Minds & Machines (2017) 27:449–472
DOI 10.1007/s11023-017-9432-7

AI and the Origins of the Functional Programming Language Style

Mark Priestley[1]

Received: 26 October 2016 / Accepted: 22 April 2017 / Published online: 2 May 2017

Abstract The Lisp programming language is often described as the first functional programming language and also as an important early AI language. In the history of functional programming, however, it occupies a rather anomalous position, as the circumstances of its development do not fit well with the widely accepted view that functional languages have been developed through a theoretically-inspired project of deriving practical programming languages from the lambda calculus. This paper examines the origins of Lisp in the early AI programming work of the mid-to-late 1950s, and in particular in the work of Allen Newell, Cliff Shaw and Herbert Simon. Their 1956 program, the Logic Theory Machine, introduced new ideas about data and program structures that were articulated in response to perceived limitations in existing programming technique. Later writers, notably John Backus, have described these features as constituting a "programming language style" distinct from the traditional style that preceded it. The paper examines the origins of the earlier style in practices of manual computation, analyses the key technical differences between it and the style first manifested in the Logic Theory Machine, and concludes that programming practice and experience play a large and underappreciated role in the development of programming styles and languages.

Keywords History of AI · Functional programming · Lisp · Lambda calculus · Logic theory machine · IPL

In 1977 John Backus became the fourteenth winner of the Association for Computing Machinery's prestigious Turing award. The award citation highlighted two of Backus's technical achievements: leading the group that developed the FORTRAN system in the mid-1950s, and introducing the metasyntactic notation

✉ Mark Priestley
m.priestley@gmail.com

[1] 3 Thurlestone Road, London SE27 0PE, UK

now known as BNF to the Algol 60 project. In his Turing award lecture, however, Backus appeared to take a rather negative view of his award-winning achievements. He described "conventional programming languages", a class surely including Fortran along with Algol's many descendants, as "fat and flabby" and creating "unnecessary confusion in the way we think about programs".[1]

For Backus, the deficiencies of these languages reflected shortcomings in machine architecture. He explained that "[c]onventional programming languages are basically high level, complex versions of the von Neumann computer", by which he meant not just the machine's architecture but also its code:

> Von Neumann programming languages use variables to imitate the computer's storage cells; control statements to elaborate its jump and test instructions; and assignment statements imitate its fetching, storing and arithmetic. (Backus 1978, 615)

Backus's particular objection was to the assignment statement. He claimed that it limited programmers to thinking in terms of one word of data at a time, a restriction imposed by what he termed the "von Neumann bottleneck". By this, Backus referred to the fact that conventional computers only transferred one word of data at a time between the store and the central processor. As a consequence, programming languages forced programmers to focus on managing the details of this low-level data transfer. Backus believed that functional languages could remove this burden, making programming easier and program texts simpler and clearer. However, he also found the functional, or applicative, style as it existed in 1978 unsatisfactory:

> [M]ost applicative systems employ the substitution operator of the lambda calculus as their basic operation. This operation is one of virtually unlimited power, but its complete and efficient realization presents great difficulties to the machine designer. Furthermore, in an effort to introduce storage and to improve their efficiency on von Neumann computers, applicative systems have tended to become engulfed in a large von Neumann system. For example, pure Lisp is often buried in large extensions with many von Neumann features. (Backus 1978, 616)

The constructive part of his lecture was devoted to outlining "a functional style and its algebra of programs". In place of the lambda calculus, Backus proposed to base his system, known as FP, on a set of "combining forms" which would allow functions to be defined without using the machinery of variables and substitution. By choosing forms with "attractive algebraic properties", Backus aimed to make his FP system a suitable vehicle for reasoning about programs as well as writing them.

For Backus, the existence of different styles was explained by the fact that "underlying every programming language is a model of a computing system that its programs control". In the case of the von Neumann languages this was a physical model, the "von Neumann architecture" of the conventional computer; functional languages, in contrast, were based on mathematical models such as the lambda calculus or Backus's own set of combining forms. In both cases, however, language

[1] For the text of the lecture, from which the inline quotes in this section are taken, see Backus (1978).

