new Working Group, 2.3, on 'programming methodology'. Peláez Valdez (1988, p. 195) wrote "The Minority Group felt that the emphasis on language design was hindering a simpler, clearer approach to programming", and this was what the new group was to handle. The unfortunate response to ALGOL 68 also meant that ALGOL Y was quietly dropped. Until the publication of ALGOL 68 discussion of that language had dominated, and afterwards, there was very little appetite for another WG-designed language (Meertens 2016). However, following the official Report (van Wijngaarden et al. 1969), WG 2.1 did manage to produce a Revised Report, with more authors contributing, including those who had been working on implementations (van Wijngaarden et al. 1977).

Despite the strong opinions of these members, there was enough support in the majority of WG 2.1 for a vote to pass requesting TC-2 to submit the language to the IFIP General Assembly for publication as an official IFIP product. TC-2 did not bundle the minority report in with the language when passing it to the Assembly, although their covering letter shows that the committee was very aware of the minority opinion: "Therefore, this Report is submitted for publication as representing one of the possible approaches to the subject [i.e. WG 2.1's charter to produce a successor to ALGOL 60], rather than in the spirit of a final answer" (TC-2 1968).

When the language was published, Zemanek, in his role as chair of TC-2, wrote a covering letter summarising the views of the minority report (Teufelhart 1969, p. 4).

What went so wrong with ALGOL 68 that nearly half of WG 2.1 felt they had to publicly distance themselves from it? One major criticism was that the meta-language was very dense and hard to follow. The minority report (Dijkstra et al. 1969) reads "We regard the high degree of inaccessibility of its contents as a warning that should not be ignored by dismissing the problems of 'the uninitiated reader' ". Consequently, argued the minority group, it was almost impossible to determine the report's correctness. Other members of the group did not like the language under definition itself. Turski (1981, p. 420) later wrote:

> The discussions about which features to consider [...] were very much based on two sources of ideas: specific [...] programming constructs and [...] advances in computer hardware. ALGOL X was then to be both an abstract language and a machine-oriented language. It was not recognised then (as it is not widely recognised today) that such a mixed approach leads to a cloudy design.

Some of the problems may have been caused by van Wijngaarden (quoted in Turski 1981, p. 422), who regarded himself as the "party ideologist". Lindsey (1993, pp. 16–7), a staunch supporter of ALGOL 68, admitted

> All the dissenters shared an exasperation with Van Wijngaarden's style, and with his obsessional behaviour in trying to get his way. He would resist change, both to the language and the document, until pressed into it. The standard reply to a colleague who wanted a new feature was firstly to say that it was too late to introduce such an upheaval to the document, then that the colleague should himself prepare the new syntax and semantics to go with it. But then, at the next meeting, he would show with great glee how he had restructured the entire Report to accommodate the new feature and how beautifully it now fitted in. Moreover, there were many occasions when he propagated vast changes of his own making throughout the document.

Łukaszewicz (1986, p. 296) agreed; although he saw this as something of a benefit, writing:

> I had an opportunity to observe him as he was presiding over one of the meetings of the Algol 68 group in Warsaw. On that occasion rigour, coherence of ideas and orthogonality (a concept invented by Aad) ruled

> absolutely, and it was obvious that Aad would rather go to the stake than renounce his principles. There were those who complained of his obstinacy, but in my opinion it was just that which ensured that Algol 68 has not turned out to be a conglomerate of brilliant but incoherent ideas—a frequent result of projects worked out by committees—but has become a harmonious whole.

The historian Mike Mahoney (2002, p. 28) has written about the concept of 'agenda' as a useful tool for historically analysing scientific groups. The idea is that the agenda of a field (i.e. the important problems, what is to be done to address them, and how that is done) is the defining feature of a field. Conflicts in a field are often over the relative importance of agenda items, and the emergence of a new field is codified when a group gains the power to set its own agenda. This period in WG 2.1 shows a great deal of fighting over exactly what the agenda of the working group should be—and indeed the agenda of the programming language they were trying to produce. The splitting off of the minority group shows those members realising the agendas of the group and ALGOL 68 did not align with their own, and taking the decisive step to set their own agenda by forming a new working group.

A further consequence of the experience of the "debacle and near-fiasco" of ALGOL 68 (Ross 1968) was that it scared other working groups away from trying to create products. Ross wrote that the lesson to be learnt was "that the proper role for international working groups is to provide a forum for mutual stimulation and cross-fertilization, and not to produce a 'product' ".

7.1.4 ALGOL 68 and its description

A full discussion of the language that caused such consternation is out of scope for the present work, but it is worth providing an overview. Like PL/I and CPL, ALGOL 68 was a very complex language, although in a different way to the looseness of the former and large and carefully labelled latter due to van Wijngaarden's orthogonal design approach. This complexity grew over time: at first, the language had relatively few core concepts and according to Lindsey (1993, p. 13) WG 2.1 had few complaints. However, committee members kept suggesting new language concepts and the requirement imposed by orthogonality for a way to combine each new feature with every existing feature led to the language growing exponentially. The powerful typing system (called 'modes') and the advanced parameter mecha-

nisms present in ALGOL 68 reflect the goals van Wijngaarden had for the design of ALGOL 60, as seen in the contributions listed by Naur (1981b, pp. 110–3).[5]

The metalanguage seemed to cause more problems. The approach was developed from van Wijngaarden's earlier work as discussed in Section 3.3 but with some significant alterations. The notation used was a two-level or W-grammar (W for Wijngaarden) which is a system for producing program statements by using repeated application of rules to alter text.[6] The grammar has two important concepts: 'notions', things that can be used in a program, and 'meta-notions', aspects of the grammar which can produce notions. All are expressed as strings and so can themselves be manipulated by the grammar at will. A powerful and flexible grammar was required by the rich type system of ALGOL 68; all types had to be expressible in every context and they therefore needed a second-level representation in the grammar (this was the string 'MODE'). Hoare (1968) wrote a letter to van Wijngaarden praising the handling of these type constraints, but added "the language description is unnecessarily difficult and obscure, without sufficient compensating gain in rigour".

Two-level grammars had a great deal of power and expressiveness. Sintzoff (1967), one editor of the Revised Report, proved that the grammar could produce any recursively enumerable set. Van Wijngaarden even proudly told Jones that his grammars were Turing-complete; Jones' reply was that it was a shame: this was more powerful than needed for the particular task.[7] Later, van Wijngaarden (1979) demonstrated this by working on a system to use two-level grammars for writing and describing programs directly, without using an intervening programming language, in a paper entitled 'Languageless programming'.[8]

As has been mentioned a number of times in the present work, one potential use for language descriptions is to enable the proving of properties about programs written in that language. This was not something which interested van Wijngaarden at all, as shown in the following exchange from WG 2.2:

> Scott: Let us ask van Wijngaarden about his experience in proving programs.

[5] These are also discussed in Section 3.3.

[6] For a good introduction to the system, read Pagan's (1981) book.

[7] Personal communication with Jones, 2017.

[8] The paper is somewhat frank about the problems of programming languages, and as it used ALGOL 68 for its examples, van Wijngaarden carefully adds "The critical remarks on ALGOL 68 should not be misinterpreted; remarks on other languages might very well have been much more critical indeed".

> van Wijngaarden: An algorithm always does exactly what it says it does.
> (Walk 1969b, pp. 13–4)

One very important contribution of the ALGOL 68 definition was the notion of 'context conditions'. A large portion of the original Report (van Wijngaarden et al. 1969, Section 4.4) described these; the idea was to delineate the subset of all syntactically valid programs that constituted 'proper' programs. This involved catching errors such as variables accessed outside of their scope, or mode of use of a variable not matching its declaration. This task, analogous to the 'static semantics' of Oxford denotational definitions, was described informally with words in the first Report; the Revised Report brought it into the syntax, utilising the power of two-level grammars (Lindsey 1993, pp. 45–6).

The semantic model used in the ALGOL 68 description was, as noted by Stoy (1977, p. 12), inherently operational "underneath its inimitable description method". This model represented a hypothetical computer with a collection of objects that could be either 'internal' (e.g. values) or 'external' (i.e. from the metalanguage); and the relationships between them, such as 'accesses', 'refers to', 'is a subname of', and so on. The actions of the model involved the creation of objects and the modification of relationships.

MR93 and the initial Report were written with 'steps' describing the semantics of each particular language concept in terms of this model; as Lindsey (1993, p. 49) notes, this made a program's description essentially a series of go tos. This can be seen as flowing from van Wijngaarden's (1966) FLDL paper, the approach of which focused on the manipulation of the processor's control. Revisions to the ALGOL 68 Reports used more of a 'structured programming' approach, with cases and tests governing the application of rules. The Revised Report (van Wijngaarden et al. 1977) also had a lengthier description of the underlying semantic model in order to shorten the semantics of the language constructs, as well as far more 'pragmatics'—explanatory comments. Part of this was to ensure the reader started with the correct mental model, as Lindsey (1993, p. 52) explained: "If a reader approaches some kind of mathematical formalism with the wrong preconceived idea of what is to be presented, he can waste hours before his internal model becomes corrected. I therefore regard the principle purpose of pragmatics to be to 'preload' the reader with the appropriate model".

Turski (1981, p. 428) made the interesting point that in a sense, the ALGOL 68

semantic approach was the inverse of van Wijngaarden's previous processor, which broke a program down into very basic notions whose meaning was then processed into a list of truths:

> The ultimate complexity of the ALGOL 68 report comes from the fact that a very low level operational semantics was employed not to verify an implementation of otherwise understood notions, but to *define* constituents of freely ('orthogonal design'!) constructible notions. Now, it was up to the human reader to act as the van Wijngaarden's processor—in reverse.
> (Turski 1981, p. 428)

All the semantic description was written in English words rather than a mathematical formalism, but it used very careful choice of words with very particular meanings, and so could be regarded almost as a formal notation with lengthier identifiers.[9] This approach was described by Mailloux (quoted in Lindsey 1993, p. 50) as "syntax-directed English". However, a problem of using English was that the large number of similar but distinct concepts in the language had to receive names in a way that reflected this, resulting in a lot of essentially analogous words being used with different meanings. Randell (2016) described this as follows:

> So his method of language definition, which involved very rigorous use of English, with huge amounts of consulting of dictionaries and thesauri, and the like, produced a very legalistic English—legalistic at the total expense of readability.

Another problem with the use of English for the description was pointed out by Hoare (1968), who noted "it seems inappropriate for a language intended for international use, particularly since it is difficult to translate accurately into other languages".

The negative response to ALGOL 68 makes an interesting counterpoint to the ULD-III descriptions, whose frequently-criticised size and complexity was a reflection of the size and complexity of PL/I. Instead, ALGOL 68 was attacked for its metalanguage, drawing perhaps a comparison with Strachey's focus more on defining CPL than that language itself. It is, however, certainly true that the complexity of the

[9] Turner (2009, p. 5) made the observation that any attempt to provide a precise description using natural language will ultimately require such narrow and unambiguous word use that it essentially becomes formal.

orthogonal design of ALGOL 68 was always going to require a detailed description technique. Bekič (1968) recognised this in a letter written to van Wijngaarden in which he sympathised with the reaction the complex document had faced. Randell said, in an interview with Peláez (1988, p. 194), "van Wijngaarden at that time I think was rather more interested in his method of language definition than the actual language he was defining". This was not something that van Wijngaarden denied, as the following exchange from WG 2.2 shows:

> Ingerman: How did the language described influence the description technique?
>
> van Wijngaarden: It was rather the other way round.
> (Walk 1968, p. 26)

Due in part to the complexity of its description, ALGOL 68 did not enjoy the success of ALGOL 60; however it did have some influence in academia, particularly with its expressive and flexible type system (Meertens 2016). Van Wijngaarden was honoured with the IEEE Computer Pioneer Award in 1986 for his work on ALGOL 68 (Verrijn-Stuart 1987). Although he carefully avoided praising the language itself, Zemanek (1981, p. 21) wrote with respect of the "incredibly concentrated and immense amount of work done by him and his collaborators". Alberts (2016, p. 14) summarised the language's impact, also arguing that the notion of agenda was at the core of the problems with the ALGOL 68 effort:

> As a piece of art, ALGOL 68 had, and still has, a small community of admirers. As a programming language, as a tool, it was a failure. It realised the ALGOL agenda set in 1958, in the most beautiful and least practical way. Ten years on, however, the agenda had shifted to software engineering with new pressing questions and a new generation of bright spirits.

7.2 IFIP's Working Group 2.2

The *Formal Language Description Languages* conference, as discussed in Chapter 4, was crucial for many reasons already covered, but also led to the creation of an IFIP working group in the area of the formal description of programming languages. This was the second WG created under the auspices of TC-2 on programming languages, and was consequently called WG 2.2. It brought together many people working in the field and was a forum for the discussion and sharing of ideas, although unlike

WG 2.1, with its control of ALGOL, it did not have the charge to create a particular product. This contributed to the less focused nature of its meetings, and lack of clear direction.

7.2.1 Origins

The first proposal for an IFIP working group on "meta languages" under TC-2 came in August 1962 (Utman 1962b, p. 3), but at that time it was not seen as a sufficiently *international* idea to come within IFIP's remit, a point particularly espoused by Wilkes. IFIP had a charge to direct projects that had this strong international flavour and at the time there were not enough people working on programming language description across the globe. The matter was discussed again at TC-2's third meeting, in September 1963, now with the proposed theme of "specification languages" (Utman 1963, p. 7). Zemanek, the chair of TC-2, had conducted a survey of experts to determine whether such a working group would be feasible, but had received sufficiently mixed responses that he felt it was not a justifiable use of IFIP's resources. Bauer commented that the August 1963 *Working Conference on Mechanical Language Structures*, already discussed a number of times in the present account, showed there was some development in the field, but more was still needed—and another survey of experts could be done in the future.

By the fifth meeting of TC-2, in May 1965, the topic came up again, and this time the positive experience of FLDL lent some credence to the notion that the area was worthy of IFIP's attention (Utman 1965, p. 12). Zemanek was interested in expanding TC-2's operations, and this seemed an appropriate field. Tom Steel (in Utman 1965, p. 12) suggested that a working group on "Language Definition" be formed, asserting "IFIP must help provide, find or develop description languages". There was some discussion over whether this would fit into the remit of WG 2.1, but fears that it would provide a distraction from their important task of defining the new version of ALGOL countered that idea. Arguments at the TC-2 meeting over exactly how international the new group would be and precisely what useful outcomes it could hope to contribute led to a contested vote, but an eventual result of six votes for, four against, and three abstentions provided enough support for plans to begin (Utman 1965, p. 13). Auerbach, IFIP's president, was not terribly hopeful, but stated "Let them try to do it, and if they are wrong so be it. But let them try" (Auerbach, in Utman 1965, p. 13). A subcommittee composed of Steel and Caracciolo was formed to create a proper proposal to present to the IFIP governance.

At TC-2's sixth meeting, Steel presented the proposal, the introduction of which is worth quoting in its entirety:

> At the IFIP TC 2 Working Conference on Formal Language Description Languages held near Vienna, September 15–18, 1964, it became quite clear that there does not exist a method for describing programming languages that enjoys general acceptance. A variety of proposals were put forward at this conference and others appear in the literature. The methodological differences between these various proposals are both manifold and nontrivial, yet all may have scientific merit. There is at the moment no way of selecting an appropriate method for use in practice when precise description of a programming language is required—short of personal taste, prejudice, or random choice. In view of the many programming languages needing systematic description, if only for comparative purposes, this situation is most unsatisfactory.
> (Steel 1965)

The proposal went on to explain that criteria for the judgement of language definition methods were needed, and a proper survey of approaches. These would provide the important international angles, as a community judgement was recognised as essential to this task. The charter for the working group would be the "investigation, evaluation, and development" of methodologies. Despite Bauer's objections that WG 2.2's work might hinder that of WG 2.1 and that another FLDL-like conference might be better, a vote on passing the proposal to the IFIP General Assembly was successful.

This proposal was accepted, and in 1966 Steel began to put together a list of nominees for the new working group (Teufelhart 1966, p. 7). The intention was to get people of as many nationalities as possible, to emphasise the international angle, and the list of nominees was organised by country. Zemanek noticed there was no representation by logicians, but Steel said he had asked a number and they were not interested. A meeting of the organising committee was hosted by Zemanek at the Vienna Laboratory (Steel 1967), showing his clear interest in the working group from the start; although he himself did not attend many meetings, his group was always represented, and Kurt Walk was the initial secretary. By the eighth meeting of TC-2 the list of invitees was confirmed and invitations were being sent (Teufelhart 1967, p. 5). See Figure 7.3 for a paper kept by van Wijngaarden, one of the members of the organising committee, with a list of proposed members for the group, with some handwritten annotations.

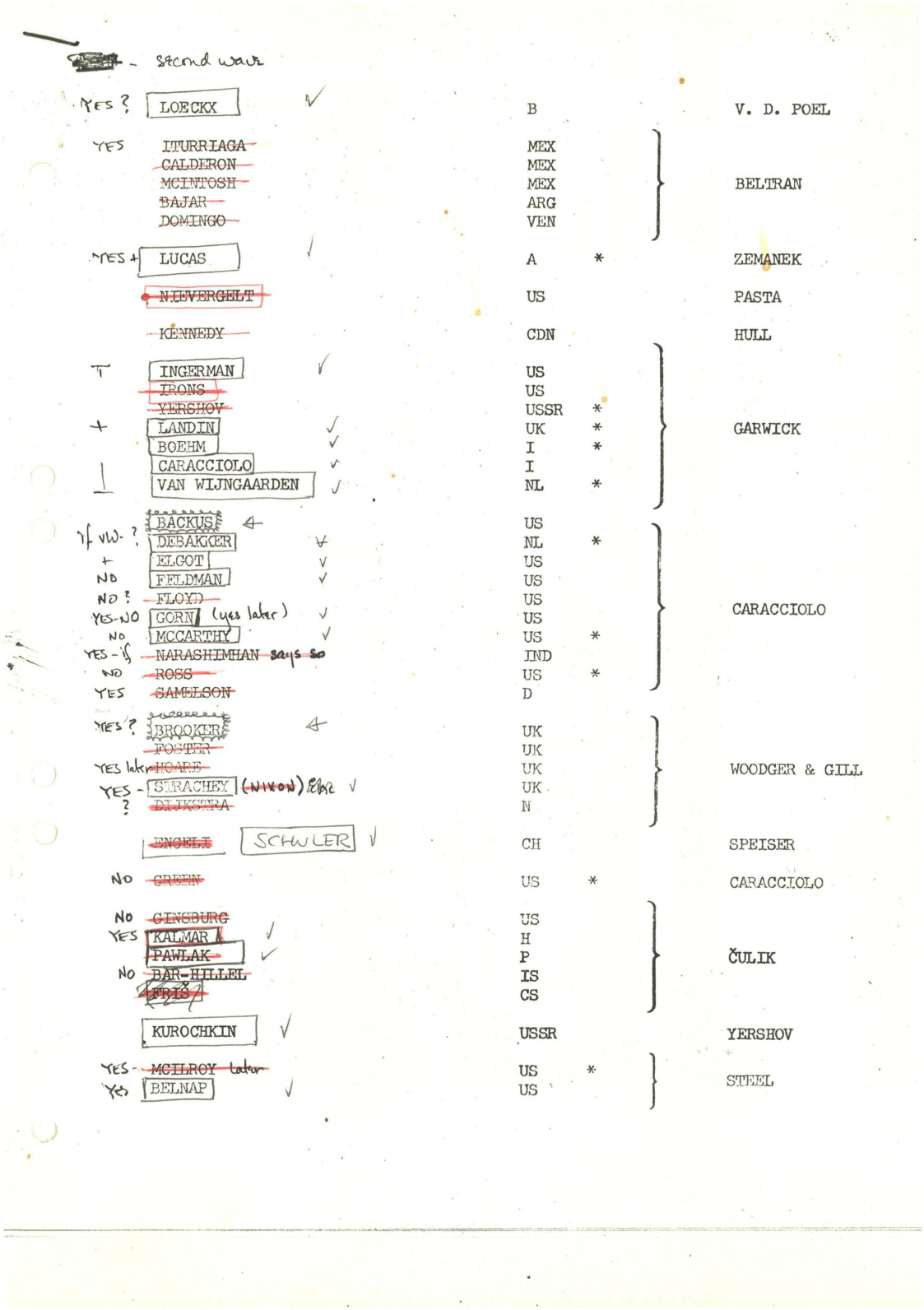


Figure 7.3: Part of a list of proposed members for IFIP WG 2.2 with handwritten annotations, likely from van Wijngaarden.

The first meeting of WG 2.2 was held in Sardinia in September 1967 under the chairship of Steel. However, right from the beginning there were administrative problems, such as constant lateness of minutes. For example, the second meeting was held in July 1968, and the minutes were circulated in February 1969, only two months before the next meeting (Walk 1968). This was probably due to the long and complicated nature of the proceedings, which were largely lengthy technical presentations.

Typically, these presentations would be from one particular practitioner on their approach and would be followed by criticism from others; van Wijngaarden, for example, hoped to get approval from WG 2.2 for his method of describing ALGOL 68, but instead received a number of arguments against his approach.

7.2.2 Crises of identity

A very common theme at the meetings was crisis of identity. WG 2.1, at that point the only other TC-2 working group, had a clear goal: the production of a successor to ALGOL 60. This was not present for WG 2.2, and so discussions of the group's purpose were frequent. McCarthy (in Walk 1969b, p. 9) made precisely this point in the fourth meeting, September 1969, and even suggested that as the group did not intend to make a joint contribution to the field it did not deserve the status of IFIP working group. The very first meeting of the group opened with a discussion of its aims, with Steel (in Walk 1967a, pp.1–2) ruling that being too specific would be a mistake. The group was unable even to decide on "what constitutes a *method* of description" (Walk 1968, p. 8), and frequently became embroiled in arguments on terminology: the minutes read "there was a lengthy discussion on the precise meaning of the terms 'concatenation', 'juxtaposition', and 'accretion' " (Walk 1968, p. 11). A particularly apposite line in the minutes of the second meeting reads "There was a discussion on the best way to proceed with the discussion" (Walk 1968, p. 23). Such debates appeared at every meeting and ran for long periods of time, with members frequently disagreeing.

Figure 7.4, from the third meeting, shows a discussion attempting to separate levels of language under study. All this discussion of the working group's aims and scopes clearly irritated Strachey (in Walk 1969a, pp. 13–4), who, following this categorisation, baldly stated "This is a waste of time".

Mahoney's (2002, p. 28) concept of agenda setting as the defining feature of a field, mentioned previously in discussion of ALGOL 68, is particularly relevant to WG 2.2.

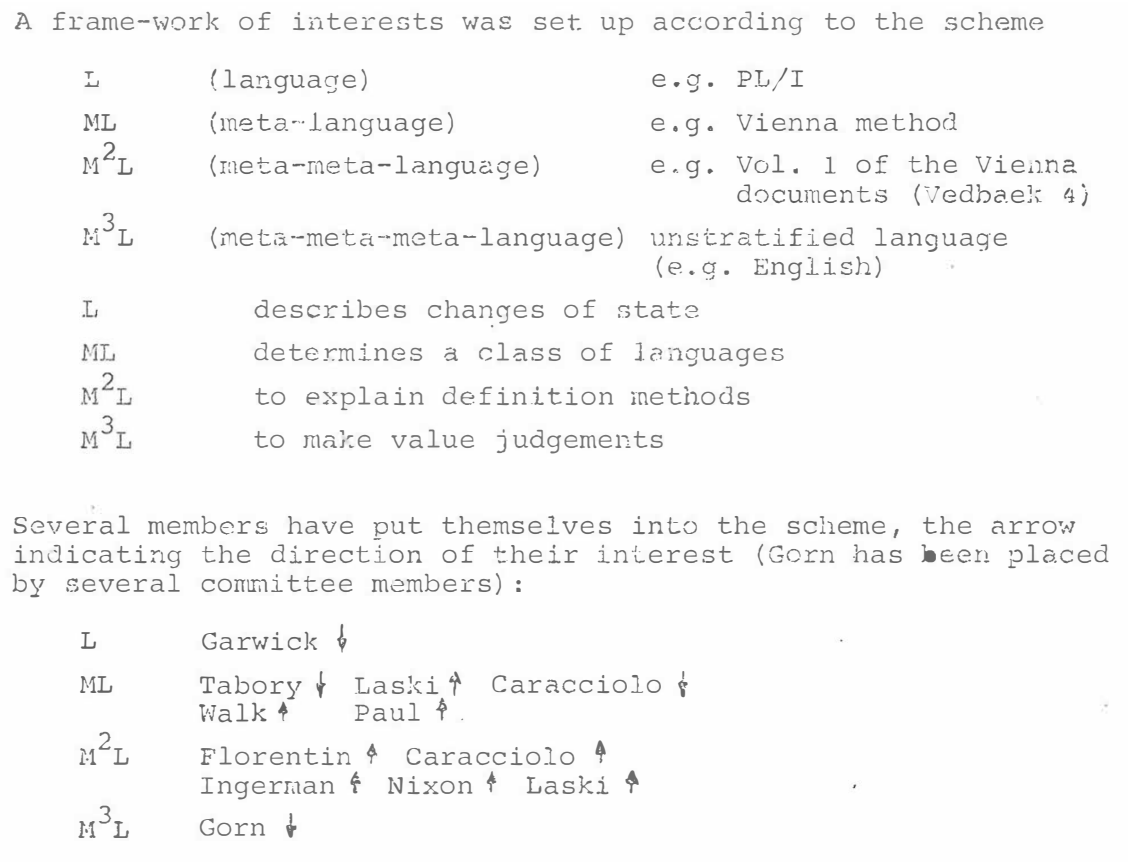


Figure 7.4: A categorisation of interest in languages in WG 2.2.

This period in the history of formal description of programming languages clearly fits this criterion, and is reflected in the constant arguments in WG 2.2 over scope and purpose. The diverse group of members can be seen, therefore, as not just setting the agenda for their group, but for the field of formal description as a whole.

Certainly, the group did not end up formulating a series of criteria for helping users to select appropriate methods for language definition, as Steel's (1965) proposal had hoped. Nor did it manage to achieve some kind of unified description technique, although Scott (in Walk 1969b, p. 18) pointed out that this might actually have been impossible: "The concept of a programming system is so abstract, it is not surprising that there is no uniform method for describing semantics".

Many projects were proposed along the lines of comparing formal language descriptions in order to better understand both the description techniques and the languages themselves. Landin (in Walk 1968, p. 28) in particular suggested a project to "compare structure of existent ALGOL 60 descriptions, paying attention to the components of each, and the interfaces between components, and distinguishing if

possible the role of each component from the means by which it is presented".[10] In other situations, Landin continued to try to provoke comparative discussions of methods, such as in a lengthy presentation on the ways referencing and name systems worked in different approaches (Walk 1969a, pp. 22–3).

This did not mean the group was a complete failure; it was rather successful at introducing practitioners to one another and provoking some collaborations. Landin's (in Walk 1970, p. 49) pessimistic pronouncement "The scope of WG 2.2 is diffuse and so is the purpose" was met by a rejoinder from Elgot (in Walk 1970, p. 49): "The group has *already* served a purpose. It brings people together, with the opportunity to criticise each other". This was particularly obvious in the case of Scott and Strachey, who had not met until they both attended the group's third meeting, in April 1969; as discussed in Section 6.4 this led to a very fruitful collaboration.

An interesting comparison can be made between WG 2.2 and the Nicholas Bourbaki group of French mathematicians in the middle of the twentieth century.[11] Bourbaki attempted to fix the foundations of mathematics through a series of jointly-written canonical books but their meetings were marred by in-fighting and the conflicting strongly-held beliefs of the participants. This comparison was not lost on the members of WG 2.2; at their second meeting, Tabory (in Walk 1968, p. 21), newly invited as an observer to the meeting and subsequently elected member, brought up Bourbaki directly, saying "Bourbaki builds a new cathedral out of well-known things and gives convenient language for these things. It is important that we are open-minded for new things which are not as yet in the cathedral". However, Bourbaki did manage to produce a series of complete and influential works, something WG 2.2 failed to achieve.