characteristics derived from the properties of the underlying model, and Backus made no mention of the possible influence of programming practices or problems on language style. This omission is noteworthy, because the historiography of programming languages had, at least intermittently, paid attention to the role of such matters in the development of individual languages. The pioneering work of Saul Rosen and Jean Sammet grouped languages by their area of application, and accounts of the origins of particular languages, such as those found in the ACM SIGPLAN conferences on the history of programming languages, usually began by discussing the motivation for the development of the language. Indeed, at the first conference, in 1978, Backus himself discussed the origins of Fortran and stressed a number of these aspects, such as the demand for automatic programming in the early 1950s and the economics of programming.[2]

The differing properties of their underlying models affected Backus's evaluation of the two language styles. The von Neumann architecture was first of all an approach to hardware design, and for Backus this meant that the conventional languages based on it were inelegant, inefficient, and unsuitable for expressing more abstract ideas about program structure. He considered that applicative languages, on the other hand, should be based on abstract mathematical models specifically chosen to deliver desirable properties. Subsequent historical accounts of functional programming have tended to follow Backus's lead here, positioning the mathematical model as the historical ancestor of the programming languages themselves. In 1989, Paul Hudak, the co-designer of the Haskell language, wrote a survey article including a historical account of the development of functional languages, structured as comments on and explications of a series of noteworthy languages (Hudak 1989). Hudak's account began with the lambda calculus, discussed Lisp, ISWIM, and APL before reaching Backus's FP system, and then moved on to the functional languages of the 1980s, concluding with Haskell itself. A more recent survey in 2012 by David Turner, designer of the functional languages SASL, KRC, and Miranda, similarly examined a series of "milestones" (Turner 2012). Turner's history, like Hudak's, began with the lambda calculus and Lisp and ended with Haskell, though there were minor differences in the languages discussed en route.

It is striking that in these accounts the lambda calculus has recovered from Backus's rather dismissive evaluation, and the development of the functional programming style is now presented as a progressive and relatively unproblematic articulation of the original insights of the lambda calculus. As Hudak put it,

> the lambda calculus is usually regarded as the first functional language, although it was certainly not thought of as a programming language at the time, given that there were no computers on which to run the programs. In any case, modern functional languages can be thought of as (nontrivial) embellishments of the lambda calculus. (Hudak 1989, 363)

[2] Rosen (1967) and Sammet (1969) are the classic references from the 1960s. Backus (1981) reflected on the history of Fortran at the first ACM SIGPLAN History of Programming Languages conference in 1978. The papers from that conference were collected and published as Wexelblat (1981).

Lisp features heavily in histories of functional programming, and Turner described it, rather than the lambda calculus, as "the first functional programming language". At the same conference at which Backus described the origins of Fortran, John McCarthy (1981) presented his view of the early development of Lisp. As both Hudak and Turner acknowledged, however, McCarthy's account raised problems for the view that functional languages were simply an articulation of ideas present in the lambda calculus.

> It is often thought that the lambda calculus also formed the foundation for Lisp, but this in fact appears not to be the case [McCarthy 1978]. The impact of the lambda calculus on early Lisp development was minimal, and it has only been very recently that Lisp has begun to evolve more towards lambda calculus ideals. (Hudak 1989, 363, citation in original)

> LISP was not based on the lambda calculus, despite using the word "LAMBDA" to denote functions. At the time he invented LISP. McCarthy was aware of (Church 1941) but had not studied it. (Turner 2012, citation in original)

These comments raise a number of questions, of which the most obvious is, if Lisp was not, in fact, straight-forwardly derived from the lambda calculus, where did it come from? What are the real historical origins of the functional programming style? This paper proposes an answer these questions based on an examination of the programming work carried out by the first AI researchers in the mid-1950s that informed the original Lisp design. It will turn out that linguistic innovation was indeed prompted by perceived shortcomings in the existing "von Neumann style" of language, but not the ones highlighted by subsequent writers on functional languages.

Backus's terminology of "language style" is perhaps rather unfamiliar, and in Sect. 1 I examine how "style" and the related term "paradigm" came to be used in connection with programming languages, and what an historical account of the origins of a "language style" might involve. The structure of the remainder of the paper is as follows. Section 2 traces the origins of the functional style to early experiments in programming AI applications in the mid-1950s. These experiments led researchers to develop new programming concepts and techniques, and in the course of doing so, to reflect on what they saw as the limitations of existing styles of programming. In Sect. 3, I use this self-reflection on the part of programmers as the starting point for a description of the conventional style, which I tag with the label "programming as planning". One of the key innovations of the md-1950s was in the way subroutines were used, and in preparation for a discussion of this in Sect. 5, Sect. 4 describes the role of subroutines in the "programming as planning" style. Section 6 then describes how these technical innovations became integrated in a new language, Lisp. Finally, Sect. 7 discusses some general conclusions emerging from the historical survey.