A contributory problem to the disunity of the working group was the technical depth of the ideas discussed and the length of the working papers describing them. These works clearly had a steep learning curve which prevented easy and fruitful discussion. This irked Landin, who wrote a letter to WG 2.2 about the upcoming meeting held in July 1968 in Copenhagen. A major agenda item was to be the discussion and comparison of ULD Version 2 with van Wijngaarden's draft ALGOL 68 Report, MR93. Landin (1968a) wrote:

> I foresee the possibility that much time will be devoted instead to tuto-

[10] This has since been accomplished, rather belatedly, by Astarte and Jones (2018).

[11] Thanks to Chris Hollings for raising this point to the author.

rials. This somewhat deters me going. I would appreciate quick answers to any or all of the following questions:

1. Can you live without tutorials on these papers?
2. Would it deter your attendance to learn that after ≤ 1 days [of a week-long meeting] optional tutorials on the papers the discussion would proceed on the basis of familiarity?
3. Would it deter your attendance to learn that ≥ 3 days would be devoted to tutorials?
4. Do you think that you can make significant criticisms and contributions in the framework of tutorials starting from scratch?

Landin's distaste for the extensive formalisms fits with a sarcastic remark made many years later:

> As an old friend of mine used to say: Show me a theorem and I'll reach for my gun. The grounds for such apparent bigotry are of course that if you want to slow down a creative development, then "Hold on while we prove some theorems about it" is as sure a way of pouring treacle over it as setting up a committee of experts, or an evaluation exercise.
> (Landin 1997, p. 1-3)

In tension with the distaste for lengthy, detailed technical documents was a strong preference amongst some members of WG 2.2 for fully complete description methods. Those which could not promise total coverage of all language concepts were often criticised. This duality of focus is illustrated by the following exchange:

> Scott: Only the most primitive, non-problematic things have been dealt with using this approach.
>
> Laski: A language description should specify as little as possible.
> (Walk 1969a, p. 7)

Following this, Hoare (in Walk 1969a, p. 9) made the simple yet astute comment that introduces the present work: "Difficult things are difficult to describe". However, Strachey noted that this depended on the framework of description chosen: assignment is tricky in lambda calculus, but recursion is hard in others—and the two concepts find easy and frequent application in programming languages.

The issue of difficult presentations left the group at something of an impasse: many talks were simply not understood by many members. An example followed a talk at the fifth meeting given by Landin about his idea of representing algebras as balls

rolling over graphs; Elgot (in Walk 1970, p. 45), a skilled mathematician, outright stated "I really can't understand Landin". Lucas (in Walk 1970, p. 46) noted immediately afterwards that a problem with their group's meetings was that all too frequently "there was a series of lectures in isolation and not much discussion".

7.2.3 WG 2.2 collapses

The fifth meeting of WG 2.2, in September 1970, ended rather pessimistically[12] and the next meeting in Geneva in early 1971 was aborted after only a few attended (Peck 1971, p. 2). Following this, the group found itself on something of a hiatus for two years; Steel resigned as chair and this left TC-2 uncertain of what action they should take. J. J. Duby was proposed as a new chair but initially failed to respond to correspondence (TC-2 1972, p. 8). This reflected a time of crisis in TC-2 generally, with meetings being poorly attended (the thirteenth, in 1971, had only eight of sixteen members present) and Čulík (in Peck 1971, p. 2) describing the TC as "dying".

Activity to resume WG 2.2 started in April 1973 (Steel 1973, p. 3) and a subcommittee met in June (Duby 1973). The Working Group met for the first time again in 1974 under Erich Neuhold[13] who kept the chair for twenty years (Neuhold 2016). The name of the working group was channged to 'Formal Description of Programming Concepts', which matched the renaming of TC-2 to 'Technical Committee on Programming' (Duby 1973). This reflected an expanding of the group's remit, and, consequently, its decreased relevance to the present work. The seventh meeting of WG 2.2, in January 1975, was two-thirds devoted to databases, and the following, in September of the same year, was to be entirely concerned with the same topic (Blum 1975). The ninth meeting, held in May 1976, was even broader in scope, with its agenda described as "Formal descriptions of operating systems, computer systems,

[12]The final item of discussion was a suggestion for changing the name of the group. In true IFIP Working Group form, the phrasing of the question for the vote was carefully worded:

> The majority of members present felt unsatisfied with the present title.
>
> Three titles were finally proposed, and a straw vote taken on the question: "who cannot live with that title?"
>
> The result was:
> Formal Language Description: 7 yes
> Formal Aspects of Programming Languages 6 yes
> Theory of Programming Languages: 0 yes
> i.e. everybody could live with the last title.
> (Walk 1970, p. 49)

[13]Neuhold had once been a member of the Vienna Laboratory, but by this point was working on databases, a topic to which he contributed for much of the remainder of his career.

and the world" (Blum 1975, p. 6). Although some members, such as Ingerman (in Walk 1968, p. 29), had been advocating as far back as the second meeting that broadening the group's scope was essential, achievement of this did not solve the group's problems with setting agendas, and the 'business' sessions continued to be dominated by questions of scope and focus.

This section will close with an amusing story. Neuhold, newly anointed as chair in 1975, was bringing together invitations for an upcoming meeting. IFIP had long held a strong role connecting East and West during the Cold War period, as Zemanek had always advocated; but sometimes this caused problems.

> I remember, I was Working Group 2.2 chair, and of course in formal description the Russians and Polish are very strong. [...] So I remember I was in IBM, and I wrote an invitation to the next meeting, which of course was a letter. And I used IBM envelopes because they were all around, but I went to the post office to post them, because I didn't want IBM to pay for it. Then, would you believe, the next day the security people of IBM showed up in my office and said I'm sending letters to the East! That was Communist! And how dare I do that!
>
> So first of all I was very interested in how they got the information. Because it was the post office and not IBM! And they asked me to open the letters. I told them, it's just an invitation! But they wanted me to open, and said "If you ever write that, you have to leave the letters open, and they have to come to us, and we will check them."
> (Neuhold 2016)

7.3 Denotational semantics in Vienna: VDM

Let us now return to Vienna, and an interesting example of interactions between practitioners of language semantics having a positive effect.

7.3.1 Moving on from VDL

One later outcome of the ULD effort (see Section 5.7) was a series of attempts to use their formal language definition approach as a basis for compiler design. This was largely the work of Lucas and Jones, the latter of whom had been interested due to his difficult experience at IBM Hursley achieving a reliable PL/I compiler. This had taught him that using test cases was not sufficient to demonstrate correctness

of a software product:[14]

> We saw 635 hand-written test cases run successfully and we had automatic tools to generate unlimited numbers of further test cases. The PL/I compiler was debugged around these test cases, shipped, and fell over on an embarrassing number of customer programs. I became convinced that testing could never substantially increase the dependability of a product and that quality had to begin at the earliest stages of the design process.
> (Jones 2001, p. 4)

Arriving at the Vienna Laboratory in 1968, Jones became quickly involved in the proof efforts, along with another new colleague Wolfgang Henhapl; see for example Jones and Lucas (1970) and Henhapl and Jones (1970b). Although this work showed that proofs about compiling techniques for language concepts could be based on VDL descriptions, it also indicated that it was more difficult than need be. An essential step and key lemma (number 10) in the Jones/Lucas paper had to show that the environment remains constant during the interpretation of successive statements in a given block—even though the first statement could be a nested block or a procedure call whose interpretation requires that a new environment is temporarily used. In the VDL grand state style, a stack of environments was part of the state, which made the proof of this lemma gratuitously difficult. This led to a desire to move away from the large, complex state of VDL and towards something smaller and perhaps more modular.

Jones also did not like the VDL style of jump handling, describing the control tree concept as "messy" (Astarte and Jones 2018, p. 100). Instead, he began to work with Henhapl on an alternative approach: the 'exit' concept published in a Vienna technical report just before Jones left to return to Hursley (Henhapl and Jones 1970a). Jones' (1970, p. 9) view was that jumps should not "take the interpreter by surprise", and the exit framework was a way to address this. The idea was to avoid the control and control tree state components and add a (possibly null) 'Abnormal' object paired with resulting states; a non-null value would indicate the presence of jumps and identification of the label that had to be caught as phrases were completed.

After finishing his first stay in Vienna, Jones returned to IBM Hursley in September 1970 to take over an 'Advanced Technology' group. Dave Allen, who had worked on

[14] Jones (2015, p. 75) further notes: "Sadly this was not widely accepted in IBM: One senior US manager maintained 'Give me a FORTRAN compiler and enough PL/I test cases and I'll give you a PL/I compiler' ".

the ULD-II project,[15] joined the group, as did Dave Chapman and Peter Gershon. The group's first chosen project was to write a formal description of ALGOL 60 with two ideas taken from Jones' stay in Vienna: a smaller state, and the exit mechanism. This description (Allen, Chapman, and Jones 1972) was referred to as 'functional semantics' as the main body was a series of recursive functions passing around small state components. However, it is important to note that this description was not 'functional' in the sense of denotational semantics, where functions are the denotations of language concepts; rather, here, the functions used are for interpretation.

Another innovation in the Hursley functional definition was the separation of static type checking. Unlike the previous ULD descriptions where all non-syntactic errors were caught dynamically (which is to say in the semantics), this description contained notes on how to trap errors like type mismatch or undeclared variables before the application of a semantic function. The aim was "basically to check those things which rely only on symbol matching and omit those checks which, in general, rely on values of symbols" (Allen, Chapman, and Jones 1972, p. 5). This idea matches the notion of 'context conditions' as used in the ALGOL 68 report, but there was no particular reference to that document and it seems the evolution was in parallel.

Finally, the description had some new capabilities for handling non-determinism, such as order of expression evaluation. This was achieved by including sets as a basic object and introducing a non-deterministic operator to arbitrarily pick set elements. However, this operator was not fully defined and so the non-determinism of the description as a whole depended on its ability to be constructed.

7.3.2 Future Systems

Meanwhile, the Vienna Laboratory had been working on automatic ways to detect potential parallelism in FORTRAN programs, but these efforts were relatively unsuccessful. A new direction came with an ambitious plan from IBM to design a machine architecture that was radically different from that of System 360 that had dominated in the 1960s. The aim of this 'Future System' (FS) project was to make computers far easier to use, and it included concepts such as a one-level address space, unforgeable pointers, and built-in support for procedure calls. Because the whole FS project was eventually cancelled and remained under wraps in IBM, little was published about it. Radin's (1976) report on potential machine architecture

[15] See Section 5.4.

and the memo by Sowa (1974) give some hints of the novelty of ideas which were explored. As part of the FS effort, the Vienna Laboratory was asked to build a PL/I compiler for the new system—and, as no constraints were placed on the methods they were to use to achieve this, they decided to start with a formal description of the ECMA/ANSI standard PL/I (ANSI 1976).

Bekič, Lucas, and Jones had been exchanging letters throughout 1972 discussing how to fit their ideas into a denotational context, the better to address the difficulties with proving properties (Jones 2015, p. 76). These experiments on a 'design language' were influenced by the fact that Bekič had spent the academic year 1968–69 with Landin at Queen Mary College London, and Jones had attended some of Strachey's Programming Research Group lectures in 1971–72. Bekič's work had been on concepts in programming languages, as evidenced by his presentation on modelling arrays in ALGOL 68 at WG 2.2 (Walk 1969a, pp. 27–9); he had also worked on fixed point constructions (Bekič 1984a). Further evidence of influence of denotational ideas on IBM Hursley can be inferred from a letter from the Oxford University Registry to Strachey in 1970, in which Peacock (1970) explains that he went to Hursley to discuss ways in which their could be more contact between the PRG and the IBM Laboratory; he wrote "It was evident that your work has more interest for them than anything else that I could discover". Another angle of influence had come from Scott's visit to Vienna in late Summer 1969. He had been invited to explain Floyd's (1967) method to the Lab, but instead talked about his work with de Bakker over the summer. This laid the seeds for understanding the denotational approach when it came (Jones 2001, p. 5).

In late 1972, Lucas called Jones to ask him to return to the Vienna Laboratory, on a permanent transfer this time. Jones (2001, p. 7) recalls that he agreed even before Lucas asked the question: to be able to create a compiler from a formal definition was the dream. Jones moved in early 1973 and was joined in May by Dines Bjørner. Bjørner was a Danish researcher who had already been working for a couple of years for IBM San Jose: his previous work was on systems architecture for various machines, particularly utilising graphical approaches to design (Bjørner 2007). At San Jose, he had worked with Backus and Codd, both important and influential researchers (Bjørner 2015, p. 72).

7.3.3 A new Viennese semantics

The Vienna Group began working on their description of PL/I in 1973, and the denotational approach they took became known as 'VDM'. This name, standing for 'Vienna Development Method'[16] was actually coined rather late in the project (Walk 2002, p. 84). The name 'META-IV' was given to the metalanguage used, the IV being a successor to the ULD-III descriptions (Bjørner 2015, pp. 72–3). There is now a certain ambiguity in the name VDM: to many people, VDM refers to a development method for all forms of computer system;[17] here, 'VDM' is taken to refer specifically to the technique for language description that evolved in and from work in the Vienna Lab between 1973 and 1976.

A key advantage to switching to a denotational framework was much greater clarity about which objects could be changed by each semantic function, as the Curried parameters were much more explicit rather than bundled in a state. The absence of an environment on the right hand side of the procedure denotation, for example, showed that it could not be changed by a call and return, neatly solving the problem of Lemma 10 from the earlier Jones and Lucas (1970) paper. Jones (1999, p. 35) argued that this meant it "kept the group honest" and prevented the use of tricks like keeping the program text in the state. Walk (2002, p. 82) agreed that switching to denotational semantics also made the construction of mathematical proofs considerably easier.

Overall control of the PL/I for FS project was by Kurt Walk; by this time, Zemanek had been made an IBM Fellow and Walk was Lab director. Initially there were two sub-groups with Viktor Kudielka managing the front-end and Peter Lucas the back-end. When Lucas transferred to IBM Research in Yorktown Heights, Kudielka became manager of the project and Jones became 'Chief Programmer' around April 1974. The project occupied most of the 20 or so professional members of the Lab. It is worth adding that the task of designing a compiler for a machine

[16] Jones notes archly that a number of different expansions of the acronym were suggested:

> At one point, the acronym 'VDM' was claimed to have stood for 'Vienna, Denmark and Manchester'; at another 'Very Diverse Methods'; my preference was for '*Vorsprung durch Mathematik*' [progress through mathematics; a play on Audi's slogan]. [...] There was some concern that 'VDM' was too close to 'VDL' and this did indeed confuse the unwary.
> (Jones 1999, p. 43)

[17] This aspect is placed in a historical context in a paper in the *Annals of the History of Computing* by Jones (2003b) on the history of program verification.

whose architecture was both novel and evolving presented considerable challenges.[18]

The process of documenting a full VDM denotational description of the ECMA/ANSI subset of PL/I was based on an early construction of the main semantic domains developed in a long meeting in the coffee room of the third floor of the Vienna Lab (Jones 2001, p. 8). These domains remained basically constant and provided a way for the authors to work somewhat independently. At the end of 1974, a Technical Report (Bekič et al. 1974) had been printed. The authors listed for TR25.139 are Hans Bekič, Dines Bjørner, Wolfgang Henhapl, Cliff Jones, and Peter Lucas, although ten further colleagues are acknowledged for contributions including detailed reviews. A number of other technical reports were published on aspects of the compiler development process using VDM (e.g. Weissenböck (1975), Izbicki (1975), Bekič et al. (1975), and Jones (1976)).

META-IV as a notation brought a richness to the basic objects available: sets, sequences, maps (a set of ordered pairs), and records (tuples with named fields) could all be used. This development stemmed from the insight that including sets as basic objects in the Hursley functional description (Allen, Chapman, and Jones 1972) had brought many advantages with relatively little to pay. The metalanguage was used for abstract syntax as well as the semantic functions and its written style was very different from the Oxford approach: textual identifiers with intuitive names were generally preferred over single Greek letters which makes the descriptions much easier to read. Jones (1999, p. 33) speculated that this decision may be due to the Vienna group's background as writers of full-size programs and their acknowledgement of the importance of intuitive names. The Oxford Programming Research Group group was, however, also populated with programmers, but their desire to link clearly with mathematics coupled with the fact that they did not have to interface with industrial non-technical managers may have been the reason for their terser style. Some of the notation in META-IV shows clear influence from

[18] An interesting cautionary tale about formal descriptions relates to that of the FS architecture itself and was described by Astarte and Jones (2018, p. 116). As indicated, the architecture was novel and quite complicated. Clearly, to design a compiler, it was necessary to have a strong description of the evolving architecture. A small team in the IBM Lab in Poughkeepsie wrote a formal description that initially used rather abstract types and implicit definitions. This was not, of course, executable. Management suggested that since this had involved a lot of work (and thus expense) it would be better if it could execute FS instructions. The team laboured to achieve this and then to respond to a subsequent request that it should be optimised to run at a more acceptable speed. At the end of this process, the long and detailed description was of little use to the Vienna Lab as a basis for reasoning about the run-time part of their compiler for the FS machine and Hans Bekič had to write a short formal description to guide the compiler code generation work.

McCarthy, such as the *mk-* function for constructing records (see McCarthy (1963) and the discussion in Section 3.1). The typing of all semantic functions was also included in VDM descriptions, which significantly aids comprehension.

A development of the static error checking first visible informally in the Hursley functional description became standard in VDM. These static checks generally took the form of functions (although some were written informally in natural language) declared alongside the semantic functions, as 'pre conditions' for each: they declared the criteria that must be satisfied for the semantic function to be applicable (Lucas 1987, p. 7). The idea was developed further in a formal description of ALGOL 60 penned by Henhapl and Jones (1978). Here, they were called 'context conditions' for the same reasons as those in the ALGOL 68 definition:[19] because they relied on contextual information that could not be supplied by a context-free grammar (Lucas 1987, p. 7). The advanced context conditions in the ALGOL 60 description were fully formalised, and used a special static environment containing information like typing. Each semantic function was accompanied by a well-formedness Boolean function; these were a family of predicates *is-wf-*θ for each syntactic class θ that determined well-formedness with respect to the declared types of variables.

Much the most significant difference between the Vienna and Oxford Programming Research Group denotational descriptions (aside from notation) is in the handling of jumps: exits instead of Oxford's continuations. Proof of equivalence of these two concepts was written by Jones (1978), so it was clear both were an acceptable way to model abnormal termination.[20] The Vienna group chose to pick up the exit idea used in the Hursley 'functional' description as a simpler mechanism for describing exceptional termination of phrase structures.

For languages without abnormal sequencing, functions from states to states ($\Sigma \to \Sigma$) could be used for the space of denotations. The denotation of the sequential composition of statements in the object language could be mapped into the composition of the denotations of the separate constructs and fixed points could be used to define (homomorphically) the denotation of repetition in terms of the denotation of the body of the loop. This was the same in both VDM and Oxford semantics.

The exit formulation in VDM used functions from states to pairs of states and an

[19] Again there was no reference made to van Wijngaarden's work, although Jones, Lucas, and Walk would have been well aware of the two-level grammars from WG 2.2.

[20] Section 6.5 explains how continuations are used to model exceptional sequencing such as is required by go to statements.

optional abnormal component ($\Sigma \to \Sigma \times [Abn]$) as the basic denotations. The denotation of say $s1; s2$ was then derived in a slightly more complicated way from their separate denotations: when the abnormal part of the pair for the denotation of $s1$ is **nil** the denotation of composition passes the state part of the denotation of $s1$ to the denotation of $s2$. If however the abnormal part of the denotation of $s1$ is non-**nil**, the pair from only $s1$ becomes the denotation of the whole composition—thus effectively ignoring $s2$.

This form of composition is made readable in semantic descriptions by defining a 'combinator' whose representation was chosen to be a semicolon.[21] This combinator included the simple functional composition with the caveat about *Abn* mentioned above. One observation worth making about combinators is that it is possible to read them operationally: although the semicolon above is defined as a combination of functions, it can be interpreted as an operational definition that first performs the computation before the semicolon followed by that after it. Jones believes that Lucas, who was somewhat resistant to the change to a denotational framework, tended to read the definitions operationally.[22] Jones (1999, p. 37) reported that Lucas also argued that using a combinator made language extension easier as the combinator could be redefined once and then used throughout the document.

The denotation of a goto statement in VDM made no change to the state but returned an abnormal value; this was defined using another combinator, called *exit*. The propagation of abnormal values had to be caught somewhere and this required one more combinator for which the name was chosen by writing "exit" backwards (*tixe*).[23] Although the approach of exits was less powerful than continuations (being unable to model co-routines, for example) Jones (1999, p. 37) argued this was actually an advantage: "where the weaker idea was adequate, employing it might say more".

[21] Peter Mosses (2011) pointed out that the use of combinators in VDM is similar to the later development of Moggi's (1989) 'monads'. The use of combinators was another factor that made denotational descriptions in VDM look different from those written in Oxford where arguments are passed to Curried functions.

[22] Jones, personal communication, 2018.

[23] Further details of how the exit concept was used in modelling programming languages can be found in the paper regarding definitions of ALGOL 60 (Astarte and Jones 2018, particularly § 4.4.5.4 and 4.6.4).

7.3.4 The death of FS and saving of VDM

On a day in 1975 which became known as the "St Valentine's Day massacre" (Jones 1999, p. 43) the FS machine project was cancelled and it gradually became clear that the next mission of the Vienna Lab would be the development of conventional IBM products. After the shock of seeing several years' work apparently discarded, many of the key researchers began to leave the Lab: Bjørner back to Denmark and a chair at the Technical University of Denmark (in Lyngby), Henhapl to a chair in Darmstadt, Germany, and finally Jones moved in 1976 to IBM's European System Research Centre in La Hulpe, Belgium.

After the cancellation of FS and thus the PL/I compiler project, Bjørner and Jones agreed to try to preserve the VDM denotational style by cajoling their former colleagues to contribute to a book (Bjørner and Jones 1978). Originally, plans were to publish a series of papers in IBM's *Journal of Research and Development*, but these aims did not pay off, perhaps due to the lengthy nature of some of the papers, and instead Springer published the book in their *Lecture Notes in Computer Science* series. The book also included the description of ALGOL 60 written by Henhapl and Jones (1978) in the same style as the denotational description of PL/I, and a reflective piece by Lucas (1978). There is a coda to this story. At that time, Springer Verlag appeared to take the attitude that once an LNCS volume had sold its initial print run that their task was complete. When they declined to reprint LNCS 61, Tony Hoare came to the rescue and offered to have a suitably updated collection of papers reprinted in his prestigious 'red and white' Prentice-Hall series. This book (Bjørner and Jones 1982) contains among other contributions a revised description of ALGOL by the same authors (Henhapl and Jones 1982). The revision differs mainly in the order of presentation.

As well as the Bjørner and Jones (1978) book, a useful and interesting source is the collected papers of Hans Bekič, who sadly died in October 1982, during a mountain climbing accident (Zemanek 1986b). Following his death, Jones gathered together a number of his works into an edited volume (Bekič 1984b). This is particularly useful given Bekič's habit of failing to finish or publish many of his writings. A photograph of Bekič not long before his death can be seen in Figure 7.5.

Another substantial development from the VDM work was its application to program specification and verification. Walk (2002, pp. 83–4) explains:

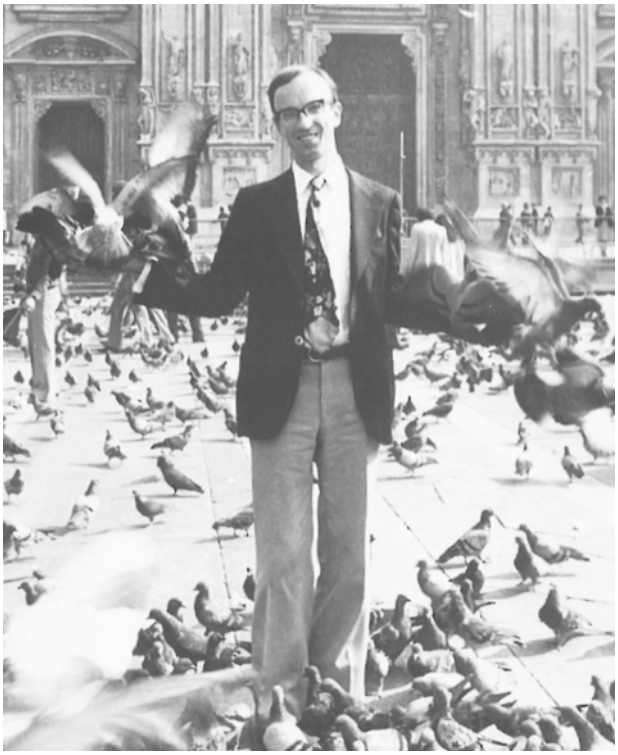


Figure 7.5: A photograph of Hans Bekič in the early 1980s.

> The thinking was: if a program is supposed to realize a mathematical function, and if the meaning of the program is formulated in mathematical terms, it must be possible to mathematically prove the correctness of the program by proving the equivalence of the two. Initial steps were taken to show the practical feasibility for non-trivial, yet small programs. The application of these methods to larger programs soon hit limits of size and complexity.

Instead, correct-by-construction ideas were developed. The VDM design process comprised a series of specifications proceeding from most abstract to a concrete implementation, linked at each level by proofs of correctness and equivalence (Lucas 1987, p. 1). This was the application of VDM which gained the most general acceptance, and its use for language definition was somewhat left behind. One key aspect of the VDM process for program description was the use of 'retrieve functions' to show homomorphism across data abstraction and reification (also called 'data refinement'; essentially the reverse of abstraction). This was used first for showing the relationship between concrete and abstract syntax, but could also be used for data (Jones 1999, p. 38). The work just cited provides a good exploration of the application of VDM to program specification, development, and verification, so that is not repeated here.

Although the main application of VDM now is for programs rather than languages, there were some other descriptions completed: Pascal (Andrews and Henhapl 1982), Smalltalk (Wolczko 1988), Modula-II (almost uniquely used as the standard) (Andrews et al. 1988), and even database languages (Welsh 1982, 1984). Furthermore, the VDM approach was used by Bjørner and Oest in a bid for the USA's Department of Defense language Ada in the late 1970s. This work was supported by a number of Master's students taught by Bjørner at the Technical University of Denmark, and followed a definition of the CHILL language for telephony (Study Group XI 1980). Although the VDM proposal was not the one taken forward in the final creation of Ada, a (deeply technical) book was published with information on the language description techniques employed (Bjørner and Oest 1980). This included some steps towards addressing parallelism by Løvengreen (1980) which used an operational approach to combining the sequential denotational definitions of Ada tasks. The Ada work, which resulted in a compiler derived from the VDM definition in a much smaller time-scale than would have been possible by traditional means, was referred to by Bjørner and Havelund (2014, p. 8) as "an unqualified formal methods success story".

Reaction to the VDM approach to language definition was rather muted. One who spoke loudly in criticism was Dijkstra, who visited Newcastle University in September 1974 for their Formal Aspects of Computer Science seminar.[24] Dijkstra heard Scott speak about his domain theory and Bekič on Vienna's language definition work using Scott's formalism—and did not care at all for either. He wrote a very acerbic trip report (Dijkstra 1974a) and posted it to a number of people, including Scott and Bekič, disparaging not only their subject material but also their lecturing style. Jones remembers[25] that Bekič came to him for advice, worried because he had been publicly attacked by a leading figure in computing. Jones cautioned quietness, counselling that nothing good would come from responding to Dijkstra's ill-tempered behaviour. Scott, however, did reply, and somewhat vehemently, which led to Bekič making a reponse as well; even Zemanek weighed in. Dijkstra eventually addressed all these people and somewhat backed down, explaining that his problems were with semantics more generally.