1 On the Very Idea of Programming Language Style

Backus's use of the notion of "style" to distinguish the two types of language he is comparing might seem natural enough, but the developers of the programming languages of the late 1950s did not think of themselves as producing stylistic variants on existing work. New languages were proposed to address practical difficulties in programming, and languages were described according to the application area they addressed or the particular features that they offered. Lisp, for example, was routinely grouped together with the non-functional languages IPL, COMIT and FLPL as a list-processing language.[3] Why did Backus instead appeal to the notion of "style" as a way to classify languages?

Programming languages first appeared as objects of historical study in the mid-1960s, a period capped by the publication of Saul Rosen's collection of reprints and Jean Sammet's comprehensive survey.[4] Rosen did not attempt to categorize languages other than by describing a small group as "Languages for Processing Lists and Strings of Symbols". Sammet, on the other hand, explicitly addressed the question of classification, and proposed a rather Borgesian taxonomy:

> it is a little easier to propose definitions for classes of programming languages. The terms to be defined are the following: Procedure-oriented and nonprocedural; problem-oriented, application-oriented, and special purpose; problem-defining, problem-describing, and problem-solving; hardware, publication, and reference. Note that some of these are overlapping and that a particular language may fall into more than one of these categories. (Sammet 1969, 19)

However, many programming languages had in fact been developed for specific application areas, such as COBOL, a project that Sammet herself had been involved in, and in the body of the book she actually classified languages according to their intended purpose or application area. Languages were grouped into chapters with titles such as "languages for numerical scientific problems", "languages for business data processing problems" and "string and list processing languages".

As Sammet continued to write about programming languages in the early 1970s, however, she began to talk about "style" as an additional dimension along which programming languages could be described.

> It is probably true that a single extensible language must restrict itself to a particular type of application area. Even within these, some of the variations in style can be enormous. It often turns out that the style of a language is based more on the personal views of the person who developed it than on any rational basis. Thus languages which are different in purpose and application area may have a similar syntax. Conversely, the same application area will surely breed different syntactic styles. (Sammet 1971, 142)

[3] See, for example, Feigenbaum and Feldman (1963), vi, Rosen (1967) and Sammet (1969).

[4] For a more extended discussion of the historiography of programming languages, see Priestley (2008), 12–17.

"Style" appears here as an attribute of a language. In the following year, Sammet discussed style in more detail, but now related it to the choices that could be made by an individual programmer:

> Style in a programming language has many facets, ranging from personal views on the importance (or non-importance) of blanks and punctuation, to the use (or non-use) of a GOTO command, to the selection of a particular word for getting input data (e.g., READ vs. GET). The major identifiable elements of style are vocabulary, personal preferences, and requirements affecting style. (Sammet 1972, 304)

Note the shift here from talking about the style *of* a language to style *in* a language. Sammet is no longer using style as a way to categorize or distinguish programming languages. Rather, the term now refers to the linguistic choices that a programmer can make when writing a program, such as choosing long or short identifiers, or eschewing the newly controversial GOTO statement.

An early use of this notion of style appeared in a 1964 discussion of Lisp by Fischer Black.[5] Black argued that style was not a measure of a program's efficiency, but of

> how easy it is to write or read a program, how easy it is to explain the program to someone else … above all, style is a matter of taste, of aesthetics, of what you think looks nice, of what you think is elegant. (Black 1964, 96)

Style, for Black, was "developed by a series of choices". After describing a number of rather superficial alternatives, he focused on "two of the more complex choices you can make in LISP", namely the decision of whether or not to use the language's "functionals" and its so-called "program feature".

The notion of style as a set of choices made by a programmer was popularized in a 1974 book, *The Elements of Programming Style*, written by Bell Labs programmers Brian Kernighan and P. J. Plauger. The book was intended as a programming language analogue of the style guides available for writers of English, such as Strunk and White's *The Elements of Style*, a book that Kernighan and Plauger described as a model for their own. For Kernighan and Plauger, style was not an attribute of programming languages. Their examples were taken from a range of languages, with an emphasis on Fortran and PL/I. They referred to the "work of E. W. Dijkstra and Harlan Mills on structured programming" as the source of their rules on "properly specifying flow of control", and the book stayed firmly within the territory of what Backus called the von Neumann languages. Kernighan and Plauger offered a "pragmatic and down-to-earth" discussion aimed at "improving current programming practice". Rather than discussing principles, they considered many examples culled from programming textbooks, and formulated 62 rather detailed "rules of programming style". Whereas Sammet and Black had been fairly

[5] Best known as an economist, in 1964 Black completed a thesis at MIT which "combined the fundamentals of logic with computer science" and spent a year at the consultancy Bolt, Beranek and Newman working on the "theory of handling information for libraries and hospitals of the twenty-first century" (Merton and Scholes 1995). Sammet (1969) may have been familiar with Black's article, as the book containing it is included in the list of references on Lisp on page 467.

non-prescriptive, Kernighan and Plauger made a series of very definite recommendations about what constituted good programming style.

A second edition of the book appeared in 1978, by which time Kernighan and Plauger felt that programming style had become a "legitimate topic of discussion". The new edition gave a greater prominence to "structured coding techniques that actually work well in practice", and at around the same time articles giving advice on how to use these techniques in established languages such as Fortran and Cobol were beginning to appear (see Sakoda 1974; Van Gelder 1977, for example). In parallel with the development of new "structured languages", some of these proposals found their way into revised versions and standards of existing languages.

This example illustrates a process by which a set of stylistic rules guiding programming *in* a language can turn into permanent syntactic features *of* a language. The notion of a programming *language style* can therefore be seen as parasitic on that of *programming style*: once a prescriptive rule has been proposed, it is but a short step to designing a language whose syntax will enforce that rule. This suggests a view of the evolution and development of programming languages that contrasts rather sharply with the idea that languages are based on explicit computational models.

It was in this context that Backus chose to use the word "style" to distinguish the two broad classes of language that he discussed in his Turing award speech. He did not explain why he chose this word: he had not used it earlier in theoretical work that defined a range of "language classes" (Backus 1973), but the popularity of Kernighan and Plauger's book, along with Sammet's earlier work, would certainly have given the term currency within the programming community.

The notion of a programming language style should be contrasted with that of a "paradigm", a term brought to prominence by Backus's successor as Turing award winner, Robert Floyd.[6] As is traditional with the term "paradigm", Floyd used it in a number of different senses. His first example of a paradigm, "the technique of structured programming", was a large-scale method of program design involving both the top-down decomposition of the problem at hand and the bottom-up synthesis of more detailed issues, such as the relevant abstract data types. But a paradigm could equally well be a very simple, low-level technique, such as the use of simultaneous assignment.

Unlike a style, which can naturally be thought of as a property of a language, Floyd's paradigms are language-independent design techniques. The relationship between the two is that a language can support a paradigm, for example by having explicit control structures that support structured programming, or a simultaneous assignment statement. This conforms to present-day usage. Computer scientist Peter van Roy (2009) defines a paradigm as "an approach to programming a computer based on a mathematical theory or a coherent set of principles". A paradigm can be supported by many languages, and equally, a language can support one or more paradigms. In this paper I will therefore continue to use Backus's term "style", which refers more directly to purely linguistic features.

[6] Floyd was the Turing award winner in 1978, and his lecture was published as Floyd (1979).

2 Complex Information Processes

Backus considered Lisp, or at least its "pure" subset, to be a functional programming language, a view endorsed by later writers including Hudak and Turner. As noted above, however, Lisp was originally classified as a list-processing language. A list in this context is a data structure in which a varying amount of information can be stored as a program runs and which allows dynamic relationships between data items to be easily defined and manipulated.

The use of lists can be traced back to the Logic Theory Machine (LT), a program developed by Allen Newell and Cliff Shaw of the RAND Institute, and Herbert Simon from the Carnegie Institute in Pittsburgh.[7] LT proved theorems in the propositional calculus, and lists were used to represent and store the structure of logical formulas and other data generated by the program. The program was running on RAND's Johnniac computer by the summer of 1956, when Newell and Simon presented it at the famous "Dartmouth research project", a summer school recognized as a foundational event in the history of artificial intelligence (McCorduck 2004; Boden 2006).