Dijkstra's concerns notwithstanding, VDM represents a unique shift in the story

[24] Dijkstra (1974a, p. 1) added "My sweater was identified as one of the informal aspects of computing science". (A reader following this citation should note that EWD documents are typically numbered from zero).

[25] Personal communication, 2018.

of semantics: a significant shift in the core approach of a group. The early VDL operational work provided certain aspects like abstract objects and syntax; Jones' input was the exit concept and context conditions; and denotational ideas came from the Oxford Programming Research Group. This all mixed together to form compact yet readable and mathematically sound descriptions.

7.4 Structural operational semantics

Another interesting shift of direction is in the story of Structural Operational Semantics, the work of Gordon Plotkin, who went from contributing dense mathematical concepts in denotational semantics to a much simpler and more intuitive operational approach. To best outline that story, it is worth first sketching some of the history of formal semantics work at Edinburgh University.

7.4.1 Computing at Edinburgh University

Edinburgh University has long been a centre for research on formal and theoretical computing, especially the Experimental Programming Group. This was headed by Donald Michie and was focused on AI research. Some of this work spilled over into formal semantics, but more was in program verification; the emphasis here is on the former. The group started in 1964 and included at that time Robin Popplestone and Bernard Meltzer (Burstall 2017).

Rod Burstall had moved to Edinburgh to join the group following a few months working for Strachey in London at the end of 1964[26]. Strachey had introduced Burstall to the ideas of semantics and programming languages during this time, and was impressed with the younger scholar, as the following reference indicates:[27]

> In the first place let me make it clear that I have the very highest opinion of Dr. Burstall's ability. I have known him and discussed the problems of programming languages for nearly five years. Even after discounting a bias in his favour which comes from the fact that his general approach is very similar to my own, I find him an impressive person.
> (Strachey 1968a)

[26] See the story in Section 6.2.

[27] In the same letter, Strachey explains that he had unsuccessfully attempted to persuade Burstall to move to Oxford to work for him.

To Burstall, the most important use of semantics was ensuring compiler consistency; he remembered that in the early days of his work in computing, two FORTRAN compilers could give different answers to the same program (Burstall 2017).

After his move to Edinburgh, Burstall became interested in using first-order logic to describe programming languages. This started with his piece 'Semantics of Assignment' (Burstall 1966), which was partly a response to "Strachey's challenge to give semantics to realistic programming languages" (Burstall 2000, p. 53). In this paper, Burstall highlighted particularly the role of semantics in enabling the prediction of programs' results, and proving properties of programs, especially equivalence. He wrote that semantics "tries to connect the very empirical subject of programming with the main body of mathematics, particularly with mathematical logic" (Burstall 1966, p. 3). Here we see again the recurring notion, especially important to researchers in the 1960s, of establishing the integrity of computing through ties to mathematics and logic.

Burstall's work shows obvious influence from Strachey and especially Landin, the latter of whom was close friends with Burstall. Landin's ISWIM notation is used as the object language in Burstall (1966), but Burstall only used the aspects of it corresponding directly to lambda calculus. He did not much care for the way Landin had treated assignment and wanted to consider a different approach (Burstall 2017). Models of storage and data structures *were* treated, however, using Strachey's ideas: an interesting synthesis of approaches.

Another piece of work from Burstall was the POP-2 programming language, written along with Edinburgh colleague Popplestone[28] (Burstall and Popplestone 1968). This was a functional programming language deeply influenced by both ISWIM and CPL (Burstall 2017) and was used to write the controlling software for Edinburgh's Freddy robot in the early 1970s (Ambler et al. 1973).

Burstall provided a more foundational contribution at the Mathematical Theory of Computation conference held at IBM in 1968. This was a paper on 'structural induction', a method for proving properties of programs, and was later printed in the *Computer Journal* (Burstall 1969a). The idea was to provide a method for proofs via induction over the structure of the data used in a program. It was adopted from McCarthy's ideas on compiler correctness; Jones (in Burstall 2017) argued that it was an improvement over McCarthy's 'recursion induction' because of its

[28]Burstall notes that Popplestone originally wanted to call the language 'Cowsel' but Burstall objected because it sounded "too rural" and did not suit a programming language (Burstall 2017).

much simpler side-conditions.

The final solo work on semantics that Burstall wrote applied a modification of his earlier approach to ALGOL 60. This, in his own words, "shows how the semantics of a programming language can be defined by a set of sentences in first-order logic" (Burstall 1969b, p. 79). Rather than being based on lambda calculus as in his previous work, predicate calculus was used as the foundation. This work shows similarities with other model-based descriptions, particularly that of McCarthy, given the use of a store state and program point as its core semantic component. The concept of program points stemmed from McCarthy and Landin's work, and was used to control the ordering (including that of jumps) in a rather intuitive way: being state components they could be manipulated directly to affect sequencing. Burstall indicated in the conclusions that he had worked on implementing this description on a resolution-based theorem prover at Edinburgh[29] with moderate success in proving properties of programs. It was slow and finicky, but showed some promise as a future research direction.

Burstall (1969, p. 96) wrote:

> One advantage of this approach to language definition, if it can be mechanised, is that it would allow a language designer to 'debug' the formal definition of his language by trying it out on a number of test programs in that language and checking mechanically that the expected outcome can be inferred. In fact we can, in theory, push the nastiness of 'suck-it-and-see' debugging back from debugging individual programs to debugging the language definition.

In a later interview, he related the idea to the fundamental concept of programming languages:

> I realise now what I was doing was taking the programming language, which was pretty much all of ALGOL, and thinking it back into assembly language, more or less. And then writing down how it would be in relations.
> (Burstall 2017)

Burstall's 1969 paper took an important new angle in semantics: it primarily used relations rather than functions to establish correspondence between program parts

[29] For more information on the history of theorem provers, including those at Edinburgh, see *Mechanizing Proof* by MacKenzie (2001), especially Chapter 8.

and their abstract representations. The work "represented the program by a relation between statements (terms) rather than as a single term [...] because I was still accounting for goto's and labels which later became 'deprecated' ('considered harmful')" (Burstall 2000, p. 53). This would also have allowed the handling of non-determinacy: where functions require unique outputs, a relation can define multiple valid outcomes; however the lack of non-determinacy in ALGOL 60 meant this was not utilised. The description is a curious mixture of axioms, functions, and relations: almost a combination of axiomatic, denotational, and the later SOS methods of semantics.

While Burstall's approach to semantics had an appealing blend of logical founding and intuitive readability, both his papers were published in Edinburgh's *Machine Intelligence* series. This was not terribly well-read outside of Edinburgh, and in some ways was rather looked down on, according to Burstall (2017), because AI was not regarded as 'proper' computer science. McCarthy had read and liked the work, but he was more interested in his own research directions—and very few other people saw these publications (Burstall 1993). A photograph of Burstall likely taken in the 1970s can be seen in Figure 7.6. Burstall later worked on category theoretic models of semantics with Joe Goguen, including producing the semantics of Clear, a specification language the two had devised (Burstall and Goguen 1980). However, this is out of scope for the current work, and focus will now shift to one of Burstall's PhD students (in fact his first), Gordon Plotkin.

7.4.2 Gordon Plotkin

Gordon Plotkin's academic career started with a degree in maths and physics from Glasgow University. He had chosen that degree through an interest in cybernetics; upon asking the admissions tutor what to study in order to work on cybernetics, maths and physics was suggested (Plotkin 1994). Plotkin (2018) explained later "I didn't want to study computing as such, although I enjoyed what I learnt; I wanted to do artificial intelligence because I wanted to understand how the brain works". He had used computing machines for his physics work, writing programs in FORTRAN and cursing the awkwardness of the formatting (Plotkin 1994). Although he had been instructed in maths, the course had no components in logic and so his knowledge in this area came only from his personal interest.

Plotkin moved to Edinburgh in 1968 to start a PhD. His initial supervisor was Pop-

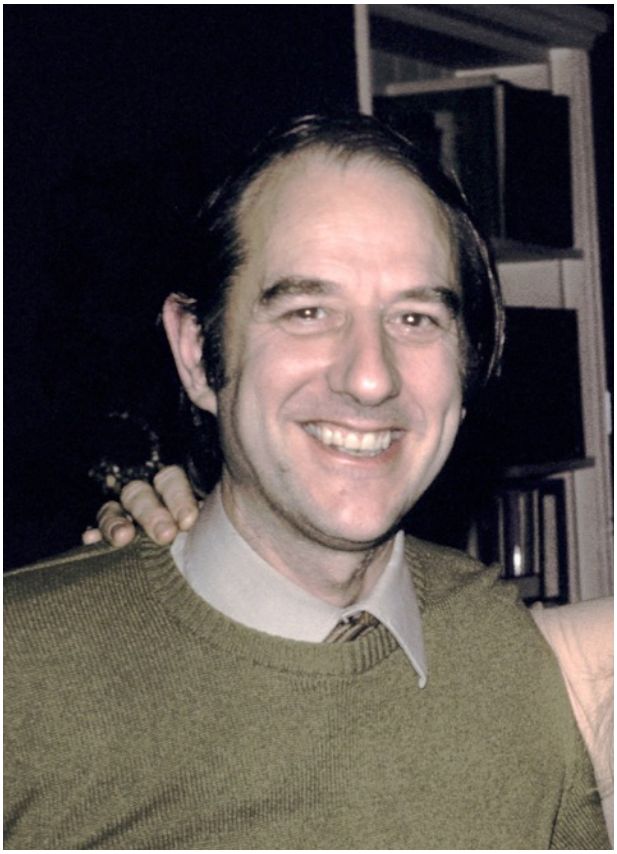


Figure 7.6: A photograph of Rod Burstall in the early 1970s, taken from a talk by MacQueen (2015).

plestone, and then Burstall took over as Plotkin's ideas shifted; he recalled "once I was at Edinburgh, there were a million interesting things going on, including semantics" (Plotkin 2018). The subject of his research (Plotkin 1971) was a system for generalisation of information from experience—the inverse of unification, which was a key contribution to theorem proving by Alan Robinson. Plotkin (1994) remembers that during the early 1970s, Edinburgh had a lot of small groups arranged around various different research topics; these groups constantly mixed, coalesced, and reformed. This was tolerated by the University due to its wealthy status at the time. Part of this experience was how Plotkin learned about the concepts of theorem proving and resolution; from Burstall specifically he learnt about untyped lambda calculus (Plotkin 2004, p. 4).

Plotkin's first work relevant to the current account was in denotational semantics, although later he also contributed significantly to operational semantics. He explained the difference in an interview (Plotkin 2018): "[Operational semantics is] more behavioural. Then you're trying to say what happens in the execution of pro-

gramming languages in some abstract way. Whereas denotationally you're trying really to say what the result is" and "If you want to understand a programming *language* and how it works, then operational semantics is very good, but if you want to understand *programs* and what they do, denotational semantics gives you a clearer view of them overall without going inside them". Plotkin had experienced a seminar from a member of the Vienna Laboratory, most likely Lucas, explaining the basics of their VDL approach, but he did not care for it, writing:

> I recall not much liking this way of doing operational semantics. It seemed far too complex, burying essential semantical ideas in masses of detail; further, the machine states were too big. The lesson I took from this was that abstract interpreting machines do not scale up well when used as a human-oriented method of specification for real languages. (Plotkin 2004, p. 4)

Although Plotkin never met Landin, he was very influenced by Landin's work. Further influences came from a good deal of contact with the Programming Research Group through Wadsworth and Milne, although he he did not like the Adams Essay's approach: "I was very influenced by the Oxford Programming Group memoranda, but I did not at all like the over-complex style of the Adams prize essay, though there were certainly important things there, including logical relations" (Plotkin 2018). Plotkin spent 1972 visiting Alan Robinson in Syracuse, but ended up talking more with John Reynolds, largely about his (1972) work on interpreters (Plotkin 2018).

Plotkin's major contribution to denotational semantics was the 'power domain', a way to handle concurrency in a denotational framework (Plotkin 1975, 1976). A problem with non-deterministic semantics is that it breaks the concept of a functional denotation: functions must have unique output, but parallel programs can have many different and equally correct meanings. Power domains were created as an analogue to powersets and their construction and use is very mathematically challenging: Reynolds, reviewing Plotkin's initial paper, took three months to understand the mathematics involved.[30] Any reader familiar with Reynolds' own contributions will understand that he was not lacking in ability, so for him to take so long reviewing is notable.

Interestingly, in an interview with the present author, Burstall (2017) claimed that the central idea for power domains may have come from him:

[30] As reported by Reynolds to Jones, who mentioned it to the present author in 2018.

> I talked to Robin Milner in a sauna, and he was talking about doing relational proofs, and I said "Why don't you use functions to sets?" and that was powersets, and he told Gordon. [...] They didn't have one of these microphones[31] in the sauna, so I can't prove that I told Robin that!

The influence of Robin Milner on Plotkin, however, is undeniable; in his piece on the history of SOS, Plotkin (2004) mentions him repeatedly. Milner's work is largely out of scope for the present account but is worth briefly reviewing. He had worked at Northampton City College during the 1960s and at that time tried to implement CPL; from this experience he learnt the difficulties of grappling with large scale programming languages (Milner 1993). At Swansea University at the end of the 1960s, Milner had worked on theorem proving under David Cooper, and he attended Scott's lectures in Oxford in 1969 (Berger and Milner 2003, p. 6). In 1971, Milner moved to Stanford and worked in the AI Laboratory with McCarthy, where he began creating a system called Logic of Computable Functions (LCF) which applied Scott's domain theory to program verification. This system could be used to write the semantics of programming languages and verify a simple compiler; this used an algebraic approach towards encoding the meaning of program constructs. When Milner moved to Edinburgh University in 1973 he kept up the work on LCF, before embarking on a lengthy project to formalise concurrent behaviour, which found fruit towards the end of the 1970s with CCS (Milner 1980) and later π-calculus (Milner 1992).

A related tangent to the LCF work was the metalanguage for the system, which became a programming language in its own right named ML (standing for metalanguage). The language was functional in paradigm and had its roots in Landin's ISWIM with the algebraic datatypes of Burstall and Landin (1969) added to enrich the expressiveness (MacQueen 2015). The inclusion of these types was important because Scott's domain logic was untyped and that caused some problems. Adding in types created robustness because a type 'theorem' could only be used if it had been proven; this meant well-typed programs were correct by definition (Berger and Milner 2003, p. 12). However, as more and more complex aspects were added to ML, Milner realised the importance of defining a simple, core language from which other extensions could be developed. This was called Standard ML (Harper, Milner, and Tofte 1988) and was designed with a formal semantics before any implementations were created. This took a long period of time—from 1982 to 1990 (Berger and

[31] Referring to the one being used to record the interview.

Milner 2003, p. 11)—but represents an almost unique case in which a language was designed semantically before any compiler was written.

7.4.3 Relations and rules for semantics

During the 1970s, researchers at Edinburgh began to explore the idea of using rules in the style of natural deduction to express the semantics of transition systems, including programming language interpreters. Plotkin (2004, p. 4) wrote that the rules notion came largely from Smullyan et al. (1961) and Barendregt (1971). Mike Gordon, another of Burstall's PhD students, wrote his thesis on the semantics of LISP using a rules-based operational semantics as well as a pure denotational model, and then proved the two equivalent (Gordon 1973). He situated his work as similar to that of Robert Milne on implementation levels in a denotational framework,[32] although the two works were achieved somewhat concurrently. Crucially, Gordon's semantics included the idea of 'configurations', a binding together of state or environment with program text.

Plotkin's first working out of the rules concept was for a language called PCF, a simple programming language based on Milner's LCF. The publication on this, like Gordon's thesis, used both a relations-based semantics and a denotational model (Plotkin 1977). Plotkin and others at Edinburgh continued to use these notions, particularly in the development of non-deterministic, concurrent, and parallel systems.[33] One example of this was Milner's (1980) Calculus of Communicating Systems (CCS) which included a rules-based operational semantics, and the concept of labelled transitions. This was written following a period of six months developing the idea while lecturing at Aarhus University.

Although these works all used operational concepts, the prevailing belief was still that denotational semantics provided the most abstract and therefore standard semantics. However, Plotkin's views began to change at the end of the 1970s:

> A realisation struck me around then. I, and others, were writing papers on denotational semantics, proving adequacy relative to an operational semantics. But the rule-based operational semantics was both simple and given by elementary mathematical means. So why not consider dropping denotational semantics and, once again, take operational

[32] See Section 6.5 for more on Milne's work.

[33] For precise references to all these works, see Plotkin (2004).

> semantics seriously as a specification method for the semantics of programming languages?
> (Plotkin 2004, p. 6)

Plotkin took a visiting lectureship for six months in 1981 at Aarhus University, as Milner had done, and formulated the idea from the above quotation in a series of lectures; the 'Aarhus Notes' accompanying these laid out the essence of Plotkin's Structural Operational Semantics (SOS). They were (as with so many documents in the present account) not officially published, as they did not cover all the material Plotkin had presented (Plotkin 2018). He wrote (2004, p. 9) "I sometimes wonder if (and hope that) the fun of obtaining 'underground' copies helped push the ideas more than conventional publication would have done!"

The core idea of SOS is to give meaning to language constructs with rules that pattern match against syntax, either abstract or concrete. A list of premises for each rule allows the handling of multiple cases naturally and the use of relations to indicate the transitions is crucial. Instead of a function producing a sole correct answer, a relation constrains whether a given output is satisfactory. The state in SOS descriptions is typically kept small and is bound together with an appropriate fragment of program text in the configuration style of Gordon. More technical detail on precisely how the SOS approach works is given by Jones and Astarte (2018) and so will not be covered further in this section.

A particularly useful aspect of the SOS approach is how well it handles parallelism: the choice of which thread of execution to follow is made when choosing which rule to apply. This, coupled with the use of relations, sidesteps a lot of the difficulties of managing concurrency. It might seem that this simplicity was a reaction to the complexity of power domains required for handling concurrency denotationally, but Plotkin (2018) does not believe this to be the case, saying in an interview:

> I really don't think there was anything ... except for the generality that when one did work in denotational semantics, one proved adequacy theorems, and I realised that adequacy theorems were kind of nice by themselves. You could just do the operational semantics; instead of it being an adjunct to adequacy theorems. That was one sense in which operational semantics grew out of denotational semantics. But not specifically power domains, I don't think.

The SOS approach took the notion of mathematical purity from denotational semantics, including the importance of a small state and the avoidance of the mechanisms

of abstract machines. This was added to the intuitive simplicity of transitions from the operational work of many Edinburgh researchers. The other important idea from denotational semantics is SOS' structural nature, analogous to the denotational compositionality principle. This allows meanings to recurse over the structure of the syntax, following Burstall's notion of structural induction.

On his return to Edinburgh, Plotkin did not teach semantics to students. Instead, Burstall taught a course on SOS—rather than denotational semantics or his own early logic-style approaches. In the intervening time, Burstall had become a Buddhist monk, and remembers his early nervousness at appearing to teach wearing his monastic robes:

> When I went back to the Edinburgh Computer Science Department, I was a bit shy the first day, so I didn't put the top thing on. I just put on the yellow blouse and the skirt. Nobody said anything, of course. The next day I put on the whole thing. And then I started teaching this course on operational semantics, in my regalia. I gave my lecture, on operational semantics, and then I said "Any questions?" One guy put up his hand and said "Could I ask about your apparel?"
>
> So I said "Well, you see, I explained to you there's operational semantics, and there's denotational semantics." And I said "People doing operational semantics wear this kind of clothes."
>
> After that, I got on quite well with the class.
> (Burstall 2017)

Plotkin still likes SOS (at time of writing) and would, if forced, use it as the tool if he was asked to define a modern programming language (Plotkin 2018). However, it is important to recognise that the majority of his contributions fall more within the denotational area, including on domains, types, and function logic. That said, SOS represents one of the simplest and most intuitive ways to define the semantics of imperative programming languages. One advantage of SOS, says Plotkin (2018), is that it is "closer to computer scientists' intuition". Indeed, upon moving to Newcastle University in 1999, Jones switched to teaching SOS instead of denotational semantics, feeling that it was better to avoid the "heaviness of the maths" of the denotational approach (Jones, in Plotkin 2018). There are now many adapted forms of SOS for different uses; one of the most common is the 'Natural Semantics' (so-named because of its similarity to natural deduction) of Gilles Kahn (1987), which was used for the definition of Standard ML discussed above.

With the introduction of SOS, the lines of influences from earlier chapters reach something of a close. It would be wrong to call SOS the endpoint for semantics, and many other approaches still see use today, but around the early-mid 1980s a shift in focus among theoretical computing researchers was starting. Areas of study moved away from semantics and towards program verification, concurrency modelling, and other areas. These will be briefly discussed in the next and final chapter, which concludes the present work.

CHAPTER 8

Conclusions

In 1970, formal description of programming languages was regarded as the state of the art in computing; the gold standard for academic research. Bauer, who had been something of a formalism sceptic throughout the 1960s—as evidenced by his arguments against the creation of an IFIP Working Group on language description—gave a talk at the tenth anniversary celebration of IFIP on the history of computation (Bauer 1972, p. 70). His story went back as far as ancient methods of counting with pebbles and coins and came right up to date by referring to the ALGOL 68 description as a kind of pinnacle of achievement in computing: how far it had come since its humble beginnings!

Throughout the present work many reasons have been given for holding formalised semantics of programming languages in such high esteem. Expectation of formal description for new programming languages was, in the late 1960s and early 1970s, the standard; witness WG 2.1's surety that their successor to ALGOL 60 would be fully formalised. However, this began to change throughout the 1980s, as many researchers interested in formalism and programming languages began to shift their focus from full semantic description to reasoning about programs written in these languages. Today, of the vast catalogue of programming languages that exist now, only a vanishing fraction include a published formalism.[1] Some examples where

[1] Most languages have their compiler's front end parser built from a formalised syntax, but even this is rarely included in textbooks on the languages. Getting parsing correct is not a totally solved problem, of course—research is still ongoing, for example in the work of Scott and Johnstone (2010). The efficiency of these tools has also taken a lot of effort, as highlighted by Johnstone, Scott, and Economopoulos (2004).

formal semantics have been used to practical effect are given by Mosses (2001, pp. 183–4) and by Astarte and Jones (2018) (at the end of each section), but these are few and far between. It would be a mistake, however, to interpret the formal semantics effort as a failure because of this.

In fact, the impact of language descriptions went beyond the early goals of formalising every language. This chapter documents some of the ways in which this work affected computing in general, and wraps up the current work by exploring the themes raised by the stories told. Section 8.1 returns to the questions motivating this research presented in Section 1.3 and demonstrates how this dissertation has provided some answers. Section 8.2 explores the contributions made herein. Section 8.3 reflects on the methodology of this research project and discusses ideas for further work. Finally, in Section 8.4, some summary remarks are made.

8.1 Summary of key findings

Section 1.3 explains the focus of the research that led to the writing of the current work, and poses a number of questions that were to be answered. In the current section, these are addressed and summarised; in this way the major findings of this research are presented. The more straightforward findings are discussed first. Each of the subsections contains a one-line summary followed by a more detailed explanation.

8.1.1 Basic findings

The questions leading to these results are the basic 'what happened' questions: detailing of events and situations that surrounded the creation of formal semantics, including who was involved, what they were trying to do, and how they went about doing it. Some of the broader contextual questions are addressed here too.

Semantic description posed a number of challenges

Challenges for semantic description were presented by the innate complexity of programming languages; the sequential, ongoing nature of programs meant that semantics had to keep track of modifications made by commands. This was in contrast to previous semantics such as those of Tarski (1944) and Kripke (1963), who did not have to take into account this constant modification of value. The property was remarked upon by Strachey, who termed it 'referential opacity' and noted that it

caused problems for a mathematical view of computing.

Further challenges were introduced by particular control constructs in programming languages, especially blocks and scoping, parameter evaluation mechanisms, and jumps. Concurrency proved to be a major additional layer of complexity: no longer could there be a view of computation as a sequential flow from inputs or starting states into outputs and end states—and multiple results of programs could be equally correct.[2]

Different focus on challenges led to different approaches

This topic is of course answered throughout the entirety of this work, with a focus on the 'interpreter' or 'operational' style as developed by McCarthy and the Vienna Group, and the 'mathematical' or 'denotational' approach first presented by Strachey and later adopted by Vienna in the VDM framework. The former describes computation by considering the steps of change represented by each statement; the latter maps program components to functions that capture the same behaviour. Landin's method was an interesting combination of ideas from the two, and van Wijngaarden took a different tack entirely. Plotkin's SOS used simple operational concepts and presented them as inference relations to better enable the capturing of concurrency.

While the various approaches discussed in this work went in different directions, it is worth observing how certain core concepts appeared repeatedly in different guises. One particularly pervasive notion was the idea of a 'state' of computation before and after a particular command—appearing at the heart of both the VDL and denotational approaches. Even Caracciolo's (1966) highly abstract and language theoretic paper at FLDL still led to this idea. In a separate paper by the present author, it is argued that denotational semantics can be seen as "a lambda abstraction away" from operational approaches (Jones and Astarte 2018, p. 186). To demonstrate this, consider the following quotation from Lucas (1981, p. 556):

> Given a composite grammatical construct, e.g., a conditional statement or iteration statement, one would like to compose the state transition of the compound from the state transitions associated with the parts. A statement, primitive or composite, could then be said to denote a function, viz., the state transition function to be applied to the state when the statement is executed.

[2]See Jones and Astarte (2018) for a lengthier technical discussion of the challenges presented by programming languages and responses to them in the main approaches.

Lucas is describing the VDL approach, but could just as easily be talking about denotational semantics. Van Wijngaarden's approaches had similar notions at their hearts of maintaining information about parts of computation (whether in the 'truth list' of his early work or the abstract machine components of the ALGOL 68 description) but rather than utilise abstract operations to manipulate these entities, he kept his focus squarely on concrete symbolic representations.

Someone who addressed this fundamental similarity between core notions was Dan Berry (1985), who showed a connection between VDL-style operational and denotational semantics. In an email to the current author,[3] Berry summarised this:

> My conclusion was that the distinction between operational and denotational semantics, at least for deterministic languages, was notational, and for non-deterministic or concurrent languages, while one could systematically convert between the two kinds of specs, the operational one made the nondeterminism more explicit.

One place in which differences emerged across various groups was in the choice of base for their work, a decision which affected the ease of modelling different constructs. Strachey pointed out (in Walk 1969a, p. 9) that while assignment was hard to model in descriptions based on lambda calculus, recursion—a core concept in those descriptions—was tough to handle in certain other frameworks; but despite this the two were frequently used in real programming languages. Samelson also noted (in Landin 1966c, p. 292) that Landin's choice to base his approach on lambda calculus meant he had to make significant changes in order to handle some of the aspects of ALGOL 60.

Another difference, and one that frequently caused lengthy arguments, was the choice of notation. This might seem superficial, but due to the immense complexities involved in semantic descriptions, the choice really could make a difference: witness the disparity in readability between the Mosses (1974) and Henhapl and Jones (1978) ALGOL 60 descriptions despite their similar denotational bases.

The core differences across operational and denotational approaches resulted in different applications. The Vienna Group kept the notion of compilers firmly at the heart of their work in both the VDL and VDM periods, and this led into work on proving compiler correctness and from that properties about programs. Strachey always aimed for a goal of formalising the foundations of computing, so his aims

[3] Received in 2019.

were largely achieved once a semantic description was written. His collaborators, particularly Milne and Mosses, found ways to take denotational semantics work into more practical applications: implementation correctness through multiple abstract definition layers, and compiler generation respectively. The VDM denotational style, representing a fusion of ideas from Vienna and Programming Research Group work, was one of the more usable approaches, and it was used to create a full description of PL/I, ready for use as the basis of a compiler. Unfortunately, as the intended target machine was cancelled before its conclusion, its success as a product cannot be easily evaluated; however, Bjørner (1980) did create a compiler, for Ada, in this style—and in record time (Bjørner and Havelund 2014, p. 8).

In some cases, the choice of approach could make a difference to the usability by non-experts. Moore (2017), who achieved some commercial success with his LISP-based theorem prover ACL2, noted that operational semantics was the best way to engage with customers:

> One of the reasons that I like operational semantics is that our customers, the ACL2 industrial customers, most often are dealing with programs in languages that are idiosyncratic to the shop. They invent a microcode and they want to write programs in this microcode and prove them. Nobody's going to build a VCG[4] or a Hoare logic for the microcode because they may not even want to say what the microcode is. It may be proprietary. But they can build an interpreter for it. That's what they do to test their programs: they build interpreters—a simulator for their little in-house programming language before they realise it in hardware.

This is a rare example of a tool to support the use of semantics, something that Mosses (2001, p. 185) mentioned as a major inhibiting factor for the uptake of formal semantics.

Semantic descriptions grew in complexity over time

A general trend can be seen for each thread of research on semantics tending towards growing complexity; this echoes Hoare's comment (in Walk 1969a, p. 9) that "difficult things are difficult to describe". McCarthy, for example, started with a very simple approach to state definition and statements as functions transforming this; as his focus expanded to include jumps and compiler correctness, the descriptions became larger and more complex. This can also be seen in the work of Landin, whose applicative expressions—and the SECD machine used for their interpretation—were

[4] Verification condition generator; a system that indicates which properties need to be proved.

relatively simple at their inception, but had to be extended with more operators and components to handle, for example, jumps. The VDL approach was largely unchanged throughout its duration, but reaction to the problems of proving properties and manipulating the huge state ultimately resulted in a radical change in direction for the group towards a smaller state denotational approach. For its part, denotational semantics grew from Strachey's early functions from state to state to include also environments and the very powerful continuations. Reaction to this complexity led Plotkin to consider an alternative approach for describing languages and to create conceptually simpler SOS. While van Wijngaarden's two approaches seem radically different from their outward appearance, careful examination shows his focus on text manipulation at the heart of both, although the ALGOL 68 description is considerably more complex due in part to the large language under definition.

Semantics workers had different intellectual backgrounds

Workers on formal semantics fell broadly into two main categories: mathematicians and logicians, and engineers and practical programmers, with most sitting in the first group. This diversity was noted at the 2004 *Mathematical Foundations of Programming Language Semantics* conference; the panel all had different backgrounds: Jones had worked in industry since leaving school; McCarthy was a mathematician; Reynolds's background was in physics; and Scott was a logician (Jones et al. 2004).

Various people brought different ideas to the field: some mathematicians, like McCarthy, were interested in improving the mathematical aspects of computing, but Strachey, arguably the loudest voice arguing for formalised foundations, was, in his own words, "not a mathematician" (quoted in Penrose 2000, p. 83). Landin (2001) and Scott (2016) both separately noted that their backgrounds learning lambda calculus and universal algebra provided the perfect skills for approaching language semantics. Van Wijngaarden's interest in practical mathematics for engineering problems may be a contributing factor in his concrete approach to language description; Gorn, on the other hand, stuck to a machine-focused view reflecting his self-image as a practical programmer. Mahoney (1988, p. 125) notes that by observing the historical model chosen by certain computing practitioners as their framing device, we can analyse the way their work was presented. He adds that we should be critical about this—take the example of McCarthy who had, as seen in Section 3.1, taken pains to give his work a fine mathematical pedigree, but despite this his work did not see as much direct impact within mathematics as he had hoped.

Despite this variety of backgrounds, the various people in this story all came to formal description through experiences with programming languages. McCarthy had worked on LISP, Landin with Strachey on autocodes, Strachey himself with CPL, and the Vienna Group (as well as many others) through ALGOL and then PL/I. Trying to implement programming languages taught them the difficulties of language complexity and the importance of understanding precisely what the language was meant to express.

Different institutions supported different styles of work

The provision of funding for the description work is interesting. IBM funded Zemanek and his group for a long time, but his Vienna Lab was expensive to the company due to its army of support staff. Zemanek felt that he was better placed at IBM to research large topics on a grand scale and turned down offers to move to universities—even when TUW suggested he found a school of informatics—because they would not have given him sufficient scope to investigate and teach the ideas he felt most important (Zemanek and Aspray 1987, p. 64). However, the tension between the group's intellectual goals and their obligation to IBM to provide usable products led eventually to the group's focus being shifted. Strachey, on the other hand, struggled for some time in the early 1960s to get financial support for language description work (Strachey 1971a), and later, in the initial stages of the Programming Research Group, funding was very uncertain. However, Strachey was given a free hand to choose his research direction at the Group, perhaps due to the success of the 1964 summer school he co-organised. This allowed him to create "a highly critical and thoughtful atmosphere in which *ad hoc* or superficial ideas [were] given very short shrift" (Strachey 1971a). The Programming Research Group's direction was much more obviously theoretical than the Vienna Lab's, but Strachey was always keen to keep practical concerns in mind, and Stoy (2016a) argues that many of the researchers there made important contributions to both theory and practice. Plotkin (2018) notes that during the late 1970s in Edinburgh, the University's wealthy status permitted the existence of many research groups with different focuses and the interactions between these groups often led to the best work.

Interactions between organisations were also important, for example between IBM and IFIP. While IBM provided support for IFIP in some respects (see their sponsoring of FLDL, and paying for Zemanek's assistants while he was IFIP President), they were less keen to allow influence on their products. TC-2 had hoped to be involved in the design of PL/I but were rebuffed, and indeed SHARE were also

taken out of the later stages of that language's creation. A major problem was that rather than involving users in the design, the committee instead listened to what every user said they wanted to be able to write rather than considering the deeper semantic structures required. As discussed in Section 5.3, this resulted in a language of massive, unwieldy size.

Formal semantics was a key part of a European style of computer science

Considering the broader context of location of research, it is very clear that the majority of work from this story took place in Europe. Mahoney (2002, p. 29) placed semantics as one of the three main pillars of theoretical computing in the period 1955–1975 alongside automata and complexity theory. It is interesting to observe that 'theoretical computing' in Europe tended towards the first category, whereas in the US more towards the second two. Plotkin (2018) remembers being at a large American conference on theoretical computing and being faced with an empty room for his presentation on semantics. Moore (2017) had noticed this divide himself upon moving to Scotland in 1970:

> The contrast between my life at MIT as an undergraduate and my life in Edinburgh as a PhD student was highlighted, exemplified, by the difference in the attitude toward computing.
>
> At MIT, there was more than enough computing power. It was something that I had been told at MIT: if you wanted to see if a program worked, just write it and run it and see! You're not going to break the computer, and if you do, it doesn't matter, we'll just reboot it. It was a very engineering, resource-rich environment.
>
> Then I came over here and suddenly the computer was this precious thing that only eight people could use at a time, and even they couldn't use it too much because the memory was so small. The idea was: don't sit down and hog the terminal until you know exactly what you're going to do and you make the best use of every key stroke. Plan it out on paper, figure out what it's supposed to do, figure out how you're going to find out if that's what it does, and then when you're ready, get on the machine and try it.
>
> It was the difference between a resource-rich engineering environment, very experimental, in America—or at least at MIT—and very theoretical, treating the computer as this delicate, holy object.

This divide was something noted by Mahoney (1988, p. 123): in the 1970s, there were "fundamental differences between the formal, mathematical orientation of Eu-

ropean computer scientists and the practical, industrial focus of their American counterparts". Moore suggested a possible reason for this difference:

> In a bigger sense, I think Europe tended, at least during the invention of all this stuff, to be poorer, partly because of the war. Rod Burstall can tell you stories about being surprised when he was offered two eggs for breakfast! Whereas in America, the war didn't affect us that way at all. I was born after the war, so I wouldn't know, but from everything I've read—and I've read a lot about World War II—Britain, and Europe in general, was just really hurting for ten or fifteen years after 1945, and well into the '60s. I came here in the '70s and I could still feel it.

Plotkin (2018) also suggested that Americans were simply more pragmatic about programming. This could be related to the earlier commercialisation of computing machines in the US, where in the UK they were products of research institutions for much longer. This also meshes with the views of McIlroy, whose view of Europe as the only place in the 1960s with proper computer scientists is discussed in Section 4.2.

Other projects, especially programming language work, were influential on formal semantics

At the Oxford Programming Research Group, different kinds of research were happening alongside formal semantics—as described by Stoy (2016a)—and this included the development of an operating system as well as the CPL language near the beginning of the story. Stoy's argument was that the focus on practical applications kept the group grounded. In contrast to the diverse projects happening in Oxford, the Vienna Lab's focus was almost entirely on the formal description of PL/I. A few other projects happened alongside at the beginning of their run (such as the vocoder work) and some other products developed out of the language description, such as the text preparation system worked on by Schwarzenberger. At Edinburgh in the late 1970s, there was an atmosphere rich in theoretical computing, especially program verification and concurrency modelling. As discussed in Section 7.4, this deeply influenced Plotkin's work.

A hugely important factor in the development of almost all the language description techniques described herein was the influence of a central programming language. This is most clearly visible with the Vienna Group and PL/I, but affected most of the other people in this story too. McCarthy had created LISP only a few years prior to writing semantics and, as argued in Section 3.1, his description technique had obvi-

ous influence from LISP. Strachey's decision to embark on a rigorous technique for semantic specification was prompted by his experiences with CPL, and van Wijngaarden's two-level grammars were created specifically for ALGOL 68. Even Landin's work was inspired by the process of writing Mercury autocodes. In all these cases, there is a marked correlation between the languages and the description techniques. It is interesting to observe that Plotkin's SOS was not written with any particular language in mind and is one of the most generally applicable styles mentioned here.

Furthermore, nearly all researchers discussed in this work presented a version of their technique applied to either a full ALGOL 60 description or at least some 'ALGOL-like' language. It seems unlikely that the ease (or otherwise) of modelling ALGOL 60 influenced design decisions, but the fact that a common demonstration language was used by all indicates the importance of ALGOL 60 as a sort of litmus test for description.

8.1.2 Main focuses

In this subsection, the findings resulting from the main focuses highlighted in Section 1.3 are discussed.

Most motivations were either 'theoretical' or 'practical'

Although all could be broadly summarised as 'getting to grips with languages', a number of motivations were put forward for writing formal semantics. These fell mainly into two categories, the first theoretical and the second practical: formalising the foundations of computing, and developing correct compilers.[5]

Formalising the foundations of computing was seen by McCarthy as fundamental to placing computation within mathematics, and by Strachey as necessary for understanding what happens when one writes programs. Zemanek (1965, p. 141) quoted Russell who made similar arguments for mathematics:

> Russell once gave four reasons for formalization which still apply to both theory and practice: (1) security of operation is assured, (2) tacit preassumptions are excluded, (3) notions are clarified, and (4) resolving structures can be applied to many other problems.

[5]Notable exceptions include van Wijngaarden's design of ALGOL 68, where the goal from the start was the creation of a language; and Plotkin's SOS, which was intended primarily as an educational tool.

This mathematisation of computation achieved a good deal of attention in the computing field, but had little impact on the mathematics community as a whole. Evidence for this is provided by examining the citations to McCarthy's work on this subject: there are plenty made by computer scientists, but few from broader mathematicians. Formal and theoretical computing, however, is still a very important area of research and in many ways can be seen as growing from the roots of semantics work (this is discussed in more depth later in the current Section).

Correctness of compilers was a major driver for McCarthy who worked on the topic towards the end of his interest in semantics; but in practical application this work saw very little success. This was in large part due to the extreme difficulty of the proofs involved as the languages became more complicated. The Vienna Group also experienced this: while they (especially Jones and Lucas) managed to prove some properties about their compilers, due to the complexity of the VDL description technique their achievements were frequently overlooked.

One early desire for formal semantics is that it would compensate for the loss of intuitive understanding once high-level languages were widespread and their actions no longer corresponded obviously to those of machines. Unfortunately, the large size and complexity that emerged every time realistic languages were modelled meant that this was not a great success. Instead, programmers learnt to understand high-level programs intuitively, as Strachey (1973a, p. 2) pointed out. Related to this idea, Landin (1965a) wrote that the correspondence he had established between ALGOL 60 and his applicative expressions would serve to clarify understanding of the language and also, conversely, those familiar with ALGOL would be able to use that to follow his method. This was taken further by de Bakker (1967) who used ALGOL 60 to define his metalanguage, which was then itself used to describe ALGOL.

These ideas are linked to the notion that writing a formal description of language helps the writer understand it better; as Reynolds pointed out (in Jones et al. 2004), this would then highlight areas of the language which could be improved. This was acted on by, for example, Landin (1966b) and especially Tennent (1977). That formal semantics should be used very early in the language design process was one of Strachey's main tenets; Schmidt (2000, p. 89) described how Strachey "led a 'second wave' of language-design research, where emphasis shifted from syntactic structure to semantic structure". This brought a much stronger foundation to programming languages: "Strachey knew that a competent language designer intuitively employs a 'meta-theory' of language design; by making this meta-theory explicit, one inserts

solid engineering into the language-design process" (Schmidt 2000, pp. 89–90).

However, the creation of a full language description prior to implementation was very rare; this was noted by Reynolds who remarked (in Jones et al. 2004) "semanticists should be the obstetricians instead of the coroners of programming languages". One notable exception is ALGOL 68, which was developed through a formal description right from the start, but due to its size and complexity was widely criticised; another is ML (Harper, Milner, and Tofte 1987). Plotkin (2018) remarked "I think denotational semantics has had essentially zero influence on programming languages, and operational semantics has had only a little bit, through ML. I think there needs to be greater interaction with language designers, though that is easier said than done". Jones added (in Jones et al. 2004) that the use of semantics in language design was rare because semanticists had failed to create a literature which was actually helpful to language designers.

Collaboration was key in the successful development of formal semantics

A number of events brought together the diverse group of practitioners working in the field of formal semantics. The 1963 *Working Conference on Mechanical Language Structures*, while not directly concerning semantics, led to much early debate of the important concepts, and its discussion (Gorn 1964) provides many insights into the views of the attendees. The really key event was the 1964 *Formal Language Description Languages* conference, and here the participants gathered in sufficient force to determine an agenda for their field. This led to the creation of IFIP's Working Group 2.2, which was a vital venue for researchers to meet and debate their ideas. A third conference, *Programming Languages and Pragmatics*, took place in August 1965, and was seen by participants as providing the finishing part in a trio of conferences corresponding to Morris' division of linguistics: WCMLS was largely syntax, FLDL semantics, and PLAP pragmatics. These were the really crucial conferences that served to ignite the field of language description in its early days; of course many others also took place later.

IFIP's Working Group 2.2 was formed with one of its major aims being to bring together different workers on language description, but it was marred by constant bickering and infighting amongst different practitioners over what were ultimately rather trivial differences, such as representation details of metalanguages. Although Strachey had always maintained that content was a much bigger concern than form, he was certainly not immune to these debates. Discussions also played out within WG

2.1 over the complex metalanguage used by van Wijngaarden for ALGOL 68. WG 2.2 and WG 2.1 also show the difficult task of agenda setting, as discussed in Sections 7.1 and 7.2. Here can be seen the attempts to establish and demonstrate credibility for their areas of focus, and disagreements over what the priorities should be.

Another way in which researchers in this field were brought together is through extended academic visits, a number of which were pivotal to the development of different techniques. Perhaps the most obvious is Scott's visit to Oxford at the end of 1969, but the time Scott spent in Vienna in the summer of 1969 was also critical as it laid the stage for the Lab's later switch to a denotational framework. Landin's time with Strachey at the latter's consulting business was the period in which both men began working out their approaches to semantics and their collaborative period—although not leading to any joint publications—was essential to the first growth of denotational semantics. Landin took his turn to host in 1968 when Bekič stayed at Queen Mary's College for an extended period; this was another important contributory factor in the later Vienna work. A further example of an important sabbatical is the time spent by Plotkin at Syracuse with Reynolds: this came just before Plotkin got fully into language description work and is mentioned by him as a crucial period (Plotkin 2018).

Influence mostly existed within a given style or group

Influences did occur between different researchers, although most often these were within a particular circle. The Vienna Group had a number of people all working together sharing ideas, and Strachey worked alongside first Scott and then many of the students at the Programming Research Group on all the developments to his core semantics idea. Reynolds' work on formal semantics was inspired and influenced by the long sabbaticals he took in Europe, including with Landin. SOS was largely the work of Plotkin, but it was strongly influenced by the environment at Edinburgh University and especially the work of Milner. In contrast, ALGOL 68's metalanguage was mostly developed by van Wijngaarden alone (although other authors did contribute) and, as discussed in Section 7.1, the continual inclusion of language features into the description led to its ultimate huge size.

Clear influences from McCarthy pervade all the semantic styles discussed in this story: state, abstract syntax, and interpretation functions. Landin's use of a machine with many components and lambda calculus was also highly influential for many different approaches. One should acknowledge as well the influence of Rod

Burstall in a number of places: Scott for example credited him with the idea of using a function from locations to contents to model store in denotational semantics, and Burstall supervised and worked with Plotkin during the run-up to his developing SOS.

The negative reactions displayed by practitioners to each other in discussion forums such as WG 2.2 may have contributed to the extremism that emerged. An example of this is the constant overlooking of the successes of the ULD effort: frequently, at WG 2.2, Lucas would explain that the challenge under discussion had already been handled by the Vienna Group, but the responses to this were lukewarm at best. One rare example of a strictly positive influence is that of the Vienna Lab switching their deep focus from operational to denotational semantics; much more commonly, writers stuck to their styles and did not shift fundamentally.

Formal semantics faced heavy criticism

Right from its very start, formal description of programming languages was met with opposition. A common criticism was the unintuitiveness of large chunks of formulae and the consequent loss of correspondence with machine operations. One such critic was Gorn: as seen in his paper at *Formal Language Description Languages* and his other work around the time,[6] Gorn much preferred an approach which kept the notion of machine clearly in mind—this is reflected in his line "I am one of those extremists who feel that it is impossible to separate a language from its interpreting machine" (Gorn 1964, p. 133). This view was echoed particularly by those like Warshall who saw themselves as practical programmers, and by Samelson who said to McCarthy (1966, p. 11):

> I would rather avoid the word "semantics" altogether. Semantics is what we have in our heads; as soon as we write it down it's not semantics anymore.

A related, more philosophical, concern was raised by DeMillo, Lipton, and Perlis (1979) and later Fetzer (1988), who argued in essence that any kind of formal modelling was inherently flawed because proofs dealt with abstract models, but programs and their machines were physical artefacts, the properties of which might be sufficiently different to render the proofs invalid. Even Zemanek (1980, p. 14) made an argument like this: "No formalism makes any sense in itself; no formal structure has

[6] See Section 4.3.

a meaning unless it is related to an informal environment [...] the beginning and the end of every task in the real world is informal". Another angle to the attack is the idea that proofs should be social rather than formal processes. This debate was nuanced and at times acrimonious but is covered well by MacKenzie (2001, ch. 6) and Eden (2007). The discussion is still ongoing amongst researchers in history and philosophy of computing, such as Daylight (2018).

Formal semantics can actually be seen as part of the *response* to this challenge. The problem is trying to determine whether the machine is doing what the programmer wants; this is a complex and difficult task and more easily handled when split into parts, as Stoy (2016b) explained:

> We want to be confident that our computer programs do what we designed them to do. Easier said than done, because it means you have to be able to write down what you want the program to do, so at some stage you have to get the designer's thought processes into some form of mathematical description, and then you have to have a semantic definition of the programming language [...] You have to know what the semantics of that is, and then you have to have confidence in the compiler, and then you have to have confidence in the hardware, so it's a heavily nested set of confidence-inspiring procedures, shall we say. But that includes a definition of the semantics of the programming language as part of that.

An example of this in practice was accomplished by J Moore with his NQTHM theorem prover. Moore (2017) explained:

> We did the stack in the late '80s. The computational logic stack. That was the thing that had gates, assembler, a linker, a loader, compilers, an operating system, and applications written in high-level languages. We proved that each of those levels was correctly implemented on top of the level below it. Then we had a few really trivial applications like a program that played a certain game and proved that it had a winning strategy.

A formal semantics of the languages involved at every level was essential for this (Moore 2017):

> If your goal in life is to prove theorems about programs, then semantics is the way you get from this informal engineering artefact that is a program to a formal system where you can prove things. From my perspective as a theorem prover, semantics is just something I have to have in order to get my formulas.

A more prosaic reason for criticism of formal semantics was the immense complexity of most full-language definitions. However, small examples in every style indicated that it was relatively trivial to give semantics to a simple language with only a few constructs like assignment and iteration; but as languages became more complex so too did their descriptions. This distinction between complexity of object language and description was often overlooked. The ULD descriptions of PL/I and van Wijngaarden's ALGOL 68 description both received great criticism for their size which was really due to the language under definition.[7] An interesting corollary to this is that both the Vienna Group and van Wijngaarden were criticised for seeming to have more interest in the general applicability of the metalanguage than tailoring it to the language in question. So these semanticists were attacked on the one hand for the large descriptions resulting from sticking precisely to their object languages; and on the other hand for not being sufficiently tied to the object languages.

More compact language descriptions had typically achieved a smaller size by ignoring certain programming language constructs. As Landin noticed, particular aspects of programming languages like assignment or jumps force the language designer to make increasing contortions of their method. This argument over breadth of expressiveness versus readability of output reoccurred frequently at WG 2.2, as shown by the discussion below (from Walk 1969a, p. 6):

> Caracciolo: A reduction to simpler questions would mean to omit the proper problem.
>
> Scott: Only the most primitive, non-problematic things have been dealt with using this approach.
>
> Laski: A language definition should specify as little as possible.

This was one reason for many people choosing to write a description of the (relatively) minimal ALGOL 60 as a way to showcase their definition methods. According to Wadsworth (2000, p. 132), Strachey tended to prefer modelling smaller languages to show off proof of concept (as seen in the first publications on continuations), but

[7] An exchange from WG 2.1 (in Turski 1967, p. 7) illustrates this:

> TURSKI: In Grenoble we decided that the proposed description method is a milestone in the development of the language.
>
> RANDELL: A milestone or a millstone?
>
> General laughter follows.

in the Adams Essay Strachey and Milne chose deliberately to model a large, full-featured language. This resulted in the complex essay and book. Mosses (2001, p. 185) wrote about this challenge as well:

> Few frameworks scale up smoothly from the tidy illustrative languages described in text-books and papers to languages such as C and Java—even Standard ML, a clean language designed by theoreticians, turned out to be a considerable challenge to describe accurately in Natural Semantics.

One vocal critic of formal semantics was Naur, who had been prepared to fight for formalised syntax of ALGOL 60 but by 1981 had come to see huge problems in the formalisation of semantics. He argued vehemently against the use of formal specifications and attacked the VDM description of ALGOL 60 (Henhapl and Jones 1978) in particular. Naur (1981a, p. 445) summarised his criticism of the ALGOL description thus:

> Compared with the standard description of the language the more formal description is quite incomplete with roughly the same size; an examination of a small part of it has revealed numerous errors and inconsistencies, and the proof of its consistency appears to be impossible, or at least so impractical that it has not been done by its authors.

There are three related challenges here that need to be disentangled: formalisation of every part of the object language; the use of only partially-specified constructs; and the explanation of the metalanguage. While many descriptions certainly did ignore trickier parts of languages (like 'own' variables in ALGOL 60), to cope with the complexity problem discussed above, there *were* complete language definitions, and so this was not necessarily true of every language descriptions.

Incomplete specifications, like the inclusion of an arbitrary set selector in the Allen, Chapman, and Jones (1972) definition of ALGOL 60, were a problem if their existence needed to be relied upon in proofs, but given that critics also tended to argue that intuitive descriptions were more useful, the occasional use would appear acceptable. The alternative, especially for difficult properties like non-determinism, was often very unintuitive: witness the bulky control tree of VDL or the heavily mathematical power domains.

Understanding the metalanguage was a serious problem. Backus (1979), argued that programming languages had become very complex in themselves and adding an arcane description only added to the problem:[8]

> Because it takes pages and pages of gobbledygook to describe how a programming language works, it's hard to prove that a given program actually does what it is supposed to. Therefore, programmers must learn not only this enormously complicated language but, to prove their programs will work, they must also learn a highly technical logical system in which to reason about them.

McCarthy, as seen particularly in his (1966, p. 11) FLDL paper, had anticipated such criticism from the beginning, writing "I have written down this notation, and unless it is substantially simpler in some intuitive way than ALGOL, you can say, 'What has been gained?' I have merely given you a new language to learn". But he argued that a similar criticism was levelled at Tarski's semantics of mathematical logic, and that work had nevertheless provided interesting and useful results. An unusual take on this problem was given by de Bakker (1967), whose ALGOL 60 description had a circular notion whereby understanding ALGOL granted insight into the metalanguage, and vice versa. Others, such as the Vienna Lab, used careful and lengthy natural language introductions to give their metalanguage meaning (e.g. Lucas, Lauer, and Stigleitner 1968); after all, as Duncan (1966) had pointed out, natural language is 'our ultimate metalanguage'.

Ultimately, Stoy (2016b) argues that the main reason formal semantics did not achieve the popularity it once expected is because it was not necessary. Using natural language definitions seemed not to have caused any tremendous troubles—so why bother going to the expense of changing this? Plotkin (2018) echoed this, explaining that real success would have required deep engagement with language designers, something that was very difficult to achieve.

Semantics research had an important impact on computer science

In the middle of the 1980s, a trend was emerging: formal researchers were shifting away from writing full descriptions of programming languages towards rules for reasoning about programs written in the language. Prompted in large part by the growing success of Hoare's axiomatic method, this was a reaction to the difficulties

[8] As seen in his Turing Award lecture, Backus (1978) had begun to favour a functional rather than procedural approach to programming as a result of this concern.

of dealing with the large objects that full languages represented: the problem could instead be made smaller by considering only individual programs.

Formal techniques for program verification grew mainly out of VDM work, axiomatic semantics (which lent itself particularly well to proof of programs by construction), and the Edinburgh work on formal proof. J Moore, who studied at Edinburgh under Burstall, developed theorem provers utilising ideas taken from McCarthy's use of LISP as a logic to specify programs in an operational way using an interpreter and state (Moore 2017). As quoted above, Moore believed formal semantics of a programming language was essential to being able to prove properties about programs written in that language. Interestingly, formal semantics was sometimes applied to specification languages themselves; examples are Clear (Burstall and Goguen 1980) and Z (Spivey 1988). A good overview of the history of proving properties about programs is written by Jones (2003b).

One curious difference in the program proving field between Europe and America is highlighted by Moore (2017):

> On top of that, you had Tony Hoare, who basically regarded semantics and program correctness as a theoretical problem to be addressed by the invention of a logic, whereas John McCarthy and Woody Bledsoe and that lot regarded it—again—more as an engineering problem where they wanted to see if they could build a machine to do it. So by the '70s, in my view, Tony was teaching and getting a lot of people to see it as a logical and theoretical problem, whereas America was building tools, and wanted to actually use the tools.
>
> It's a pity—from the American perspective—because in order to use the tools you have to have the theoretical understanding and the mathematical finesse to actually come up with the formulas that these tools could prove. It's not like the tools are independently powerful of the theory; you need the theory to use the tools, but you don't need the tools to use the theory. So there's a sense in which I see Europe as better positioned because they can just get the tools from us, if we ever got a tool that was good enough! The point is, the tools are something you can more easily acquire than the knowledge and the attitude.
>
> So I regard Tony as having done Europe a great favour—and not just Tony, but Edsger Dijkstra, and Robin Milner, and Christopher Strachey. That whole generation of European computer scientists came at it in such a way that when the tools are ready, Europe will be better positioned to exploit them.