The curious description of the LT program as a "machine" is echoed in the language of the organizers of the Dartmouth meeting. They proposed an investigation into the mechanization of intelligence and learning, and programming had a rather subsidiary role:

> The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. … If a machine can do a job, then an automatic calculator can be programmed to simulate the machine.[8]

The detailed work proposals attached to the proposal support the sense that programming would not be central to the work of the Dartmouth meeting. It was not mentioned in proposals by Claude Shannon and Marvin Minsky, while John McCarthy planned to study the relation of language to intelligence, including the development of "an artificial language which a computer can be programmed to use on problems requiring conjecture and self-reference". The most concrete proposal was from IBM's Nathaniel Rochester who, motivated by frustrations with existing programming technique, planned an investigation into machine invention and discovery.

Newell and Simon were not involved in planning the Dartmouth meeting, and their approach was less theoretical. They first met in 1952 when Simon began a consultancy contract at RAND. He recalled that Newell was then engaged on an air

[7] Newell and Simon (1956b) give a comprehensive description of LT.

[8] McCarthy et al. (1955). McCorduck (2004), 119, notes that "[a]rtificial-intelligence workers continually use *machine* when they mean what an outsider would call a *program*". This usage is reminiscent of the equivalence that Turing noted between individual machines and their symbolic representations in a universal machine, but this may not its source. Early AI shared the cybernetic fascination with special-purpose gadgets, and was slow to fully internalize Turing's point about the universality of the computer. See Priestley (2011), 147–153, for more on this.

defence project, and "had this marvellous device … for simulating maps on old tabulating machines".[9] Simon and Newell struck up a productive collaboration, focusing on their shared interest in extending the use of computers beyond numerical applications. They became interested in chess, and by 1954 were talking about the possibility of putting a chess program on Johnniac.

In November 1954, Oliver Selfridge came to RAND to talk about some programming experiments in pattern recognition that he and Gerald Dinneen were carrying out at MIT's Lincoln Laboratories. The details of Selfridge and Dinneen's programs were a revelation to Newell, and by the end of the year, he had identified a class of "ultracomplicated problems" such as chess, "design problems, including programming a computer … [and] very complex information processing tasks like translating languages or abstracting scientific articles".[10] Computers could not be programmed to solve these problems, Newell felt, without adopting the strategies outlined by Selfridge and Dinneen.

Over the following year, Newell and Simon considered the prospects of programming a number of problems, including chess and theorem-proving in geometry. As they thought more about the details of programming ultracomplicated problems, they came to refer to them instead as *complex information processes*. LT was an example of such a process, but Newell and Simon didn't develop it out of an interest in theorem proving *per se*. Rather, it was, like chess, a small and tractable representative of the broader class of problems.[11]

Following Newell and Simon's presentation of LT at the Dartmouth meeting, a number of other researchers began writing programs to carry out a variety of complex tasks. Papers describing several of these were collected in the influential 1963 anthology *Computers and Thought*.[12] At this time, machine-building and neural net approaches was not seen to be delivering significant results, and the infant discipline of AI chose to constitute itself around the writing of programs, where a number of apparently innovative and substantial achievements could be found.

These programs were described as sharing a "heuristic" approach to programming. Newell and Shaw defined a heuristic as "a process that *may* solve a given problem, but offers no guarantee of doing so", and referred to "complex processes (heuristics) that are effective in problem solving".[13] Light can be thrown on these rather vague definitions by considering what Newell and Simon had previously had earlier said about *simple* processes: a simple process was one that could be precisely specified in advance. Simple processes can be found to solve difficult tasks, such as playing chess, but may be computationally infeasible to carry out.

[9] Simon's recollection is quoted in McCorduck (2004), 148. The historical narrative in the following paragraphs draws extensively on McCorduck's book.

[10] Newell (1954), 1. See Selfridge (1955) and Dinneen (1955) for details of their work on pattern recognition.

[11] Newell and Simon (1956a) introduces the term "complex information process" and gives the earliest description of LT.

[12] See Feigenbaum and Feldman (1963). For the historical background, see McCorduck (2004) and Boden (2006).

[13] Newell and Shaw (1957), 220, 218. The Oxford English Dictionary cites this paper as the earliest computer-oriented use of the term "heuristic".