This lack of good tools was, as mentioned, something that Mosses (2001, p. 185)

had highlighted as a barrier to the uptake of formal semantics.

Related to the concept of program verification, and growing in large part out of axiomatic semantics, was the idea of unifying theories of programming. One person who worked on this was Tony Hoare, who had been first exposed to the idea through the works of his student Lauer (1971) whose thesis was on determining consistency of axiomatic semantics with respect to an operational model; they later published joint work (Hoare and Lauer 1974).[9] While Hoare was the head of the Programming Research Group during the 1980s, he saw a number of people there with independent theories of programming and decided he would like to bring them all together (Hoare, in He 2018). This resulted in a book authored by Hoare and He (1998) which used algebra to unify these various approaches in much the same way that unified algebras of number had done in mathematics. This was not an easy task; Scott (in Walk 1969b, p. 18) had argued that "The concept of a programming system is so abstract, it is not surprising that there is no uniform method for describing semantics". Indeed, this was a large part of Milne's contribution to the Adams Essay and subsequent book; Stoy (1977) too devoted a chapter (13) of his book to non-standard semantics, and introduced Strachey's concept of a 'semantic bridge' linking different definitions. Woodcock (in He 2018) claims that today He and Hoare's algebras have found use in industry for bringing together homogeneous models of real-world systems.

Another area of work flowing from formal semantics is type theory, according to Schmidt (2000, pp. 96–7). In particular, this was influenced by the semantic domains of denotational semantics, and the context conditions used in other methods. Rich typing systems give users the ability to specify their own types and the properties of these, which give structure and inherent properties to programs using them. This is one of the ways in which theoretical and formal computing has moved much more into mainstream programming and is one of the better success stories of formal semantics. A good historical treatment of the development of type theory, particularly in regard to its origins in logic, is provided by Cardone and Hindley (2006, § 8).

A final area in which formal semantics made a real impact is in the concept of programming languages as objects of study. This was started by the development of ALGOL 60 as a machine-independent language, as argued by Priestley (2011, § 9.3). Formal semantics contributed significantly to this by providing methods by which languages could be studied, and tools for examining and constructing lan-

[9] This work was not widely known and the more common reference is to Donahue (1976).

guages (Schmidt 2000, p. 90). Tennent (1977; 1981) was one particular researcher who took this idea and provided fruitful results.

8.2 Contributions of this research

"History is *unavoidable*, whether one likes it or not." So wrote historian of mathematics Ivor Grattan-Guinness (2005, p. 6). He argued that in mathematics, all work is influenced by and built on the work that preceded it—despite the best efforts of many mathematicians to present their work as fresh and complete by itself. The history is present in the ideas considered so basic that they need no explanation, in the specific choice of question, and in the very framework of presentation. The situation is no better in computing, a field with a (quite understandable) obsession with being forward-looking. However, history can be included purposefully and usefully in modern work and teaching, giving practitioners something to refer to in order ground their work, and providing a useful distance at which key ideas emerge. Doing so requires a strong body of literature setting out the stories of computing. The present work is an attempt to add something substantial to that body, so that anyone with a desire to understand something of the history of formal semantics can have at least a starting point.

This section is a summary of the contributions made in the present work, which together form a unique piece of material so far not seen in the literature on history of computing. It shows that a careful study of formal and theoretical computing tells an interesting story about the impact (or lack of it) on computing at large.

The technical parts reveal connections between historical work that might otherwise have been missed and allow the conclusion that for all the major surface differences, there are many deep similarities between approaches to writing the semantics of programming languages. These ideas show the long-running influence of McCarthy, who demonstrated that simple languages are relatively easy to define; many of his core notions such as 'state' remain relevant in nearly every method of understanding of programming languages. Technical reading also shows the contributions of Burstall, and the variety of important concepts laid out by Landin. Knowing the difficulty of modelling certain programming languages suggests that this may have been a contributory factor in their disappearances: witness that jumps caused problems for most modelling techniques, and were shown to be removable by van Wijngaarden; this may have prompted Dijkstra (1968) to write his famous 'Go to

Statement Considered Harmful' letter. Understanding the real technical challenge presented by handling non-determinism brings insight into why so many different approaches were proposed.

The historical material gives context to the technical development and allows explanation of some technical decisions. For example, many description writers were strongly influenced by the struggles of handling the design of large languages such as CPL and PL/I. Considering the very different backgrounds and motivations of the various actors helps explain why they chose to attack the problem of language definition from different directions. The current work should be a good source for future historians as a lot of material has been gathered in one place, including some more overlooked aspects. An example of this is the *Formal Language Description Languages* conference which was crucial in the formation of the field of formal semantics and arguably by extension theoretical computer science as a whole (at least as it existed in Europe).

One important theme that emerges is the importance of formal semantics in the foundation of theoretical computer science. As a part of formal language description, work on semantics was a central part of early theoretical computer science in the European style.[10] A full formal description was a key goal for many early programming languages, such as ALGOL 68, PL/I, Ada, and CPL. Although this was not achieved frequently, it set standards for thinking about programming. Difficulties in coping with the large entities needed to represent programming languages prompted a shift towards reasoning at an individual program level instead. Verifying individual programs and reasoning about concurrency in individual cases was a smaller challenge, and so may be an explanation for the shift away from full language description as a goal.

The current work also demonstrates the human side of all the people involved, with their own hobby horses and agendas. This led to difficulty in establishing agreed focuses and constraints for the field of work, as seen in the WG 2.2 echo chamber, with people shouting over each other about the differences in their approaches rather than attempting to compromise together. This was, of course, strongly affected by the disaster of WG 2.1's attempts to create the successor to ALGOL 60, and the subsequent fear of trying to produce a shared outcome. So the story of semantics paints a picture of scientific progress as it really is: no lone geniuses making amaz-

[10] As discussed earlier in this chapter, there was quite a difference in the styles of computing in Europe and America in this period.

ing breakthroughs alone, and no 'Whiggish' series of inexorable steps towards an enlightened future, but rather a mass of confused and argumentative people trying their best to figure out what to do.

Finally, the most important contribution of this work is its originality. The story of formal semantics has not been told in its entirety anywhere else; elements of it show up in other places such as *Mechanizing Proof* (MacKenzie 2001) and Strachey's biographical note (Campbell-Kelly 1985), but the current work is the longest and most thorough treatment. It shows part of a trend towards the history of computer *science* and offers a good gathering of sources for more focused studies. Even a document of this length is not enough to fully complete the story, of course; discussion of what more could be done is presented in the following section.

8.3 Reflection and further work

For this work, a balance between technical and contextual information was deliberately chosen. Plenty of work already existed giving technical summaries of formal semantics, including a piece by the current author (Jones and Astarte 2018), but it was desirable to get a broader picture—while keeping technical details still present. This could have been omitted for a much more standard 'history of science' approach, but this would have been less attractive to computer scientists, who it is really important to engage in history of computing.

A conscious decision was made to limit the scope of the work in order to tell a deep and compelling story about two major threads (Oxford and Vienna) and some associated inputs and outcomes. The work could have been written at a higher level about more areas, and could have come closer to the present day, but staying further in the past gave better historical distance, and narrowing scope allowed more depth. Coming closer to the present would have allowed, for example, the conducting of more interviews—but the origin story of formal semantics would have been an essential prequel anyway. Further work could certainly address more recent developments.

A number of risks for the present work are identified in Section 1.4 and these should be addressed. The present chapter indicates that while formal semantics did not achieve the success many early practitioners had hoped for, lack of positive direct use should not been as a complete failure. A great deal of impact in other fields was highlighted above; the in-fighting between semanticists in different camps also serves to cloud that there were a number of engaged and productive workers. The

present work considers a broad range of techniques for formal semantics, with some exceptions mentioned below.

Choice of sources is an extremely important decision for any historical work, and the decisions made for the project reported here should be evaluated. Archive sources were extremely useful, especially the Strachey Collection with its huge number of documents and items. However, its size perhaps counted against it in some ways: it was much too big to cover in its entirety within the timeframe. Having to head to the archive without yet having a deep understanding of Strachey's life and works meant the selection of boxes to check was made in a somewhat scattergun manner. Some interesting and relevant documents were no doubt missed because of this, and there were lots of technical papers which could have been studied in much more depth to really tease apart the development of ideas. However, choosing some boxes reasonably arbitrarily did result in finds which could have been missed otherwise (like the academic references). There is easily enough material in that archive for a full-length biography of Strachey, which would be a great avenue for further work.

The IFIP papers (stored in various locations) were useful sources for getting down facts like membership of committees and timing of events, but even more so for their recording of discussions. The minutes of meetings are clearly an edited form of the actual conversations that took place, but still give a good idea of the tone of meetings, the bonds between various people, the jokes, and the criticism. Plenty of IFIP material exists in different places and it could easily be gathered together into a full-length institutional history.

However, some archive sources that would have been good to use, such as the collections of Zemanek and Landin, were not accessible during the research period. The van Wijngaarden archive had plenty of material but as it was not organised, archived, and indexed, the gems were buried and hard to find in an appropriate time. Some extra chasing could lead to some great finds: for example, the tapes of the *Formal Language Description Languages* conference could very well be stored in the Zemanek Nachlass.

Oral history testimony is used carefully throughout with appropriate caveats included; note, for example, the scepticism with which some of Zemanek's claims are reported. Interviews are not perfect sources, and the approach chosen for this project led to some angles not being covered (in large part this was due to many of the ideal interviewees being dead; that said, talking to people who were close to

them got another side of the story). The memory of interviewees is, of course, prone to fallibility: misremembering, forgetting, and re-interpreting old events. One has to be sensitive about these issues when doing interviews rather than confrontational. An area ripe for further interviews would be with people whose work was impacted by formal semantics, even though they did not produce it themselves. Interviewing J Moore was very useful in this regard, and doing more would have allowed better exploration of the impact and influence of formal semantics. The transcription process was lengthy, but was not as bad as initially feared, so ultimately doing them personally was probably the best decision.

It would have been preferable for all the interviews to have been conducted by the current author, but it turned out to be much easier to arrange some interviews with Cliff Jones present as well. Having two interviewers was manageable but did lead to different outcomes: some interviews became more of a conversation than a classic interview. Additionally, it was unfortunately not always possible to record in an ideal environment; but making these concessions was worth it to get an interview. The decision to keep the interviewer out of interviews as much as possible did mean that some ended up straying off topic in some places; while not keeping focused did take up more time, in many cases these tangents also led to interesting stories that could not have been predicted. Asking specifically for anecdotes about people sometimes led to really useful and entertaining stories; at other times the question was met with little interest.

The obvious avenue for further work is a similar treatment of axiomatic semantics. This was developed largely by Hoare towards the end of the 1960s; the key reference is Hoare (1969). The approach uses axioms about program constructs in the form of pre and post conditions, and rules of inference for joining them, taking cues from axioms of mathematics. Once a language has sufficient rules and axioms to enable the proof of any true property and the proof of no incorrect properties, the language can be regarded as fully and formally specified; it is for this reason that the axiomatic method can be seen as one for formal semantics. It is often regarded as the easiest approach for a programmer to use due to its clearer link with the operations of programs (Pagan 1981, § 4.3). An extension to the axiomatic approach was provided by Dijkstra (1975) who used 'predicate transformers' to form the weakest pre condition that still satisfies all the properties of the statement. Hoare's later algebraic approach adapted the axiomatic work into algebras which use defined properties of programming languages and rules about the manipulation of them to determine

properties of programs.

A good overview of further approaches is given by Mosses (2001) in his paper 'The varieties of programming language semantics and their uses'. Some further developments in the field are Kahn's Natural Semantics: an extension to SOS that uses larger steps for more intuitive meaning (Kahn 1987); monadic semantics from Moggi (1991) which uses a category-theoretical approach; and action semantics which uses the concept of actions as the core of programming languages and models them in a hybrid operational/denotational approach (Mosses 1992).

The reason for the present account being incomplete in this regard is simply a matter of size. It was decided that a more in-depth look at two main semantics threads (Vienna and Oxford) was preferable to a broader but less detailed study. It is hoped that the present material, together with a full treatment of axiomatic semantics and perhaps some other expansions, can be worked into a book.

A possible direction for more historical study of formal and theoretical computing would be to look a lot more at program verification work, and to consider the period in which many people who had worked on defining whole programming languages narrowed down their focus to individual programs. This would involve looking at VDM, axiomatic semantics, pre- and post-conditions, schemas (as in Z), and graphical and other non-formal methods of modelling. Syntax and parsing is largely ignored in the present work, but is a very important research area and has been for a long time. A parser based on a concrete syntax definition is a crucial part of a compiler's front end, and this has led to syntax descriptions being probably the most successful part of language description generally. Finally, handling non-determinism and concurrency was a major challenge for formal semantics, and this led to many diverse alternative approaches such as process algebras, Petri nets, programming tools like semaphores, and hardware description languages. These methods are so different that at first glance it is difficult to tell they are addressing the same kinds of challenges. This is a rich vein for exploring historically, with many obvious practical impacts in industry, such as semi-conductor design.

8.4 Final remarks

The introduction of high level programming languages brought great challenges to computing, and getting to grips with these complex objects required a lot of effort as well as new ideas not previously seen in mathematics and logic. The costs (in

terms of time and money) associated with getting languages and their compilers *right* were huge, and addressing this using formalism was an important activity in the 1960s and through into the 1980s. Formal syntax was a clear success story with great applicability in parsers and compilers but semantics was a much bigger challenge. While it is not the case that, as the early semanticists had hoped, every programming language has a full formal definition, work on semantics has impacted in a number of important technical areas.[11]

Looking more specifically at the two core stories in the present work, some comparisons may be drawn between the work of the Vienna Lab and the Oxford Programming Research Group. Both groups wanted their methods to be generally applicable tools; both aimed for theoretical and practical contributions as well as formal proofs; and both resulted in large and complex models when applied to full-scale languages such as PL/I and Sal. The ALGOL 60 definitions written by the two groups are arguably similarly complex: the VDL work is long, has many parts, and has to cope with manipulation of the grand state; the denotational description has tricky mathematics and dense notation. (The later VDM definition lies somewhere in the middle of the two, and took many of the best parts from each earlier approach).

However, the approaches devised by the two groups were very different. Vienna was always more compiler-oriented with proofs coming as a secondary outcome: this was a natural result of their greater experience as practical programmers, and their requirement to produce industrial products. Neither the description of PL/I (not a language designed for the exhibition of interesting ideas) in the first phase nor the building of a compiler in the second phase were deeply theoretical goals. Despite this, some of the group being more theoretically-minded did result in publications on proofs and the modelling technique. The Programming Research Group came from a much more theoretical perspective, although Strachey was certainly not a classic 'ivory tower' academic due to his practical experience of programming—but this group was more able to choose its own challenges. Strachey's claims of practical grounding are easy to lose amongst the complex mathematics in denotational semantics, but domain theory especially remains as a deep and important contribution to theoretical computing. VDL, by constrast, has had very little direct influence on computing more broadly—although reaction to some of the challenges of using that approach was influential in the creation of VDM.

[11] Experience of modelling programming languages led into formal modelling techniques being applied in different domains, such as biological systems; see, for example, the work of Rozenberg, Bck, and Kok (2011) and Steggles et al. (2006).

Arguably VDM synthesised some of the best parts of both VDL and the Oxford denotational style. It was, however, still too 'operational' for the denotational purists yet not sufficiently simple for industrial use. Despite this, VDM led into useful and continuing program verification work where denotational semantics now sees less common use. SOS is another example of a synthesis of styles and continues to be used, especially for teaching.

The work now concluding tells an interesting story containing a lot of interconnection and interactions, but plenty of resistance to change. Despite some philosophical objections, formal semantics can be seen as really starting off the idea of formalised theoretical computing—at least in the European sense. This indicates that theoretical computing is a rich and fertile area for further historical research.

APPENDIX A

List of acronyms

ACM Association for Computing Machinery.

AE Applicative Expression, a term used by Landin.

ALGOL Algorithmic Language (typically followed by 58, 60, or 68).

ANSI American National Standards Institution. Notable for producing a standard definition of PL/I that used some ideas of formalism taken from VDL.

BCS British Computer Society.

BNF Backus Normal Form/Backus-Naur Form.

CACM *Communications of the ACM*, a regular publication from the ACM containing letters, discussions, algorithms, and short-form papers.

ECMA European Computer Manufacturers' Association.

EE English Electric.

FLDL *Formal Language Description Languages*, an IFIP Working Conference held in September 1964 in Baden-bei-Wien.

GAMM *Gesellschaft für Angewandte Mathematik und Mechanik* (German) Association for Applied Mathematics and Mechanics.

I/O input/output.

IAE Imperative Application Expression (see AE).

IBM International Business Machines.

IFIP International Federation for Information Processing.

ICIP International Congress in Information Processing, the large conference that led to the creation of IFIP.

ISO International Organisation for Standardisation.

MC Mathematisch Centrum, a research institute in computing and mathematics located in Amsterdam, Netherlands.

MFPS *Mathematical Foundations of Programming Semantics*, a conference series. Particularly relevant is the 20th meeting, held on 22nd May 2004, at Carnegie Mellon University, at which a panel comprising Cliff Jones, John McCarthy, John Reynolds, and Dana Scott discussed some reminiscences.

NPL National Physical Laboratory; a science research facility in Teddington, UK.

NPL New Programming Language; IBM's proposal in the mid-1960s that later became published under the name 'PL/I'.

NRDC National Research Development Corporation, a semi-autonomous research organisation in the UK.

PL/I Programming Language One; IBM's ambitious programming language initially designed for its System/360 range which aimed to combine the functionality of FORTRAN, COBOL, and ALGOL.

PLAP *Programming Languages and Pragmatics*, a conference held in California on 8–12 August 1965.

PRG Programming Research Group, Oxford University: set up by Christopher Strachey primarily to investigate non-numerical theory and applications of programming.

SDC System Development Corporation.

SECD The SECD machine is an abstract machine consisting of four parts: a stack, an environment, a control, and a dump. It is an invention of Peter Landin and inspiration for the Vienna group.

SHARE IBM's user group (not an acronym).

SRL Systems Reference Library; a document used as the official standard for the PL/I programming language.

TUW *Technische Universität Wien* (German) Technical University of Vienna.

ULD Universal Language Document; refers, with numerical suffixes, to various descriptions of PL/I, particularly those associated with the IBM Laboratory Vienna's formal definitions.

WCMLS *Working Conference on Mechanical Language Structures*, held in Princeton in 1963. Proceedings published in *CACM*, Volume 7, Number 2 (February 1964).

WG Working Group. Commonly used for IFIP working groups, particularly 2.1 and 2.2, concerning ALGOL and formal language description respectively.

VDL Vienna Definition Language; the ultimately-accepted term used for the no-

tation of ULD (above).

VDM Vienna Development Method, an approach for describing programming language semantics and program development.

APPENDIX B

List of interviews

Note: for brevity, the following acronyms are used:

TKA: Troy Astarte

CBJ: Cliff Jones

NCL: Newcastle University

Interviewee	Interviewer	Date	Duration	Location
Brian Randell	TKA	2016-08-02	01:34:09	Claremont Tower, NCL
David Beech[1]	TKA, CBJ	2016-08-12	02:09:55	Claremont Tower, NCL
Joe Stoy	TKA	2016-11-17	01:28:27	Balliol College, Oxford
Erich Neuhold	TKA, CBJ	2016-12-16	01:15:27	Café Eiles, Vienna
Rod Burstall	TKA, CBJ	2017-01-12	01:26:57	Burstall's home, Edinburgh
Rod Burstall	TKA	2017-01-12	01:10:58	Burstall's home, Edinburgh
Rod Burstall	TKA, CBJ	2017-07-04	03:12:32	Claremont Tower, NCL
J Moore	TKA	2017-11-07	01:22:07	Urban Sciences Building, NCL
Gordon Plotkin	TKA, CBJ	2018-04-23	00:52:46	Skype to Plotkin's home
Doug McIlroy	CBJ	2018-05-07	01:03:17	Hilton Hotel, Providence

[1] Brian Randell was also present for part of this interview.

Index

Page numbers **in bold** indicate the beginning of the main section where the entry is discussed, where appropriate. Numbers *in italics* indicate locations where photographs may be found.

References

ACM News (1977). "ACM News". In: *Communications of the ACM* 20.9 (Sept. 1977), pp. 681–682.

Alber, K. (1967). *Syntactical Description of PL/I Test and Its Translation into Abstract Normal Form*. Technical Report TR 25.074. IBM Laboratory Vienna.

Alber, K. and Oliva, P. (1968). *Translation of PL/I into Abstract Syntax*. Technical Report 25.086. IBM Laboratory Vienna, ULD III Version II.

Alber, K., Oliva, P., and Urscler, G. (1968). *Concrete Syntax of PL/I*. Tech. rep. 25.084. IBM Laboratory Vienna, ULD Version II.

Alber, K. et al. (1969). *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*. Technical Report TR 25.099. ULD Version III. IBM Laboratory Vienna.

Alberts, Gerard, ed. (2014a). *Algol Culture and Programming Styles*. Vol. 36. Annals of the History of Computing 4. IEEE, 2014.

Alberts, Gerard (2014b). "Algol Culture and Programming Styles [Guest editor's introduction]". In: *IEEE Annals of the History of Computing* 36.4 (2014), pp. 2–5. DOI: 10.1109/MAHC.2014.49.

Alberts, Gerard (2016). "International Informatics - Aad van Wijngaarden's 100th Birthday". In: *ERCIM News* 2016.106, pp. 6–7.

Alberts, Gerard and Daylight, Edgar G. (2014). "Universality versus Locality: The Amsterdam Style of ALGOL Implementation". In: *IEEE Annals of the History of Computing* 36.4, pp. 52–63. DOI: 10.1109/MAHC.2014.61.

Allen, C. D. (1967). *ULD3 interpreter*. IBM internal memo to J. E. Nicholls. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Allen, C. D., Chapman, D. N., and Jones, C. B. (1972). *A Formal Definition of ALGOL 60*. Tech. rep. 12.105. IBM Laboratory Hursley. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR12.105.pdf.

Allen, C. D. et al. (1966). *An Abstract Interpreter of PL/I*. Technical Note TN 3004. IBM Hursley Laboratories.

Ambler, A Patricia et al. (1973). "A Versatile Computer-Controlled Assembly System." In: *IJCAI*, pp. 298–307.

Andrews, Derek and Henhapl, Wolfgang (1982). "Pascal". In: *Formal Specification and Software Development*. Ed. by Dines Bjørner and Cliff B. Jones. Prentice Hall International. Chap. 6, pp. 175–252. ISBN: 0-13-329003-4. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982/Chapter-7.pdf.

Andrews, D.J. et al. (1988). "The formal definition of Modula-2 and its associated interpreter". In: *VDM '88 VDM — The Way Ahead*. Ed. by Robin E. Bloomfield, Lynn S. Marshall, and Roger B. Jones. Vol. 328. Lecture Notes in Computer Science. Springer, pp. 167–177. ISBN: 978-3-540-50214-2. DOI: 10.1007/3-540-50214-9_15. URL: http://dx.doi.org/10.1007/3-540-50214-9_15.

ANSI (1976). *Programming Language PL/I*. Tech. rep. X3.53-1976. American National Standard.

Arbab, Bijan and Berry, Daniel M (1987). "Operational and denotational semantics of Prolog". In: *The Journal of Logic Programming* 4.4, pp. 309–329.

Astarte, Troy K. and Jones, Cliff B. (2018). "Formal Semantics of ALGOL 60: Four Descriptions in their Historical Context". In: *Reflections on Programming Systems - Historical and Philosophical Aspects*. Ed. by Liesbeth De Mol and Giuseppe Primiero. Springer Philosophical Studies Series, pp. 71–141.

Auerbach, Isaac L. (1986a). "IFIP—The Early Years: 1960–1971". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. 1986, pp. 71–94.

Auerbach, Isaac L. (1986b). "Personal recollections on the origin of IFIP". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. 1986, pp. 41–71.

Backus, J. W. et al. (1963a). "Revised Report on the Algorithm Language ALGOL 60". In: *Communications of the ACM* 6.1 (Jan. 1963). Ed. by P. Naur, pp. 1–17. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/BBG63.pdf.

Backus, John (1978). "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641. URL: http://doi.acm.org/10.1145/359576.359579.

Backus, John (1981). "The history of FORTRAN I, II, and III". In: *History of Programming Languages*. Ed. by Richard L. Wexelblat, pp. 25–45.

Backus, John et al. (1956). *FORTRAN: Programmer's reference manual*. Tech. rep. Applied Science Division and Programming Research Dept., IBM.

Backus, John W. (1954a). *Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN*. Tech. rep. Applied Science Division, IBM, Nov. 1954.

Backus, John W. (1954b). "The IBM 701 speedcoding system". In: *Journal of the ACM* 1.1 (1954), pp. 4–6.

Backus, John W. (1980). "Programming in America in the 1950s—Some Personal Impressions". In: *A History of Computing in the Twentieth Century*. Elsevier, pp. 125–135.

Backus, John W et al. (1957). "The FORTRAN automatic coding system". In: *Papers presented at the February 26-28, 1957 Western Joint Computer Conference: Techniques for reliability*. ACM, pp. 188–198.

Backus, John W. et al. (1960). "Report on the algorithmic language ALGOL 60". In: *Numerische Mathematik* 2.1, pp. 106–136.

Backus, John W. et al. (1963b). "Revised Report on the Algorithmic Language ALGOL 60". In: *The Computer Journal* 5.4 (1963), pp. 349–367.

Backus, John Warner (1959). "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference". In: *Proceedings of the International Conference on Information Processing*. UNESCO, pp. 125–132.

de Bakker, J. W. (1965). *Formal Definition of Algorithmic Languages, with an application to the definition of ALGOL 60*. Tech. rep. MR–74. Stichting Mathematisch Centrum.

de Bakker, J. W. and Scott, D. (1969). "A Theory of Programs". Manuscript notes for IBM Seminar, Vienna.

de Bakker, Jacobus Willem (1967). *Formal definition of programming languages. With an application to the definition of ALGOL 60*. Mathematical Centre tracts: 16. Amsterdam: Mathematisch Centrum.

Bandat, K. (1967). *On the Formal Definition of PL/I*. Technical Report TR 25.073. IBM Laboratory Vienna.

Bandat, K (1968). "On the formal definition of PL/I". In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. ACM, pp. 363–373.

Bandat, Kurt (1965). *Tentative Steps Towards a Formal Definition of Semantics of PL/I*. Tech. rep. TR 25.056. IBM Laboratory, Vienna.

Bandat, Kurt et al. (1964). *Review of the Report of the SHARE Advanced Language Development Committee*. Tech. rep. IBM Vienna Science Group.

Bandat, Kurt et al. (1965). *Unambiguous definition of PL/I*. IBM internal memo. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Barendregt, Hendrik Pieter (1971). "Some extensional term models for combinatory logics and lambda-calculi". PhD thesis. Utrecht: Rijksuniversiteit Utrecht.

Barron, D. W. et al. (1963). "The Main Features of CPL". In: *Computer Journal* 6, pp. 134–143.

Barron, David W. (1975). "Christopher Strachey: a personal reminiscence". In: *Computer Bulletin*, pp. 8–9.

Bauer, Friedrich L. (1972). "From Scientific Computation to Computer Science". In: *The Skyline of Information Processing: Proceedings of the tenth anniversary celebration of the IFIP*. Ed. by Heinz Zemanek. IFIP. North-Holland.

Beech, D., Nicholls, J. E., and Rowe, R. (1966). *A PL/I Translator*. Technical Note TN 3003. IBM Hursley Laboratories, PL/I Dept.

Beech, D. et al. (1966). *Concrete Syntax of PL/I*. Technical Note TN 30001. Note: "This replaces and supersedes the issue dated 1st June 1966". IBM United Kingdom Laboratories Limited, PL/I Dept.

Beech, D. et al. (1967). *Abstract Syntax of PL/I*. Technical Note TN 3002 (Version 2). IBM United Kingdom Laboratories Limited, PL/I Language Dept.

Beech, David (1970). "A Structural View of PL/I". In: *ACM Computing Surveys* 2.1, pp. 33–64. URL: http://doi.acm.org/10.1145/356561.356564.

Beech, David (2016). *Interview with David Beech*. Conducted by Troy Astarte and Cliff Jones. Also present: Brian Randell.

Beech, David and Marcotty, Michael (1973). "Unfurling the PL/I Standard". In: *SIGPLAN Notices* 8.10 (Oct. 1973), pp. 12–43. DOI: 10.1145/987012.987014.

Bekič, H. (1965a). *Recursion*. Tech. rep. LDV 1. IBM Laboratory Vienna, 1965.

Bekič, H. and Walk, K. (1971). "Formalization of Storage Properties". In: *(Engeler 1971)*. Ed. by E. Engeler. Springer, pp. 28–61.

Bekič, Hans (1964). *Defining a Language in its own terms*. Tech. rep. 25.3.016. IBM Laboratory Vienna.

Bekič, Hans (1965b). *Mechanical transformation rules for the reduction of ALGOL to a primitive language M and their use in defining the compiler function*. Tech. rep. TR 25.051. IBM Laboratory Vienna, 1965.

Bekič, Hans (1965c). *Mr. Pfeiffers discussion on ALGOL and PL/I recursion*. IBM internal memo. 1965.

Bekič, Hans (1965d). *Recursion*. Memo to Language Resolution Board. 1965.

Bekič, Hans (1968a). *Letter to A. van Wijngaarden*. 1968.

Bekič, Hans (1968b). *Response to question about priorities of Working Group*. Online. From 10th Meeting of IFIP WG2.1. 1968. URL: http://ershov.iis.nsk.su/en/node/806196.

Bekič, Hans (1984a). "Definable operations in general algebras, and the theory of automata and flowcharts". In: *Programming Languages and Their Definition: Hans Bekič (1936-1982). Selected papers*. Ed. by Cliff B. Jones. Vol. 177. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1984, pp. 30–55. ISBN: 978-3-540-38933-0. URL: http://dx.doi.org/10.1007/BFb0048939.

Bekič, Hans (1984b). *Programming Languages and Their Definition: Hans Bekič (1936-1982). Selected papers*. Ed. by Cliff B. Jones. Vol. 177. Lecture Notes in Computer Science. Springer-Verlag, 1984.

Bekič, Hans and Jones, C. B., eds. (1984). *Programming Languages and Their Definition: Selected Papers of H. Bekič*. Vol. 177. LNCS. Springer-Verlag. ISBN: 978-3-540-13378-0. DOI: 10.1007/BFb0048933. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/LNCS177-Bekic/.

Bekič, Hans et al. (1974). *A Formal Definition of a PL/I Subset*. Tech. rep. 25.139. IBM Laboratory Vienna. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/TR25139/.

Bekič, Hans et al. (1975). *Some experiments with using a formal language definition in compiler development*. Laboratory note LN 25.3.107. IBM Laboratory Vienna.

Belgraver Thissen, W. P. C. et al. (2007a). *Adriaan van Wijngaarden*. 2007. URL: http://www-set.win.tue.nl/UnsungHeroes/heroes/vwijngaarden.html.

Belgraver Thissen, W. P. C. et al. (2007b). *Computing Girls*. 2007. URL: http://www-set.win.tue.nl/UnsungHeroes/heroes/computing-girls.html.

Berger, Martin and Milner, Robin (2003). *An interview with Robin Milner*.

Bergin, Thomas J. and Gibson, Richard G., eds. (1996). *History of programming languages—II*. New York, NY, USA: ACM Press. ISBN: 0-201-89502-1.

Berry, Daniel M (1985). "A denotational semantics for shared-memory parallelism and nondeterminism". In: *Acta informatica* 21.6, pp. 599–627.

Beyer, Kurt W. (2009). *Grace Hopper and the Invention of the Information Age*. The MIT Press. ISBN: 026201310X, 9780262013109.

Bjørner, D., ed. (1980). *Abstract Software Specifications: 1979 Copenhagen Winter School Proceedings*. Vol. 86. Lecture Notes in Computer Science. Berlin: Springer-Verlag.

Bjørner, D. and Jones, C. B., eds. (1978). *The Vienna Development Method: The Meta-Language*. Vol. 61. LNCS. Springer-Verlag. ISBN: 978-3-540-08766-3. DOI: 10.1007/3-540-08766-4. URL: http://dx.doi.org/10.1007/3-540-08766-4.

Bjørner, Dines (2007). *Dines Bjørner: Formal CV*. Online. URL: http://www.imm.dtu.dk/~dibj/biography/biography.html.

Bjørner, Dines (2015). "The Wiener Informatik Kreis – on an Origin of OCG. IBM Wiener Labor: May 1973–August 1975". In: *In memoriam Heinz Zemanek*. Ed. by Johann Stockinger Karl Anton Froeschl Gerhard Chroust. Vol. Band-311. OCG. ISBN: 978-3-903035-00-3.

Bjørner, Dines and Havelund, Klaus (2014). "40 years of formal methods". In: *International Symposium on Formal Methods*. Springer, pp. 42–61.

Bjørner, Dines and Jones, Cliff B., eds. (1982). *Formal Specification and Software Development*. Prentice Hall International. ISBN: 0-13-329003-4. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982.

Bjørner, Dines and Oest, Ole Nybye, eds. (1980). *Towards a formal description of Ada*. LNCS 98. Springer-Verlag.

Bjørner, Dines et al., eds. (1987). *VDM – A Formal Definition at Work*. Vol. 252. LNCS. Springer-Verlag. ISBN: 978-3-540-17654-1. DOI: 10.1007/3-540-17654-3. URL: http://www.springer.com/computer/swe/book/978-3-540-17654-1.

Bloom, Stephen L. (1982). "Preface". In: *Selected Papers: Calvin C. Elgot*. Springer-Verlag.

Blum, Edward K. (1975). *Minutes of the 7th meeting of IFIP WG 2.2 on Formal Description of Programming Concepts*. Held in Rigi-Kaltbad, Switzerland. Chaired by E. Neuhold.

Böhm, C. (1966). "The CUCH as a formal and description language". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 179–197.

Bonsall, G. W. et al. (1967). *ULD3 and Language Development*. IBM internal memo to J. L. Cox. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Bornat, Richard (2009a). "Peter Landin: a computer scientist who inspired a generation, 5th June 1930 - 3rd June 2009". In: *Formal Aspects of Computing* 21.5 (2009), pp. 393–395. ISSN: 1433-299X. URL: https://doi.org/10.1007/s00165-009-0122-y.

Bornat, Richard (2009b). *Peter Landin obituary*. The Guardian. 2009. URL: https://www.theguardian.com/technology/2009/sep/22/peter-landin-obituary.

Bowlden, H. J. (1973). *Minutes of the 18th meeting of IFIP WG 2.1.*

Brooker, Ralph Anthony (1958). "The autocode programs developed for the Manchester University computers". In: *The Computer Journal* 1.1, pp. 15–21.

Brooks, Fred and Shustek, Len (2015). "An Interview with Fred Brooks". In: *Communications of the ACM* 58.11, pp. 36–40.

Buchwald, Art (1959). "Of Machines and Men". In: *New York Herald Tribune*. Press clipping.

Burstall, R. M. (1969a). "Proving Properties of Programs by Structural Induction". In: *Computer Journal* 12 (1969). Earlier available as Experimental Programming Report, No. 17, DMIP, Edinburgh, 1968, pp. 41–48.

Burstall, R. M. and Landin, P. J. (1969). "Programs and their Proofs: an Algebraic Approach". In: *Machine Intelligence 4*. Ed. by B. Meltzer and D. Michie. Edinburgh University Press, pp. 17–43.

Burstall, Rod (2000). "Christopher Strachey—understanding programming languages". In: *Higher-Order and Symbolic Computation* 13.1, pp. 51–55.

Burstall, Rod M. (1966). "Semantics of assignment". In: *Machine Intelligence* 2, pp. 3–20.

Burstall, Rod M. (1969b). "Formal description of program structure and semantics in first-order logic". In: *Machine Intelligence* 5 (1969), pp. 79–98.

Burstall, Rod M. (1993). *Interviewed by Tony Dale*. Edinburgh, UK.

Burstall, Rod M. (2017). *Interview with Rod Burstall*. Conducted by Troy Astarte and Cliff Jones.

Burstall, Rod M. and Goguen, Joseph A. (1980). "The semantics of Clear, a specification language". In: *Abstract Software Specifications*. Lecture Notes in Computer Science 86. Springer, pp. 292–332.

Burstall, Rod M and Popplestone, Robin J (1968). "POP-2 reference manual". In: *Machine Intelligence* 2.205-249, p. 1.

Buxton, J. N. et al., eds. (1966). *CPL Working Papers*. Technical Report.

Campbell-Kelly, Martin (1980a). "Programming the EDSAC: Early programming activity at the University of Cambridge". In: *Annals of the History of Computing* 2.1 (1980), pp. 7–36.

Campbell-Kelly, Martin (1980b). "Programming the Mark I: Early programming activity at the university of Manchester". In: *Annals of the History of Computing* 2.2 (1980), pp. 130–168.

Campbell-Kelly, Martin (1981). “Programming the Pilot ACE: Early Programming Activity at the National Physics Laboratory”. In: *Annals of the History of Computing* 3.2, pp. 133–162.

Campbell-Kelly, Martin (1985). “Christopher Strachey, 1916-1975: A Biographical Note”. In: *IEEE Annals of the History of Computing* 1.7, pp. 19–42.

Campbell-Kelly, Martin (2003). *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge University Press.

Campbell-Kelly, Martin (2007). “The history of the history of software”. In: *IEEE Annals of the History of Computing* 29.4.

Campbell-Kelly, Martin and Aspray, William (1996). *Computer: a history of the information machine*. The Sloan technology series. New York, N.Y: Basic Books. ISBN: 0465029892.

Caracciolo di Forino, A. (1966). “On the concept of formal linguistic systems”. In: *Formal Language Description Languages for Computer Programming*. Ed. by T. B. Steel. North-Holland, pp. 37–51.

Cardone, Felice and Hindley, J Roger (2006). “History of lambda-calculus and combinatory logic”. In: *Handbook of the History of Logic* 5, pp. 723–817.

de Chadarevian, Soraya (2002). *Designs for life: Molecular biology after World War II*. Cambridge University Press.

Cox, Jim L. (1967). *Further Vienna PL/I activity*. IBM internal memo to H. Zemanek. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Čulík, Karel (1966). “Well-translatable grammars and ALGOL-like languages”. In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 76–85.

Danvy, Olivier (2009a). *In Memoriam Peter Landin*. Talk given at International Conference on Functional Programming, Edinburgh. 2009. URL: `https://vimeo.com/6638882`.

Danvy, Olivier (2009b). “Peter J. Landin (1930–2009)”. In: *Higher-Order and Symbolic Computation* 22.2 (2009), pp. 191–195.

Danvy, Olivier and Millikin, Kevin (2008). “A Rational Deconstruction of Landin’s SECD Machine with the J Operator”. In: *Logical Methods in Computer Science* 4.12, pp. 1–67.

Daylight, Edgar G. (2012). *The Dawn of Software Engineering: From Turing to Dijkstra*. Ed. by Kurt De Grave. Belgium: Lonely Scholar. ISBN: 9491386026, 9789491386022.

Daylight, Edgar G. (2018). *Towards a History of Model–Modellee Conflations in Computer Science*. Launch event "What is a (Computer) Program?", Lille, France.

DeMillo, Richard A, Lipton, Richard J, and Perlis, Alan J (1979). "Social processes and proofs of theorems and programs". In: *Communications of the ACM* 22.5, pp. 271–280.

Dijkstra, E. W. (1968). "Go to Statement Considered Harmful". In: 11.3, pp. 147–148.

Dijkstra, E. W. (1972). "The Humble Programmer". In: 15.10, pp. 859–866.

Dijkstra, E. W. (1975). "Guarded commands, nondeterminacy and formal derivation of programs". In: 18, pp. 453–457.

Dijkstra, Edsger W. (1961a). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 289, F.15. 1961.

Dijkstra, Edsger W. (1974a). *Trip report E. W. Dijkstra, Edinburgh and Newcastle, 1–6 September 1974*. Held in the Dijkstra archive online. 1974. URL: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD448.html`.

Dijkstra, Edsger W (2001a). "Under the spell of Leibniz's dream". In: *Information processing letters* 77.2-4 (2001), pp. 53–61.

Dijkstra, Edsger W. et al. (1969). *Minority Report*. Held in the Ershov archive (online). URL: `http://ershov.iis.nsk.su/en/node/805785`.

Dijkstra, Edsger Wybe (1961b). *On the Design of Machine Independent Programming Languages*. Report MR 34. Mathematisch Centrum, 1961. URL: `http://www.cs.utexas.edu/users/EWD/transcriptions/MCreps/MR34.html`.

Dijkstra, Edsger Wybe (1962). *An attempt to unify the constituent concepts of serial program execution*. Tech. rep. MR 46. Paper to be presented at the Symposium on Symbolic Languages in Data Processing. Rome, March 1962. Mathematisch Centrum.

Dijkstra, Edsger Wybe (1970a). *Letter to Tony Hoare*. Held in the Dijkstra archive online. EWD 292. 1970. URL: `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD292.html`.

Dijkstra, Edsger Wybe (1970b). *Notes on structured programming*. Hard copy. 1970.

Dijkstra, Edsger Wybe (1974b). *Letter to Hans Bekič*. Held in the Dijkstra archive online. EWD 454. 1974. URL: `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD454.html`.

Dijkstra, Edsger Wybe (1980). "A programmer's early memories". In: *A History of Computing in the Twentieth Century: a Collection of Essays*. Ed. by N. Metropolis, Howlett J., and Gian-Carlo Rota. Academic Press, pp. 563–573.

Dijkstra, Edsger Wybe (2001b). *What led to "Notes on Structured Programming"*. Held in the Dijkstra archive online. EWD 1308. 2001. URL: http://www.cs.utexas.edu/users/EWD/transcriptions/EWD13xx/EWD1308.html.

Dixit, Uday Shanker, Hazarika, Manjuri, and Davim, J Paulo (2017). *A brief history of mechanical engineering*. Materials Forming, Machining and Tribology. Springer.

Dobrusky, W. B. and Steel, T. B. (1961). "Universal Computer-oriented Language". In: *cacm* 4.3 (Mar. 1961). URL: http://doi.acm.org/10.1145/366199.366220.

Donahue, J. E. (1976). *Complementary Definitions of Programming Language Semantics*. Vol. 42. LNCS. Springer-Verlag.

Duby, J. J. (1973). *Minutes of the policy subcommittee of Working Group 2.2*. Held in the Ershov archive (online). URL: http://ershov.iis.nsk.su/en/node/806092.

Duncan, Fraser G. (1966). "Our ultimate metalanguage: an afterdinner talk". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 295–295.

Duncan, Fraser G. (1969). *Resignation from WG 2.1*.

Durnova, Helena and Alberts, Gerard (2014). "Was Algol 60 the First Algorithmic Language?" In: *IEEE Annals of the History of Computing* 36.4, p. 104. DOI: 10.1109/MAHC.2014.63.

Eden, Amnon H. (2007). "Three Paradigms of Computer Science". In: *Minds and Machines* 17.2, pp. 135–167. ISSN: 1572-8641. DOI: 10.1007/s11023-007-9060-8. URL: https://doi.org/10.1007/s11023-007-9060-8.

Eickel, Jürgen and Paul, Manfred (1966). "The parsing and ambiguity problem for Chomsky-languages". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 52–75.

Elgot, C. C. (1966a). *A Notion of Interpretability of Algorithms in Algorithms*. Technical Report TR 25.068. IBM Laboratory Vienna, 1966.

Elgot, C. C. and Robinson, A. (1964). "Random Access Stored-Program Machines: An Approach to Programming Languages". In: *Journal of the ACM* 11, pp. 365–399.

Elgot, Calvin C. (1966b). "Machine species and their computation languages". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 160–178.

Endres, Albert (2013). "Early language and compiler developments at IBM Europe: A personal retrospection". In: *Annals of the History of Computing, IEEE* 35.4, pp. 18–30.

Engeler, E. (1971). *Symposium on Semantics of Algorithmic Languages*. Lecture Notes in Mathematics 188. Springer-Verlag.

Feeney, J. M. (1981). "Management Information Systems - The Failure of Technology". In: *Business Information Systems*. 9 7. Pergamon/Infotech.

Fetzer, James H (1988). "Program verification: the very idea". In: *Communications of the ACM* 31.9, pp. 1048–1063.

Fleck, M. (1969). *Formal Definition of the PL/I Compile Time Facilities*. Technical Report TR 25.095. ULD Version III. IBM Laboratory Vienna.

Fleck, M. and Neuhold, E. (1968). *Formal Definition of the PL/I Compile Time Facilities*. Technical Report TR 25.080. ULD Version II. IBM Laboratory Vienna.

Floyd, R. W. (1962). "On the Nonexistence of a Phrase Structure Grammar for ALGOL 60". In: *Communications of the ACM* 5, pp. 483–484.

Floyd, R. W. (1967a). "Assigning Meanings to Programs". In: *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*. American Mathematical Society, 1967, pp. 19–32.

Floyd, Robert W (1967b). "The verifying compiler". In: *Computer Science Research Review Carnegie-Mellon University Annual Report* 967 (1967), pp. 18–19.

Forsythe, George E. (1966). "Welcoming Remarks". In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 137–138. URL: http://doi.acm.org/10.1145/365230.365247.

Fox, Leslie (1961). "Computing machines for teaching and research". In: *The Computer Journal* 4.3, pp. 212–216.

Fox, Leslie, ed. (1966). *Advances in Programming and Non-Numerical Computation*. Pergamon.

Gaboury, Jacob (2013). *A Queer History of Computing: Part Four*. URL: http://rhizome.org/editorial/2013/may/6/queer-history-computing-part-four/.

Garwick, Jan V. (1966). "The definition of programming languages by their compilers". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 139–147.

Garwick, Jan V. (1969). *Resignation from WG 2.1*.

Gilmore, P. C. (1963). "An Abstract Computer with a Lisp-Like Machine Language Without a Label Operator". In: *Computer Programming and Formal Systems*. Ed. by P. Braffort and D. Hirschberg. Vol. 35. Studies in Logic and the Foundations of Mathematics. Elsevier, pp. 71 –86. DOI: https://doi.org/10.1016/S0049-

237X(08)72019-6. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08720196.

Ginsburg, Seymour (1966). "A survey of ALGOL-like and context-free language theory". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 68–99.

Glennie, Alick E. (1952). *The automatic coding of an electronic computer*. Talk delivered at Cambridge University, February 1953.

Goldstein, Gordon D. (1958). *Digital Computer Newsletter*. Tech. rep. Vol. 10, No. 1. Office of Naval Research, Mathematical Sciences Division.

Goldstine, Herman H. and von Neumann, John (1947). *Planning and Coding of Problems for an Electronic Computing Instrument*. Tech. rep. Institute of Advanced Studies, Princeton.

Gordon, M. (1975). *Operational Reasoning and Denotational Semantics*. Tech. rep. STAN-CS-75-506. Stanford University, Computer Science Department.

Gordon, Michael J. C. (1973). "Evaluation and denotation of pure LISP programs: a worked example in semantics". PhD thesis. The University of Edinburgh.

Gordon, Mike (2000). "Christopher Strachey: recollections of his influence". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 65–67.

Gorn, Saul (1961a). "Some Basic Terminology Connected with Mechanical Languages and Their Processors: A Tentative Base Terminology Presented to ASA x3.4 As a Proposal for Subsequent Inclusion in a Glossary". In: *cacm* 4.8 (Aug. 1961), pp. 336–339. URL: http://doi.acm.org/10.1145/366678.366682.

Gorn, Saul (1961b). "Specification Languages for Mechanical Languages and Their Processors a Baker's Dozen: A Set of Examples Presented to ASA x3.4 Subcommittee". In: *Communications of the ACM* 4.12 (Dec. 1961), pp. 532–542. URL: http://doi.acm.org/10.1145/366853.366856.

Gorn, Saul (1964). "Summary Remarks (to a Working Conference on Mechanical Language Structures)". In: *Communications of the ACM* 7.2 (Feb. 1964), pp. 133–136. URL: http://doi.acm.org/10.1145/363921.363946.

Gorn, Saul (1966). "Language-naming languages in prefix form". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 249–265.

Grattan-Guinness, Ivor (2005). "History or heritage? An important distinction in mathematics and for mathematics education". In: *Mathematics and the Historian's Craft: the Kenneth O. May Lectures*. Ed. by Glen van Brummelen and Michael Kinyon. Springer. Chap. 1, pp. 7–21.

Haigh, Thomas (2004). "The history of computing: An introduction for the computer scientist". In: *Using history to teach computer science and related disciplines*. Computing Research Association Washington, DC, pp. 5–26.

Haigh, Thomas (2015). "The tears of Donald Knuth". In: *Communications of the ACM* 58.1, pp. 40–44.

Hamming, Richard W. (1980). "We would know what they thought when they did it". In: *A History of Computing in the Twentieth Century*. Academic Press.

Harder, Edwin L. (1986). "Financing IFIP". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. North Holland, pp. 335–339.

Harper, R., Milner, R., and Tofte, M. (1988). *The Definition of Standard ML Version 2*. Tech. rep. ECS-LFCS-88-62. Deptartment of Computer Science, University of Edinburgh.

Harper, Robert, Milner, Robin, and Tofte, Mads (1987). *The Semantics of Standard ML: Version 1*. Hard copy. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science.

Hartley, David (2000). "Cambridge and CPL in the 1960s". In: *Higher-Order and Symbolic Computation* 13.1, pp. 69–70.

He, Jifeng (2018). *Unifying theories of refinement*. Unified Theories of Programming 20th-anniversary BCS-FACS Evening Seminar. Introduction by Tony Hoare; summary by Jim Woodcock. Oct. 2018. URL: https://www.bcs.org/content/ConWebDoc/59563.

Hendry, John (1989). *Innovating for Failure*. MIT Press.

Henhapl, W. and Jones, C. B. (1970a). *On the Interpretation of GOTO Statements in the ULD*. Tech. rep. LN 25.3.065. IBM Laboratory, Vienna, Mar. 1970.

Henhapl, W. and Jones, C. B. (1970b). *The Block Concept and Some Possible Implementations, with Proofs of Equivalence*. Tech. rep. 25.104. IBM Laboratory Vienna, 1970.

Henhapl, Wolfgang and Jones, Cliff B. (1978). "A Formal Definition of ALGOL 60 as Described in the 1975 Modified Report". In: *The Vienna Development Method: The Meta-Language*. Ed. by D. Bjørner and Cliff B. Jones. Vol. 61. Lecture Notes in Computer Science. Springer-Verlag, pp. 305–336. DOI: 10.1007/3-540-08766-4_12. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/HJ82.pdf.

Henhapl, Wolfgang and Jones, Cliff B. (1982). "ALGOL 60". In: *Formal Specification and Software Development*. Ed. by Dines Bjørner and Cliff B. Jones. Prentice Hall International. Chap. 6, pp. 141–174. ISBN: 0-13-329003-4. URL: http://

homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/BjornerJones1982/Chapter-6.pdf.

Hicks, Marie (2017). *Programmed inequality: How Britain discarded women technologists and lost its edge in computing*. MIT Press.

Hoare, C. A. R. (1968a). *Recommendation on Draft Report on ALGOL 68*. 1968.

Hoare, C. A. R. (2000). "A Hard Act to Follow". In: *Higher-Order and Symbolic Computation* 13.1, pp. 71–72.

Hoare, C. A. R. and He, Jifeng (1998). *Unifying Theories of Programming*. Prentice Hall.

Hoare, C. A. R. and Lauer, P. E. (1974). "Consistent and Complementary Formal Theories of the Semantics of Programming Languages". In: *Acta Informatica* 3, pp. 135–153.

Hoare, Charles Anthony Richard (1968b). *Letter to A. van Wijngaarden*. 1968.

Hoare, Charles Anthony Richard (1994). *Interviewed by Tony Dale*. Oxford, UK.

Hoare, Charles Antony Richard (1969). "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10, pp. 576–580.

Hoare, Charles Antony Richard (1973). *Hints on Programming Language Design*. Tech. rep. Stanford, CA, USA: Stanford University.

Hoare, Charles Antony Richard (1985). "The mathematics of programming". In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, pp. 1–18.

Hopper, Grace Murray (1981). "Keynote address at ACM SIGPLAN History of Programming Languages conference, June C1–3 1978". In: *(Wexelblat 1981)*. Ed. by Richard L. Wexelblat. ACM Monograph Series.

van den Hove, Gauthier (2014). "On the Origin of Recursive Procedures". In: *The Computer Journal* 58.11, pp. 2892–2899.

IBM (1964). *The New Programming Language*. Tech. rep. IBM World Trade Laboratories (Great Britain).

IBM (2001). *IBM Highlights, 1885–1969*. In the IBM archive online. URL: https://www-03.ibm.com/ibm/history/documents/pdf/1885-1969.pdf.

IBM Informationsabteilung (1964). *Menschen sprechen mit Maschinen*. in IBM Presse Information.

IFIP (1964a). *Working Conference Vienna 1964 Formal Language Description Languages. Preliminary Program*. Held in the Ershov archive (online). 1964. URL: http://ershov.iis.nsk.su/en/node/805668.

IFIP (1964b). *Working Conference Vienna 1964 Formal Language Description Languages. Program*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39. 1964.

Ingerman, Peter Z. (1966). "The parameterization of the translation process". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 221–230.

Ingerman, Peter Zilahy (1969). *Resignation letter sent to WG 2.1*.

Izbicki, H. (1975). *On a Consistency Proof of a Chapter of a Formal Definition of a PL/I Subset*. Tech. rep. 25.142. IBM Laboratory Vienna.

Jackson, Michael (2000). "Christopher Strachey: A personal recollection". In: *Higher-Order and Symbolic Computation* 13.1, pp. 73–74.

Johnstone, Adrian, Scott, Elizabeth, and Economopoulos, Giorgios (2004). "Generalised parsing: Some costs". In: *International Conference on Compiler Construction*. Springer, pp. 89–103.

Jones, C. B. (1976). *Formal Definition in Compiler Development*. Tech. rep. 25.145. IBM Laboratory Vienna.

Jones, C. B. (2001). "The Transition from VDL to VDM". In: *Journal of Universal Computer Science* 7.8, pp. 631–640. DOI: 10.3217/jucs-007-08-0631. URL: http://www.jucs.org/jucs_7_8/the_transition_from_VDL.

Jones, C. B. and Lucas, P. (1970). *Proving Correctness of Implementation Techniques*. Tech. rep. TR 25.110. IBM Laboratory Vienna.

Jones, Cliff (2012). "John McCarthy". In: *Formal Aspects of Computing* 24, pp. 305–306. DOI: 10.1007/s00165-012-0226-7. URL: http://www.springerlink.com/content/c633860046q22243.

Jones, Cliff B. (1978). "Denotational Semantics of Goto: An Exit Formulation and its Relation to Continuations". In: *The Vienna Development Method: The Meta-Language*. Ed. by D. Bjørner and C. B. Jones. Vol. 61. Lecture Notes in Computer Science. Springer-Verlag, pp. 278–304. DOI: 10.1007/3-540-08766-4_11. URL: http://dx.doi.org/10.1007/3-540-08766-4_11.

Jones, Cliff B. (1999). "Scientific Decisions which Characterize VDM". In: *FM'99 – Formal Methods*. Vol. 1708. Lecture Notes in Computer Science. Springer-Verlag, pp. 28–47. DOI: 10.1007/3-540-48119-2_2. URL: http://www.springerlink.com/content/e8xfjt85eeulpt1u/.

Jones, Cliff B. (2003a). "Operational Semantics: concepts and their expression". In: *Information Processing Letters* 88.1-2 (2003), pp. 27–32. ISSN: 0020-0190. DOI:

10.1016/S0020-0190(03)00383-1. URL: http://dx.doi.org/10.1016/S0020-0190(03)00383-1.

Jones, Cliff B. (2003b). "The Early Search for Tractable Ways of Reasoning about Programs". In: *IEEE, Annals of the History of Computing* 25.2 (2003), pp. 26–49. DOI: 10.1109/MAHC.2003.1203057. URL: http://doi.ieeecomputersociety.org/10.1109/MAHC.2003.1203057.

Jones, Cliff B. (2015). "Thanks to Zem". In: *In memoriam Heinz Zemanek*. Ed. by Johann Stockinger Karl Anton Froeschl Gerhard Chroust. Vol. Band-311. OCG. ISBN: 978-3-903035-00-3.

Jones, Cliff B. (2017). "Turing's 1949 paper in context". In: *Computability in Europe 2017*. Ed. by Jarkko Kari, Florain Manea, and Ion Petre. Vol. 10307. LNCS. Springer, pp. 21–41.

Jones, Cliff B. and Astarte, Troy K. (2018). "Challenges for semantic description: comparing responses from the main approaches". In: *Proceedings of the 3rd School on Engineering Trustworthy Software Systems*. Ed. by Jonathan P. Bowen and Zhiming Liu. Lecture Notes in Computer Science 11174, pp. 176–217.

Jones, Cliff B. et al. (2004). *Reminiscences on Programming Languages and their Semantics*. Panel at Mathematical Foundations of Programming Language Semantics XX.

Kahn, Gilles (1987). "Natural Semantics". In: *STACS '87: Proc. Fourth Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, pp. 22–39.

Kelly, Michael J. and Randell, Brian (1958). *Preliminary Report on EASICODE*. Tech. rep. W/AT 216. Atomic Power Division, English Electric, Whetstone.

Klop, J. W., Meijer, J. J. C., and Rutten, J. J. M. M., eds. (1989). *J.W. de Bakker, 25 jaar semantiek*. CWI.

Knuth, Donald E. (1964a). "Backus Normal Form vs. Backus Naur Form". In: *Communications of the ACM* 7.12 (Dec. 1964), pp. 735–736. ISSN: 0001-0782. DOI: 10.1145/355588.365140. URL: http://doi.acm.org/10.1145/355588.365140.

Knuth, Donald E. (1964b). "Man or boy". In: *ALGOL Bulletin* 17.7 (1964).

Knuth, Donald E. (2014). *Let's not dumb down the history of computer science*. Online. Kailath lecture, Stanford University. URL: https://www.youtube.com/watch?v=gAXdDEQveKw.

Knuth, Donald E. and Trabb Pardo, Luis (1976). *The early development of programming languages*. Tech. rep. STAN-CS-76-562. Stanford University.

Kripke, Saul A. (1963). "Semantical Considerations on Modal and Intuitionistic Logic". In: *Acta Philosophica Fennica* 16, pp. 83–94.

Kruseman Aretz, F. E. J. (2003). *The Dijkstra–Zonneveld ALGOL 60 compiler for the Electrologica X1 The Dijkstra–Zonnefeld ALGOL 60 compiler for the Electrologica X1*. Technical Note SEN-N0301. CWI.

Kuich, W. (1969a). *On the Entropy of Context-Free Languages I. Structure Generating Functions*. Technical Report TR 25.092. IBM Laboratory Vienna, 1969.

Kuich, W. (1969b). *On the Entropy of Context-Free Languages II. Infinite Digraphs, Infinite Matrices, and Stochastic Processes*. Technical Report TR 25.093. IBM Laboratory Vienna, 1969.

Kuich, W. (1970). "Systems of pushdown acceptors and context-free grammars". In: *Elektronische Informationsverarbeitung und Kybernetik* 6.2, pp. 95–113.

Landin, P. J. (1966a). "A λ-Calculus Approach". In: *Advances in Programming and Non-numerical Computation*. Ed. by L. Fox. Hard copy + book. Pergamon Press, 1966, pp. 97–141.

Landin, P. J. (1966b). "The next 700 Programming Languages". In: *Communications of the ACM* 9 (1966), pp. 157–166.

Landin, Peter (2002). "Rod Burstall: A Personal Note". In: *Formal Aspects of Computing* 13.3, pp. 195–195.

Landin, Peter J. (1964). "The mechanical evaluation of expressions". In: *The Computer Journal* 6.4, pp. 308–320.

Landin, Peter J. (1965a). "A Correspondence Between ALGOL 60 and Church's Lambda-notation: Part I". In: *Communications of the ACM* 8.2 (Feb. 1965), pp. 89–101. URL: http://doi.acm.org/10.1145/363744.363749.

Landin, Peter J. (1965b). "A Correspondence Between ALGOL 60 and Church's Lambda-notation: Part II". In: *Communications of the ACM* 8.3 (Mar. 1965), pp. 158–167. URL: http://doi.acm.org/10.1145/363791.363804.

Landin, Peter J. (1966c). "A formal description of ALGOL 60". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 266–294.

Landin, Peter J. (1967). *Application of P. J. Landin for Readership in Computing Science*. Show's Landin's CV and a statement of research interests. Held in Strachey Papers, Bodleian Library, Oxford. Folder J56.

Landin, Peter J. (1968a). *Letter to members of IFIP Working Group 2.2*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39. 1968.

Landin, Peter J. (1968b). *Notice of fortnightly seminars by Hans Bekič*. Held in Cliff Jones' personal collection and to be archived. 1968.

Landin, Peter J. (1968c). *Paper presented by Landin for Zürich, 31/5 '68*. 1968.

Landin, Peter J. (1997). "Histories of discoveries of continuations: Belles-lettres with equivocal tenses". In: *ACM SIGPLAN Workshop on Continuations*. BRICS Notes NS-96-13.

Landin, Peter J (2000). "My years with Strachey". In: *Higher-Order and Symbolic Computation* 13.1, pp. 75–76.

Landin, Peter J. (2001). "Reminiscences". In: *Program Verification and Semantics: The Early Work*. A seminar held at the Science Museum, London. URL: `https://vimeo.com/8955127`.

Larner, R. A. and Nicholls, J. E. (1965). *Plan for Development of Formal Definition of PL/I*. IBM internal memo. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Laski, John (1974). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.22.

Lauer, P. E. (1971). "Consistent Formal Theories of the Semantics of Programming Languages". Printed as TR 25.121, IBM Lab. Vienna. PhD thesis. Queen's University of Belfast.

Lauer, Peter E. (1967). *The formal explicates of the notion of algorithm: an introduction to the theory of computability with special emphasis on the various formalisms underlying the alternate explicates*. Tech. rep. TR 25.072. IBM Laboratory Vienna.

Lauer, Peter E. (1968a). *An Introduction to H. Thiele's Notions of Algorithm, Algorithmic Process, and Graph Schemata Calculus*. Tech. rep. TR 25.079. IBM Laboratory Vienna, 1968.

Lauer, Peter E. (1968b). *Formal definition of ALGOL 60*. Tech. rep. 25.088. IBM Laboratory Vienna, 1968. URL: `http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Lau68.pdf`.

Lee, John A. N. (1972). "The formal definition of the BASIC language". In: *The Computer Journal* 15.1, pp. 37–41.

Lee, John A. N. (1995). *Computer pioneers – Thomas B. Steel*. URL: `http://history.computer.org/pioneers/steel.html`.

Lee, John A. N. and Delmore, W. (1969). "The Vienna Definition Language, a generalization of instruction definitions". In: *SIGPLAN Symposium on Programming Language Definitions, San Francisco*.

Ligler, George T (1975). “A mathematical approach to language design”. In: *Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, pp. 41–53.

Ligler, George Todd (1973). “A Survey of Approaches to Programming Language Semantics”. MA thesis. Oxford University.

Lindsey, C. H. (1993). “A History of ALGOL 68”. In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. ACM, pp. 97–132. DOI: 10.1145/154766.155365.

Løvengreen, Hans Henrik (1980). “Parallism in Ada”. In: *Towards a formal description of Ada*. Ed. by Dines Bjørner and Ole Nybye Oest. Lecture Notes in Computer Science 98. Springer-Verlag. Chap. 3.

Lucas, P. (1972). “On the Semantics of Programming Languages and Software Devices”. In: *(Rustin 1972)*, pp. 41–57.

Lucas, P. (1987). “VDM: Origins, Hopes, and Achievements”. In: *(Bjørner et al. 1987)*, pp. 1–18.

Lucas, P. et al. (1968). *Informal Introduction to the Abstract Syntax and Interpretation of PL/I*. Tech. rep. 25.083. IBM Laboratory Vienna, ULD Version II.

Lucas, Peter (1968). *Two constructive realisations of the block concept and their equivalence*. Tech. rep. TR 25.085. IBM Laboratory Vienna.

Lucas, Peter (1973). “On Program Correctness and the Stepwise Development of Implementations”. In: *Proceedings Convegno di Informatica Teorica*. University of Pisa, pp. 219–251.

Lucas, Peter (1978). “On the formalization of programming languages: Early history and main approaches”. In: *The Vienna Development Method: The Meta-Language*. Vol. 61. LNCS. Springer, pp. 1–23.

Lucas, Peter (1981). “Formal Semantics of Programming Languages: VDL.” In: *IBM Journal of Research and Development* 25.5, pp. 549–561.

Lucas, Peter and Bekič, Hans (1962). *Compilation of ALGOL, Part I—Organization of the Object Program*. Laboratory report LR 25.3.001. IBM Laboratory Vienna.

Lucas, Peter, Lauer, Peter E., and Stigleitner, H. (1968). *Method and notation for the formal definition of programming languages*. Tech. rep. 25.087. IBM Laboratory Vienna. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/VDL-TRs/TR25.087.pdf.

Lucas, Peter and Walk, Kurt (1969). “On the formal description of PL/I”. In: *Annual Review in Automatic Programming* 6, pp. 105–182.

Łukaszewicz, Leon (1986). "A Handful of Recollections about IFIP People". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. North Holland, pp. 295–299.

MacKenzie, Donald (2001). *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press.

MacQueen, David B (2015). *The history of Standard ML: Ideas, Principles, Culture*. Talk given at Higher-order, Typed, Inferred, Strict: ACM SIGPLAN ML Family Workshop. URL: Higher-order,Typed,Inferred,Strict:ACMSIGPLANMLFamilyWorkshop.

Mahoney, Michael S (1988). "The history of computing in the history of technology". In: *Annals of the History of Computing* 10.2, pp. 113–125.

Mahoney, Michael S. (1996a). "Making History". In: *History of Programming languages—II*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996, pp. 24–26. URL: http://doi.acm.org/10.1145/234286.1057808.

Mahoney, Michael S. (1996b). "What Makes History?" In: *History of Programming languages—II*. Ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996, pp. 831–832. DOI: 10.1145/234286.1057848. URL: http://doi.acm.org/10.1145/234286.1057848.

Mahoney, Michael S (2011). "Computer Science: The Search for a Mathematical Theory". In: *Histories of Computing*. Ed. by Thomas Haigh. Harvard University Press. Chap. 10, pp. 128–46.

Mahoney, Michael Sean (2002). "Software as Science—Science as Software". In: *History of Computing: Software Issues*. Ed. by Ulf Hashagen, Reinhard Keil-Slawik, and Arthur Norberg. Springer, pp. 25–48.

Mahoney, Michael Sean and Haigh, Thomas (2011). *Histories of Computing*. Cambridge, MA, USA: Harvard University Press. ISBN: 0674055683, 9780674055681.

Manna, Z. (1968a). *Formalization of Properties of Programs*. Technical Memorandum AI-64. Stanford Artificial Intelligence Department, 1968.

Manna, Z. (1968b). "Termination of Algorithms". PhD thesis. Carnegie-Mellon University, 1968.

Manna, Z. and McCarthy, J. (1969). "Properties of Programs and Partial Function Logic". In: *Machine Intelligence, 5*. Ed. by B. Meltzer and D. Michie. Edinburgh University Press, pp. 27–37.

Marcotty, Michael, Ledgard, Henry, and Bochmann, Gregor V (1976). "A sampler of formal definitions". In: *ACM Computing Surveys (CSUR)* 8.2, pp. 191–276.

Martin, D. C. (1964). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39.

Maurer, H. (1966a). *Bibliography on Theoretical Foundations of Programming Languages*. Technical Report TR 25.067. IBM Laboratory Vienna, 1966.

Maurer, H. (1966b). *Elimination of Certain PL/I Concepts by Means of Rewriting-Rules*. Tech. rep. LDV 14. IBM Laboratory Vienna, 1966.

Mazurkiewicz, Antoni W (1971). "Proving algorithms by tail functions". In: *Information and Control* 18.3, pp. 220–226.

McCarthy, J. (1959). "Programs with Common Sense". In: *Teddington Conference on the Mechanization of Thought Processes*.

McCarthy, John (1960). "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Communications of the ACM* 3.4, pp. 184–195.

McCarthy, John (1961). "A Basis for a Mathematical Theory of Computation, Preliminary Report". In: *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '61 (Western). ACM, pp. 225–238. URL: http://doi.acm.org/10.1145/1460690.1460715.

McCarthy, John (1963). "Towards a Mathematical Science of Computation". In: *IFIP Congress*. Vol. 62, pp. 21–28.

McCarthy, John (1965). *Letter to A. P. Ershov*. Held in the Ershov archive (online). URL: http://ershov.iis.nsk.su/en/node/777964.

McCarthy, John (1966). "A formal description of a subset of ALGOL". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 1–12.

McCarthy, John (1968). *Letter to A. P. Ershov*. Held in the Ershov archive (online). URL: http://ershov.iis.nsk.su/en/node/779549.

McCarthy, John (1980). "LISP—notes on its past and future—1980". In: *Proceedings of the 1980 ACM conference on LISP and functional programming*. ACM, pp. 5–viii.

McCarthy, John (1981). "History of LISP". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, pp. 173–197.

McCarthy, John and Painter, James A. (1967). "Correctness of a compiler for arithmetic expressions". In: *Mathematical aspects of computer science* 19.

McIlroy, Doug (1968). "Coroutines: Semantics in Search of a Syntax". Work done while a guest at the PRG in Oxford.

McIlroy, M. Douglas (2018). *Interview with Doug McIlroy*. Conducted by Cliff Jones.

Meertens, Lambert G. L. T. (2016). *ALGOL X and ALGOL Y*. Talk given at CWI Lectures in honour of Adrian van Wijngaarden.

Meyer, Bertrand (2011). *John Mccarthy*. Online. URL: https://cacm.acm.org/blogs/blog-cacm/138907-john-mccarthy/.

Michie, Donald (1971). "Computing: Can Britain set the pace?" In: *University of Edinburgh Bulletin* 8.3, pp. 1–2.

Milne, R. and Strachey, C. (1976). *A Theory of Programming Language Semantics (Parts A and B)*. Chapman and Hall.

Milne, Robert (2000). "From Language Concepts to Implementation Concepts". In: *Higher-Order and Symbolic Computation* 13.1, pp. 77–81.

Milne, Robert (2016). *Semantic relationships: reducing the separation between theory and practice*. Talk given at the Strachey 100 symposium, Oxford University.

Milne, Robert and Strachey, Christopher (1974). *A Theory of Programming Language Semantics*. Privately circulated. Submitted for the Adams Prize.

Milne, Robert E. (1972). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 276, C.241.

Milner, R. (1980). *A Calculus for Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer-Verlag.

Milner, R. (1992). "The Polyadic π-Calculus: A Tutorial". In: *Logic and Algebra of Specification*. Ed. by F. L. Bauer and W. Brauer. Springer-Verlag.

Milner, R., Morris, L., and Newey, M. (1975). "A Logic for Computable Functions with Reflexive and Polymorphic Types". In: *Proceedings of Conference on Proving and Improving Programs*.

Milner, Robin (1993). *Interviewed by Tony Dale*. Edinburgh, UK.

Moggi, Eugenio (1989). "An Abstract View of Programming Languages". PhD thesis. Edinburgh University Laboratory for the Foundation of Computer Science.

Moggi, Eugenio (1991). "Notions of computation and monads". In: *Information and computation* 93.1, pp. 55–92.

Mol, Liesbeth De and Primiero, Giuseppe (2015). "When Logic Meets Engineering: Introduction to Logical Issues in the History and Philosophy of Computer Science". In: *History and Philosophy of Logic* 36.3, pp. 195–204. DOI: 10.1080/01445340.2015.1084183. eprint: https://doi.org/10.1080/01445340.2015.1084183. URL: https://doi.org/10.1080/01445340.2015.1084183.

Moore, J Strother (2017). *Interview with J Strother Moore*. Conducted by Troy Astarte.

Moreau, Luc (1998). "A syntactic theory of dynamic binding". In: *Higher-Order and Symbolic Computation* 11.3, pp. 233–279.

Morris, Charles (1946). *Signs, Language, and Behavior*. Prentice-Hall.

Morris, F. L. (1970). "The next 700 Formal Language Descriptions". Manuscipt.

Moser, Nora B. (1954). "Compiler method of automatic programming". In: *Symposium on Automatic Programming for Digital Computers*. Washing, D.C.: Office of Naval Research, Dept. of the Navy, pp. 15–21.

Mosses, Peter D. (1992). *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge Press.

Mosses, Peter D (2000). "A foreword to 'Fundamental concepts in programming languages'". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 7–9.

Mosses, Peter D (2001). "The varieties of programming language semantics and their uses". In: *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, pp. 165–190.

Mosses, Peter D. (2011). "VDM semantics of programming languages: combinators and monads". In: *Formal Aspects of Computing* 23.2, pp. 221–238. DOI: 10.1007/s00165-009-0145-4. URL: http://dx.doi.org/10.1007/s00165-009-0145-4.

Mosses, Peter D. (2016). *Christopher Strachey's influence on the development of SIS, a semantics implementation system*. Talk given at the Strachey 100 symposium, Oxford University.

Mosses, Peter David (1974). *The mathematical semantics of ALGOL 60*. Tech. rep. Programming Research Group. URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/OCRd/Mosses74.pdf.

Mosses, Peter David (1975a). "Mathematical semantics and compiler generation". PhD thesis. University of Oxford, 1975.

Mosses, Peter David (1975b). "The semantics of semantic equations". In: *Mathematical Foundations of Computer Science: 3rd Symposium at Jadwisin near Warsaw, June 17–22, 1974*. Ed. by A. Blikle. Berlin, Heidelberg: Springer Berlin Heidelberg, 1975, pp. 409–422. URL: http://dx.doi.org/10.1007/3-540-07162-8_701.

Myers, Andrew (2011). *Stanford's John McCarthy, seminal figure of artificial intelligence, dies at 84*. Stanford Report, online. URL: https://news.stanford.edu/news/2011/october/john-mccarthy-obit-102511.html.

Napper, R. B. E. (1966). "A system for defining language and writing programs in "Natural English"". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 231–248.

Naur, Peter (1963). "Documentation Problems: ALGOL 60". In: *Communications of the ACM* 6.3 (Mar. 1963), pp. 77–79. DOI: 10.1145/366274.366286.

Naur, Peter (1968). “Successes and Failures of the ALGOL Effort”. In: *ALGOL Bulletin* 28, pp. 58–62.

Naur, Peter (1981a). “Formalization in program development”. In: *BIT Numerical Mathematics* 22.4 (1981), pp. 437–453. ISSN: 1572-9125. DOI: 10.1007/BF01934408. URL: http://dx.doi.org/10.1007/BF01934408.

Naur, Peter (1981b). “The European side of the last phase of the development of ALGOL 60”. In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, 1981. Chap. 3, pp. 92–137.

Neuhold, E J (1971). “The formal description of programming languages”. In: *IBM Systems Journal* 10.2, pp. 86–112. ISSN: 0018-8670.

Neuhold, Erich J. (2016). *Interview with Erich J. Neuhold*. Conducted by Troy Astarte and Cliff Jones.

Nivat, Maurice P. and Nolin, N. (1966). “Contribution to the definition of ALGOL semantics”. In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 149–159.

Nofre, David (2010). “Unraveling Algol: US, Europe, and the Creation of a Programming Language”. In: *IEEE Annals of the History of Computing* 32.2, pp. 58–68. DOI: 10.1109/MAHC.2010.4.

Nofre, David, Priestley, Mark, and Alberts, Gerard (2014). “When Technology Became Language: The Origins of the Linguistic Conception of Computer Programming, 1950–1960”. In: *Technology and Culture* 55.1, pp. 40–75.

Ollongren, A. (1971). *A Theory for the Objects of the Vienna Definition Language*. Technical Report 25.123. IBM Laboratory Vienna.

Pagan, Frank G. (1981). *Formal Specification of Programming Languages*. Prentice-Hall.

Painter, J. A. (1967a). *Semantic Correctness of a Compiler for an Algol-like Language*. Tech. rep. AI Memo 44. Computer Science Department, Stanford University, 1967.

Painter, James A. (1967b). *Invitation to Mathematical Theory of Computation conference, sent to A. P. Ershov*. Held in the Ershov archive (online). 1967. URL: http://ershov.iis.nsk.su/en/node/779029.

Park, D. (1969). “Fixpoint Induction and Proofs of Program Properties”. In: *Machine Intelligence, 5*. Ed. by B. Meltzer and D. Michie. Edinburgh University Press, pp. 59–78.

Park, David Michael Ritchie (1968). "Some Semantics for Data Structures". In: *Machine Intelligence, 3*. Ed. by Donald Michie. Edinburgh University Press, pp. 351–371.

Park, David Michael Ritchie (1970). *Letter to Dana Scott*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 275, C.237.

Peacock, Roger B (1970). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.22.

Peck, J. E. L. (1971). *Minutes of the 13th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by J. Peck. Held in Ljubljana. URL: http://ershov.iis.nsk.su/en/node/806010.

Peláez Valdez, María Eloína (1988). "A gift from Pandora's box: The software crisis." PhD thesis. University of Edinburgh.

Penrose, Roger (2000). "Reminiscences of Christopher Strachey". In: *Higher-Order and Symbolic Computation* 13.1, pp. 83–84.

Pérez, Jorge A. (2017). "Report on CWI Lectures in Honor of Adriaan van Wijngaarden". In: *ACM SIGLOG News* 4.1 (Feb. 2017), pp. 40–41. DOI: 10.1145/3051528.3051536.

Perlis, Alan J. (1981). "The American Side of the Development of ALGOL". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press. Chap. 3, pp. 75–91.

Perlis, Alan J. and Samelson, Klaus (1958). "Preliminary report: international algebraic language". In: *Communications of the ACM* 1.12, pp. 8–22.

Perlis, Alan J. and Samelson, Klaus (1959). "Report on the Algorithmic Language ALGOL by the ACM committee on programming languages and the GAMM committee on programming". In: *Numerische Mathematik* 1.1, pp. 41–60.

Petricek, Tomas (2017). "Miscomputation in software: Learning to live with errors". In: *The Art, Science, and Engineering of Programming* 1.2, pp. 14–1.

Plotkin, G. D. (1975). *A Powerdomain for Countable Non-determinism*. Tech. rep. 3. Department of A.I., University of Edinburgh.

Plotkin, G. D. (1976). "A Powerdomain Construction". In: 5, pp. 452–487.

Plotkin, Gordon (1971). "Automatic methods of inductive inference". PhD thesis. The University of Edinburgh.

Plotkin, Gordon D. (1977). "LCF considered as a programming language". In: *Theoretical computer science* 5.3, pp. 223–255.

Plotkin, Gordon D. (1994). *Interviewed by Tony Dale*. Manchester, UK.

Plotkin, Gordon D. (2004). "The origins of structural operational semantics". In: *Journal of Logic and Algebraic Programming* 60–61, pp. 3–15. ISSN: 1567-8326. DOI: doi:10.1016/j.jlap.2004.03.009. URL: http://www.elsevier.com/locate/jlap.

Plotkin, Gordon D. (2018). *Interview with Gordon D. Plotkin*. Conducted by Troy Astarte and Cliff Jones.

van der Poel, Willem L. (1986). "Some notes on the history of ALGOL". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. North Holland.

Priestley, Mark (2011). *A Science of Operations: Machines, Logic and the Invention of Programming*. Springer Science & Business Media.

Radin, George (1967). *Formal definition of PL/I*. IBM internal memo to H. Zemanek. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Radin, George (1981). "The early history and characteristics of PL/I". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, pp. 551–589.

Radin, George and Rogoway, H Paul (1965). "NPL: highlights of a new programming language". In: *Communications of the ACM* 8.1, pp. 9–17.

Radin, George and Schneider, Peter R. (1976). *An Architecture for an Extended Machine with Protected Addressing*. Technical report TR 00.2757. IBM Poughkeepsie Laboratory.

Randell, B. (1975). *The Origins of Digital Computers: Selected Papers*. Second. Springer-Verlag.

Randell, B. and Russell, L. J. (1962). "Discussions on ALGOL Translation at Mathematisch Centrum". In: *English Electric Report W/AT* 841.

Randell, Brian (1965). *Minutes of the 6th meeting of IFIP WG 2.1*. Online. Held in St Pierre de Chartreuse, France. Chaired by W. L. van der Poel. Archived by Andrei Ershov. URL: http://ershov.iis.nsk.su/en/node/778121.

Randell, Brian (2010). *Reminiscences of Whetstone ALGOL*. Tech. rep. CS-TR-1190. Newcastle University School of Computer Science.

Randell, Brian (2011). *Occurrence Nets Then and Now: The Path to Structured Occurrence Nets*. Tech. rep. CS-TR-1243. Computing Science, Newcastle University.

Randell, Brian (2016). *Interview with Brian Randell*. Conducted by Troy Astarte.

Randell, Brian and Russell, Lawford J. (1964). *Algol 60 Implementation*. Academic Press, Inc.

Ray, Baishakhi et al. (2017). "A Large-scale Study of Programming Languages and Code Quality in GitHub". In: *cacm* 60.10 (Sept. 2017), pp. 91–100. URL: http://doi.acm.org/10.1145/3126905.

Reynolds, J. C. (1972). "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of the 1972 ACM Annual Conference*. New York: ACM, pp. 717–740.

Reynolds, John C (1970). "GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept". In: *Communications of the ACM* 13.5, pp. 308–319.

Reynolds, John C (1974a). "On the relation between direct and continuation semantics". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1974, pp. 141–156.

Reynolds, John C (1974b). "Towards a theory of type structure". In: *Programming Symposium*. Springer. 1974, pp. 408–425.

Reynolds, John C. (1981). "The essence of Algol". In: *Algorithmic Languages: Proceedings of the International Symposium on Algorithmic Languages*. Ed. by J. W. de Bakker and J. C. van Vliet. Hard copy. North-Holland, pp. 345–372.

Reynolds, John C. (1993). "The discoveries of continuations". In: *Lisp and symbolic computation* 6.3-4, pp. 233–247.

Reynolds, John C. (2004). "The Discoveries of Continuations". In: *Program Verification and Semantics: Further Work*. A seminar held at the Science Museum, London.

Reynolds, John C. (2012). *Professional Vita*. Online. URL: https://www.cs.cmu.edu/~jcr/vita.pdf.

Richards, Martin (2000). "Christopher strachey and the Cambridge CPL compiler". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 85–88.

Rochester, Nathaniel (1966). "A formalization of two-dimensional syntax description". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 124–138.

Rochester, Nathaniel (1983). "The 701 Project as Seen by Its Chief Architect". In: *Annals of the History of Computing* 5.2, pp. 115–117.

Rodger, A (1938). *Report on Mr. Christopher Strachey*. Tech. rep. National Institute of Industrial Psychology, Vocational Guidance Department.

Rosen, Saul (1972). "Programming systems and languages 1965-1975". In: *Communications of the ACM* 15.7, pp. 591–600.

Ross, Douglas T. (1968). *Letter to members of IFIP Working Group 2.1*.

Rozenberg, Grzegorz, Bck, Thomas, and Kok, Joost N (2011). *Handbook of natural computing*. Springer Publishing Company, Incorporated.

Russell, Bertrand (1903). *Principles of Mathematics*. Cambridge University Press.

Russell, Bertrand (1931). *The Scientific Outlook*. London: George Allen and Unwiri.

Russell, Bertrand and Whitehead, Alfred North (1912). *Principia Mathematica*. Vol. 2. University Press.

Rustin, R. (1972). *Formal Semantics of Programming Languages*. Courant Computer Science Symposium 2, September 14-16, 1970. Prentice-Hall.

Ryder, Barbara G. and Hailpern, Brent, eds. (2007). *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. San Diego, California: ACM. ISBN: 978-1-59593-766-7.

Sammet, Jean E (1972). "Programming languages: history and future". In: *Communications of the ACM* 15.7, pp. 601–610.

Sammet, Jean E (1981). "The Early History of COBOL". In: *History of programming languages*. Ed. by Richard L. Wexelblat. Academic Press, pp. 199–243.

Schmidt, David A (2000). "Induction, domains, calculi: Strachey's contributions to programming-language engineering". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 89–101.

Schützenberger, Marcel-Paul (1966). "Classification of Chomsky languages". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 100–104.

Schwarzenberger, F. and Zemanek, H. (1966). *Editing Algorithms for Texts over Formal Grammars*. Tech. rep. 25.066. IBM Laboratory Vienna.

Scott, D. (1969). "A Type-Theoretical Alternative to CUCH, ISWIM, OWHY". Typed script – Oxford.

Scott, D. (1970). *The Lattice of Flow Diagrams*. Tech. rep. PRG-3. Oxford University Computing Laboratory, Programming Research Group.

Scott, D. (1971a). *Continuous Lattices*. Tech. rep. PRG-7. Oxford University Computing Laboratory, Programming Research Group, 1971.

Scott, D. (1973). "Models for Various Type-Free Calculi". In: *Studies in Logic and Foundations of Mathematics Vol. 74 (Proc. of the 4th International Congress for Logic, Methodology and Philosophy of Science, Bucharest, 1971)*. Ed. by P. Suppes et al. North Holland Publishing Company, pp. 158–187.

Scott, Dana (1971b). "The lattice of flow diagrams". In: *(Engeler 1971)*. Springer. 1971, pp. 311–366.

Scott, Dana (2000). “Some reflections on Strachey and his work”. In: *Higher-Order and Symbolic Computation* 13.1, pp. 103–114.

Scott, Dana and Strachey, Christopher (1971). *Toward a mathematical semantics for computer languages*. Technical Monograph PRG-6. Oxford University Computing Laboratory, Programming Research Group.

Scott, Dana S. (1972). *Letter to Robin Milner*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 276, C.241.

Scott, Dana S (1977). “Logic and programming languages”. In: *Communications of the ACM* 20.9, pp. 634–641.

Scott, Dana S (1993). “A type-theoretical alternative to ISWIM, CUCH, OWHY”. In: *Theoretical Computer Science* 121.1, pp. 411–440.

Scott, Dana S. (1994). *Interviewed by Tony Dale*. Carnegie-Mellon University.

Scott, Dana S. (2016). *Greetings to the Participants at “Strachey 100”*. A talk read out at the Strachey 100 centenary conference. Available online in the conference booklet. URL: `http://www.cs.ox.ac.uk/strachey100/Strachey_booklet.pdf`.

Scott, Dana S. (2018). *Looking Backward; Looking Forward*. Talk given at Domains13 workshop at FLoC.

Scott, Elizabeth and Johnstone, Adrian (2010). “GLL parsing”. In: *Electronic Notes in Theoretical Computer Science* 253.7, pp. 177–189.

Searle, John R (1980). “Minds, brains, and programs”. In: *Behavioral and brain sciences* 3.3, pp. 417–424.

Sintzoff, Michel (1967). “Existence of a Van Wijngaarden sytnax for every recursively enumerable set”. In: *Annales Soc. Sci. Bruxelles* 81.2, pp. 115–118.

Smullyan, Raymond M et al. (1961). *Theory of formal systems*. Princeton University Press.

Sowa, John F. (1974). *Analysis of Selected Aspects of GRAD*. Memorandum 125. IBM ASDD Products Architecture. URL: `http://www.jfsowa.com/computer/memo125.htm`.

Spivey, J. M. (1988). *Understanding Z—A Specification Language and its Formal Semantics*. Cambridge Tracts in Computer Science 3. Cambridge University Press.

Stedall, Jacqueline (2012). *The History of Mathematics: A Very Short Introduction*. Vol. 305. Oxford University Press.

Steel, T. B. (1966a). *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

Steel, T. B. (1973). *Minutes of the 15th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by T. B. Steel. Held in Munich. URL: http://ershov.iis.nsk.su/en/node/806075.

Steel, Thomas B. (1964). "Beginnings of a Theory of Information Handling". In: *cacm* 7.2 (Feb. 1964), pp. 97–103. URL: http://doi.acm.org/10.1145/363921.363937.

Steel, Thomas B. (1965). *Proposal for a working group on programming language description*. Held in the Ershov archive (online). URL: http://ershov.iis.nsk.su/en/node/805736.

Steel, Thomas B. (1966b). "A formalization of languages for programming language description". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 25–36.

Steel, Thomas B. (1966c). "Preface". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966.

Steel, Thomas B. (1967). *Letter to A. van Wijngaarden*.

Steel, Thomas B. (1970). *Letter to members of IFIP Working Group 2.2*.

Steggles, L Jason et al. (2006). "Qualitatively modelling and analysing genetic regulatory networks: a Petri net approach". In: *Bioinformatics* 23.3, pp. 336–343.

Stegmann, Claire and Backus, John W. (1979). "Pathfinder". In: *Think* 45.4.

Stoy, Joe (2000). "Christopher Strachey and Fundamental Concepts". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 115–117.

Stoy, Joseph E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA, USA: MIT Press. ISBN: 0262191474.

Stoy, Joseph E. (1980). "Foundations of denotational semantics". In: *Abstract Software Specifications: 1979 Copenhagen Winter School January 22 – February 2, 1979 Proceedings*. Ed. by Dines Bjørner. Springer Berlin Heidelberg, pp. 43–99. URL: http://dx.doi.org/10.1007/3-540-10007-5_35.

Stoy, Joseph E (1981). "The congruence of two programming language definitions". In: *Theoretical Computer Science* 13.2, pp. 151–174.

Stoy, Joseph E. (2016a). *Christopher Strachey and the Programming Research Group*. Talk given at the Strachey 100 symposium, Oxford University. 2016.

Stoy, Joseph E. (2016b). *Interview with Joseph E. Stoy*. Conducted by Troy Astarte. 2016.

Stoy, Joseph E and Strachey, C (1972a). "OS6—an experimental operating system for a small computer. Part 1: general principles and structure". In: *The Computer Journal* 15.2 (1972), pp. 117–124.

Stoy, Joseph E and Strachey, Christopher (1972b). "OS6—an experimental operating system for a small computer. Part 2: input/output and filing system". In: *The Computer Journal* 15.3 (1972), pp. 195–203.

Strachey, Barbara (1980). *Remarkable relations: the story of the Pearsall Smith family*. Gollancz.

Strachey, C. (1966a). "Towards a Formal Semantics". In: *(Steel 1966a)*. North-Holland, 1966.

Strachey, C. (1973a). *The Varieties of Programming Language*. Technical Monograph PRG-10. Oxford University Computing Lab, 1973.

Strachey, C. and Wilkes, M. V. (1961). "Some Proposals for Improving the Efficiency of ALGOL 60". In: *Commun. ACM* 4.11 (Nov. 1961), pp. 488–491. ISSN: 0001-0782. DOI: 10.1145/366813.366816. URL: http://doi.acm.org/10.1145/366813.366816.

Strachey, Christopher (1959). "Time sharing in large fast computers". In: *Communications of the ACM* 2.7, pp. 12–13.

Strachey, Christopher (1965a). "A general purpose macrogenerator". In: *The Computer Journal* 8.3 (1965), pp. 225–241.

Strachey, Christopher (1965b). *Report on Project MAC*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 253, A.66. 1965.

Strachey, Christopher (1966b). "System analysis and programming". In: *Scientific American* 215 (1966), pp. 112–124.

Strachey, Christopher (1967). "Fundamental Concepts in Programming Languages". Notes from a series of lectures given at the Summer School in Computer Programming held in Copenhagen in August 1967.

Strachey, Christopher (1970a). "Is Computing Science?" In: *Bulletin of the Institute of Mathematics and its Applications* (1970), pp. 80–82.

Strachey, Christopher (1971a). *Curriculum Vitae*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 248, A.3. Written by Strachey to send to the Times newspaper in case of the need for obitual information. 1971.

Strachey, Christopher (1971b). *Letter to Lord Halsbury*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 248, A.3. 1971.

Strachey, Christopher (1973b). "A mathematical semantics which can deal with full jumps". In: *Théorie des Algorithmes, des Langages et de la Programmation*. IRIA (INRIA), 1973, pp. 175–191.

Strachey, Christopher (1973c). *The Lighthill Debate*. Online. Comment made at the debate on AI arranged by Sir James Lighthill. 1973. URL: https://www.youtube.com/watch?v=3GZWFnWOqkA.

Strachey, Christopher (2000). "Fundamental concepts in programming languages". In: *Higher-order and symbolic computation* 13.1-2, pp. 11–49.

Strachey, Christopher and Wadsworth, Christopher Peter (1974). *Continuations: A Mathematical Semantics for Handling Jumps*. Monograph PRG-11. Oxford University Computing Laboratory, Programming Research Group.

Strachey, Christopher S. (1952). "Logical or Non-mathematical Programmes". In: *Proceedings of the 1952 ACM National Meeting (Toronto)*. ACM '52. Toronto, Ontario, Canada, pp. 46–49. URL: http://doi.acm.org/10.1145/800259.808992.

Strachey, Christopher S. (1964a). *Letter to Donald Michie*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.25. 1964.

Strachey, Christopher S. (1964b). *Letter to Royal Society*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39. 1964.

Strachey, Christopher S. (1965c). *Letter to Lord Halsbury*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.18. 1965.

Strachey, Christopher S. (1968a). *Letter to F. H. Stewart*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.54. 1968.

Strachey, Christopher S. (1968b). *Letter to Lord Halsbury*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.19. 1968.

Strachey, Christopher S. (1969). *Letter to Dana Scott*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.40.

Strachey, Christopher S. (1970b). *Letter to J. B. Butterworth*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.58. 1970.

Strachey, Christopher S. (1970c). "Program Segments". Christopher Strachey Collection, Bodleian Library, Oxford. Box 275, C.229. Manuscript. 1970.

Strachey, Christopher S. (1971c). *An Abstract Model for Storage (1st Draft)*. Tech. rep. Oxford University Programming Research Group, 1971.

Strachey, Christopher S. (1971d). *Letter to D. F. Hartley*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.59. 1971.

Strachey, Christopher S. (1972a). *Letter to Olvi L. Mangasarian*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.59. 1972.

Strachey, Christopher S. (1972b). *Letter to The Editor [of The Times]*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.52. 1972.

Strachey, Christopher S. (1973d). *Letter to A. R. K. Watkinson*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.47. 1973.

Strachey, Christopher S. (1974a). *Letter to Chris Wadsworth*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.44. 1974.

Strachey, Christopher S. (1974b). *Letter to John Laski*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 300, J.22. 1974.

Strachey, Christopher S. and Gill, Stanley (1962). "Correspondence". In: *The Computer Journal* 5.1, p. 71.

Strachey, Ray (1928). *The cause: a short history of the women's movement in Great Britain*. G Bell & Sons.

Study Group XI (1980). *CHILL Language Definition*. Tech. rep. C.C.I.T.T. Period 1977–1980.

Tarski, Alfred (1944). "The semantic conception of truth: and the foundations of semantics". In: *Philosophy and phenomenological research* 4.3, pp. 341–376.

TC-2 (1968). *Report of IFIP TC 2 to the IFIP General Assembly in the matter of the Algorithmic Language ALGOL 68*. Held in the Ershov archive (online). URL: `http://ershov.iis.nsk.su/en/node/805977`.

TC-2 (1972). *Minutes of the 14th meeting of IFIP TC2*. Held in the Ershov archive (online). Held in Zakopane, Poland. Minutes incomplete and do not show chair or secretary. URL: `http://ershov.iis.nsk.su/en/node/806019`.

Tennent, R. D. (1981). *Principles of Programming Languages*. Prentice-Hall International.

Tennent, Robert D. (1976). "The Denotational Semantics of Programming Languages". In: *Communications of the ACM* 19, pp. 437–453.

Tennent, Robert D (1977). "Language design methods based on semantic principles". In: *Acta Informatica* 8.2, pp. 97–112.

Tennent, Robert D and Ghica, Dan R (2000). "Abstract models of storage". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 119–129.

Teufelhart, N. (1965). *Minutes of the 6th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in Nice. URL: `http://ershov.iis.nsk.su/en/node/805734`.

Teufelhart, N. (1966). *Minutes of the 7th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in London. URL: `http://ershov.iis.nsk.su/en/node/805755`.

Teufelhart, N. (1967). *Minutes of the 8th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in Oslo. URL: http://ershov.iis.nsk.su/en/node/805766.

Teufelhart, N. (1969). *Minutes of the 10th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in Guildford. URL: http://ershov.iis.nsk.su/en/node/805956.

Trakhtenbrot, B. A., Halpern, J. Y., and Meyer, A. R. (1983). *From Denotational to Operational and Axiomatic Semantics for Algol-like Langauges: An Overview*. Tech. rep. RJ 4105. IBM Research Division, New York.

Turing, A. M. (1949). "Checking a Large Routine". In: *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, pp. 67–69.

Turner, Raymond (2007). "Understanding programming languages". In: *Minds and Machines* 17.2, pp. 203–216.

Turner, Raymond (2009). "The Meaning of Programming Languages". In: *American Philosophical Association Newsletter on Philosophy and Computers* 9.1, pp. 2–6.

Turner, Raymond (2018). *Computational artifacts: Towards a philosophy of computer science*. Springer.

Turski, W. M. (1967). *Minutes of the 8th meeting of IFIP WG 2.1*. Online. Held in Zandvoort, Netherlands. Chaired by W. L. van der Poel. Archived by Andrei Ershov. URL: http://ershov.iis.nsk.su/en/node/778136.

Turski, W. M. (1981). "ALGOL 68 Revisited Twelve Years Later or from AAD to ADA". In: *Algorithmic Languages*. Ed. by J. W. de Bakker and J. C. van Vliet. IFIP, North-Holland, pp. 417–431.

ULD-III (1966). *Formal Definition of PL/I (Universal Language Document No. 3)*. Tech. rep. 25.071. Author given as 'PL/I – Definition Group of the Vienna Laboratory'. IBM Laboratory Vienna.

Urschler, G. (1969a). *Concrete Syntax of PL/I*. Technical Report TR 25.096. ULD Version III. IBM Laboratory Vienna, 1969.

Urschler, G. (1969b). *Translation of PL/I into Abstract Syntax*. Technical Report TR 25.097. ULD Version III. IBM Laboratory Vienna, 1969.

Utman, R. E. (1962a). *Minutes of the 1st meeting of IFIP TC2*. Chaired by H. Zemanek. Held in Rome. 1962.

Utman, R. E. (1962b). *Minutes of the 2nd meeting of IFIP TC2*. Chaired by H. Zemanek. Held in Munich. 1962.

Utman, R. E. (1963). *Minutes of the 3rd meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in Oslo. URL: http://ershov.iis.nsk.su/en/node/805662.

Utman, R. E. (1964a). *Minutes of the 3rd meeting of IFIP WG 2.1*. Online. Held in Tutzing, West Germany. Chaired by W. L. van der Poel. Archived by Andrei Ershov. 1964. URL: http://ershov.iis.nsk.su/en/node/778087.

Utman, R. E. (1964b). *Minutes of the 4th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in Liblice. 1964. URL: http://ershov.iis.nsk.su/en/node/805663.

Utman, R. E. (1964c). *Minutes of the 4th meeting of IFIP WG 2.1*. Online. Held in Baden-bei-Wien, Austria. Chaired by W. L. van der Poel. Archived by Andrei Ershov. 1964. URL: http://ershov.iis.nsk.su/en/node/778103.

Utman, R. E. (1965). *Minutes of the 5th meeting of IFIP TC2*. Held in the Ershov archive (online). Chaired by H. Zemanek. Held in New York. URL: http://ershov.iis.nsk.su/en/node/805728.

de Vere Roberts, M. (1965). *Radiogram to Kurt Bandat*. IBM internal memo. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents.

Verrijn-Stuart, A. A. (1987). "Obituary, A. van Wijngaarden". In: *IFIP Newsletter* 4.3.

Wadsworth, C. P. (1976). "The Relation Between Computational and Denotational Properties for Scott's D_∞-Models of the Lambda-Calculus". In: *SIAM Journal on Computing* 5.3, pp. 488–521.

Wadsworth, Christopher P. (1970). *Another approach to jumps—continuations*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 275, C.238.

Wadsworth, Christopher P. (1972). *Notes on continuations*. Unpublished.

Wadsworth, Christopher P. (1974). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 302, J.44.

Wadsworth, Christopher P. (2000). "Continuations revisited". In: *Higher-Order and Symbolic Computation* 13.1-2, pp. 131–133.

Wadsworth, Christopher Peter (1971). "Semantics and Pragmatics of the Lambda-Calculus". PhD thesis. Programming Research Group, University of Oxford.

Walk, K. et al. (1968). *Abstract Syntax and Interpretation of PL/I*. Tech. rep. 25.082. IBM Laboratory Vienna, ULD Version II.

Walk, K. et al. (1969). *Abstract Syntax and Interpretation of PL/I*. Technical Report TR 25.098. IBM Laboratory Vienna.

Walk, Kurt (1966). "Entropy and testability of context-free languages". In: *Formal Language Description Languages for Computer Programming*. North-Holland, pp. 105–123.

Walk, Kurt (1967a). *Minutes of the 1st meeting of IFIP WG 2.2 on Formal Language Description Languages*. Kept in the van Wijngaarden archive. Held in Porto Conte, Alghero, Sardinia. Chaired by T. B. Steel. 1967.

Walk, Kurt (1967b). *PL/I formal definition*. IBM internal memo to H. Zemanek. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I History Documents. 1967.

Walk, Kurt (1968). *Minutes of the 2nd meeting of IFIP WG 2.2 on Formal Language Description Languages*. Held in Vedbaek-Copenhagen, Denmark. Chaired by T. B. Steel.

Walk, Kurt (1969a). *Minutes of the 3rd meeting of IFIP WG 2.2 on Formal Language Description Languages*. Held in Vienna, Austria. Chaired by T. B. Steel. 1969.

Walk, Kurt (1969b). *Minutes of the 4th meeting of IFIP WG 2.2 on Formal Language Description Languages*. Held in Colchester, Essex, England. Chaired by T. B. Steel. 1969.

Walk, Kurt (1970). *Minutes of the 5th meeting of IFIP WG 2.2 on Formal Language Description Languages*. Held in Boston, USA. Chaired by T. B. Steel.

Walk, Kurt (2002). "Roots of Computing in Austria: Contributions of the IBM Vienna Laboratory and Changes of Paradigms and Priorities in Information Technology". In: *Human Choice and Computers*. Springer, pp. 77–87.

Walk, Kurt (2015). "Jahre mit Prof. Dr. Heinz Zemanek (Years with Prof. Dr. Heinz Zemanek)". In: *In memoriam Heinz Zemanek*. Ed. by Johann Stockinger Karl Anton Froeschl Gerhard Chroust. Vol. Band-311. OCG. ISBN: 978-3-903035-00-3.

Wardhaugh, Benjamin (2010). *How to Read Historical Mathematics*. Princeton University Press.

Wegner, Peter (1972). "The Vienna definition language". In: *ACM Computing Surveys (CSUR)* 4.1, pp. 5–63.

Weissenböck, F. (1975). *A Formal Interface Specification*. Tech. rep. 25.141. IBM Laboratory Vienna.

Welsh, A. (1982). "The Specification, Design and Implementation of NDB". Also published as technical report UMCS-82-10-1. MA thesis. Department of Computer Science, University of Manchester.

Welsh, A. (1984). "A Database Programming Language: Definition, Implementation and Correctness Proofs". Also published as technical report UMCS-84-10-1. PhD thesis. Department of Computer Science, University of Manchester.

Wexelblat, Richard L., ed. (1981). *History of programming languages*. Academic Press.

van Wijngaarden, A. and Mailloux, B. J. (1966). *A draft proposal for the algorithmic language ALGOL X*. Working paper 47. IFIP WG 2.1.

van Wijngaarden, A. et al. (1969). *Report on the Algorithmic Language ALGOL 68*. Second printing , MR 101. Mathematisch Centrum, Amsterdam.

van Wijngaarden, A. et al. (1977). "Revised Report on the Algorithmic Language ALGOL 68". In: *SIGPLAN Not.* 12.5 (May 1977), pp. 1–70. ISSN: 0362-1340. DOI: 10.1145/954652.1781176. URL: http://doi.acm.org/10.1145/954652.1781176.

van Wijngaarden, Aadrian (1966a). "Numerical analysis as an independent science". In: *BIT Numerical Mathematics* 6.1 (1966), pp. 66–81.

van Wijngaarden, Adriaan (1959). "The state of computer circuits containing memory elements". In: *Proceedings of the International Symposium on the Theory of Switching, Cambridge, MA, 2–5 April 1957*. Ed. by Howard Aiken. Vol. XXX. Annuals of the Computation Laboratory. Harvard University Press.

van Wijngaarden, Adriaan (1962). "Generalized ALGOL". In: *Symbolic Languages in Data Processing, Proc. Symp. Intl, Computation Center Rome*, pp. 409–419.

van Wijngaarden, Adriaan (1965). *Orthogonal design and the description of a formal language*. Technical report MR 76. Distributed at WG 2.1 meeting in St. Pierre de Chartreuse in October 1965. Mathematisch Centrum.

van Wijngaarden, Adriaan (1966b). "Recursive definition of syntax and semantics". In: *Formal Language Description Languages for Computer Programming*. North-Holland, 1966, pp. 13–24.

van Wijngaarden, Adriaan (1979). "Languageless programming". In: *Proceedings on Algorithms in Modern Mathematics and Computer Science*. Springer-Verlag, pp. 459–459.

van Wijngaarden, Adriaan et al. (1968). *Draft report on the Algorithmic Language ALGOL 68*. Report MR 93. Mathematisch Centrum.

Wilkes, M. V. (1985). *Memoirs of a Computer Pioneer*. MIT Press.

Wilkes, Maurice Vincent, Wheeler, David J., and Gill, Stanley (1951). *The preparation of programs for an electronic digital computer: with special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press.

Wilkinson, James Hardy (1972). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 248, A.3.

Wirth, N (1965). *A Proposal for a Report on a Successor of ALGOL 60*. Tech. rep. MR 75/65. Mathematisch Centrum.

Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

Wirth, N. and Hoare, C.A.R. (1966). "A Contribution to the Development of ALGOL". In: *Communications of the ACM* 9.6, pp. 413–432.

Wirth, Niklaus (1963). "A generalization of ALGOL". In: *Communications of the ACM* 6.9, pp. 547–554.

Wirth, Niklaus and Weber, Helmut (1966a). "EULER: a generalization of ALGOL and it formal definition: Part I". In: *Communications of the ACM* 9.1 (1966). hard copy of TR, pp. 13–25.

Wirth, Niklaus and Weber, Helmut (1966b). "EULER: a generalization of ALGOL, and its formal definition: Part II". In: *Communications of the ACM* 9.2 (1966), pp. 89–99.

Wolczko, M. I. (1988). "Semantics of Object-Oriented Languages". Also published as Technical Report UMCS-88-6-1. PhD thesis. Department of Computer Science, University of Manchester.

Woodger, M. and Green, J. (1966). "Summary Session". In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 220–224. URL: `http://doi.acm.org/10.1145/365230.365271`.

Woodger, Michael (1964). *Letter to A. P. Ershov*. Held in the Ershov archive (online). URL: `http://ershov.iis.nsk.su/en/node/777280`.

Woodger, Michael (1965). *Report on the 5th meeting of IFIP WG 2.1*. Online. Held in Princeton, USA. Chaired by W. L. van der Poel. Archived by Andrei Ershov. No minutes were published for this meeting and Woodger's report is the best record. URL: `http://ershov.iis.nsk.su/en/node/778105`.

Zemanek, H. (1965). *Semiotics and Programming Languages*. Tech. rep. 25.057. Also published in CACM 9.3 (Mar 1966). IBM Laboratory Vienna.

Zemanek, Heinz (1964a). *Letter to A. van Wijngaarden*. Kept in the van Wijngaarden archive, CWI. May 1964.

Zemanek, Heinz (1964b). *Report on the Vienna conference*. 1964.

Zemanek, Heinz (1966). "Semiotics and Programming Languages". In: *Communications of the ACM* 9.3, p. 5.

Zemanek, Heinz (1967a). *Letter to Christopher Strachey*. Christopher Strachey Collection, Bodleian Library, Oxford. Box 287, E.39. 1967.

Zemanek, Heinz (1967b). *Review of the formal definition of PL/I*. Sent to attendees of a workshop on the ULD. Technical University of Vienna NL. 14. 072/2 Zemanek. PL/I Second Version. 1967.

Zemanek, Heinz (1972a). "Some philosophical aspects of information processing". In: *The Skyline of Information Processing: Proceedings of the tenth anniversary celebration of the IFIP*. Ed. by Heinz Zemanek. IFIP. North-Holland, 1972.

Zemanek, Heinz, ed. (1972b). *The Skyline of Information Processing: Proceedings of the tenth anniversary celebration of the IFIP*. IFIP. North-Holland, 1972.

Zemanek, Heinz (1980). "Abstract Architecture". In: *Abstract Software Specifications: 1979 Copenhagen Winter School January 22 – February 2, 1979 Proceedings*. Ed. by Dines Bjørner. Vol. 86. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-540-38136-5.

Zemanek, Heinz (1981). "The Role of Professor A. van Wijngaarden in the History of IFIP". In: *Algorithmic Languages*. Ed. by Jacobus Willem de Bakker and J. C. van Vliet. North-Holland Publishing Company.

Zemanek, Heinz, ed. (1986a). *A Quarter Century of IFIP*. North Holland, 1986.

Zemanek, Heinz (1986b). "Hans Bekic (1936–1982)". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. North Holland, 1986, p. 464.

Zemanek, Heinz (1986c). "Klaus Samelson (1918 - 1980)". In: *A Quarter Century of IFIP*. Ed. by Heinz Zemanek. North Holland, 1986, p. 457.

Zemanek, Heinz and Aspray, William (1987). *Interview: Heinz Zemanek*. Typescript.

Zhang, Yingzhou and Xu, Baowen (2004). "A Survey of Semantic Description Frameworks for Programming Languages". In: *ACM SIGPLAN Notices* 39.3, p. 17.

Zimmermann, K. (1969). *Outline of a formal definition of FORTRAN*. Laboratory Report LR 25.3.053. IBM Lab. Vienna.