> We shall find that we pay the price of complexity of process in order to carry out difficult tasks with limited computing power – i.e. limited in speed and memory. (Newell and Simon 1956a, 3–4)

However, complex processes proved a bit harder to define. After giving an impressionistic account of the thought processes of a human chess player, Newell and Simon observed:

> Our inability to [name all the different processes that go on] seems to be a major characteristic of complex processes – they consist of a great many subprocesses, each quite different from the others. It also seems characteristic that the complexity lies not in the component processes, but in the ways these are organized, so that the choice among them at each moment is highly conditional and flexible. (Newell and Simon 1956a, 5–6)

The force of these definitions comes from the fact that, rather than being an abstract notion, the idea of a simple process was firmly rooted in Newell and Simon's perception of contemporary programming practice. The authors of the Dartmouth proposal had expressed a sense that all was not well with current understanding of programming technique:

> [T]he major obstacle is not lack of machine capacity, but our inability to write programs taking full advantage of what we have. (McCarthy et al. 1955, 1)

Newell and Simon didn't just note the symptom, however, but offered a diagnosis:

> [O]ne of the sober facts about current computers is that, for all their power, they must be instructed in minute detail on everything they do. To many, this has seemed to be harsh reality and an irremovable limitation of automatic computing. It seems worthwhile to examine the necessity of the limitation of computers to easily specified tasks. (Newell and Simon 1956a, 1)

In his detailed work proposal for the Dartmouth meeting, Rochester had made similar observations:

> In writing a program for an automatic calculator, one ordinarily provides the machine with a set of rules to cover each contingency which may arise and confront the machine … one must sometimes go at problems in a very laborious manner whereas, if the machine had just a little intuition or could make reasonable guesses, the solution of the problem could be quite direct. (McCarthy et al. 1955, 10)

In Rochester, Newell and Simon's view, then, progress depended on the development of new programming techniques which would allow complex as well as simple processes to be coded. Before looking at the details of their proposals, however, it is useful to outline the characteristics of the existing languages and approaches that they believed suitable only for programming simple problems.

3 Programming as Planning

In 1947, John von Neumann and Herman Goldstine had published the first in a series of three reports entitled *Planning and Coding of Problems for an Electronic Computing Instrument*. These reports outlined a methodology for program development and included a compendium of example routines coded for the machine under development at the Institute of Advanced Study in Princeton. The reports are best known for introducing the flow diagram to a wide audience as a notational aid for designing programs.[14]

The reports' title highlights a key feature of the proposals, a division of labour between the two activities of planning and coding. "Planning" referred to a three-step "mathematical stage of preparations" preceding coding. The first step was to model the problem to be solved as a set of equations and conditions. These equations were then to be transformed into "arithmetical and explicit procedures" which would typically replace the "rigorous" mathematical definition of the problem with an "approximate" method of numerical solution. The third stage in planning was therefore to estimate the precision of the approximation process, by analysing the numerical properties of the chosen method of solution.

Once the planning stage was complete, coding could begin. Von Neumann and Goldstine's use of this term was much wider than its modern usage. The coding stage included the drawing of flow diagrams, the preparation of tables indicating how memory was to be allocated, and an number of rather more technical intermediate steps to be carried out before the actual program instructions were written down.

Von Neumann and Goldstine recognized that there was nothing in their description of the planning stage that was specific to automatic computing, and the details of their proposals reflected established practice in the organization of large-scale computing.[15] Historians have traced the origins of the division of labour in computation back to the work of Gaspard de Prony in the early nineteenth century (Grattan-Guinness 1990), and little changed in the 150 years that separated this from the depression-era Mathematical Tables Project:

> The operations of the Mathematical Tables Project were overseen by a planning committee, a group of six mathematicians who prepared the computing plans. … Each member of the committee would take responsibility for one computation, researching the background for the table, recommending a certain mathematical approach, preparing worksheets for the computers, and checking the final results. (Grier 2005, 212–3)

[14] See Goldstine and von Neumann (1947). In 1946, Haskell Curry and Willa Wyatt used a "Flow Chart" to illustrate the structure of an ENIAC program, but they did not use it in the development of the program and their notation differed "in principle" from von Neumann and Goldstine's (Curry 1949, 7).

[15] They wrote that "the first step has nothing to do with computing or machines … the second step has, at least, nothing to do with mechanization: It would be equally necessary if the problems were to be computed 'by hand' … [the third step] is necessary because of the computational character of the problem, rather than because of the use of a machine" (Goldstine and von Neumann 1947, 19).

Thanks to the changing meaning of the word "computer", this would serve almost verbatim as a summary of the procedures adopted by Harvard's Computation Laboratory in 1944 for preparing problems for the "Automatic Sequence Controlled Calculator" built for them by IBM:

> When a problem is referred to the Computation Laboratory, the first step in its solution is taken by the mathematician who chooses the numerical method best adapted to computation by the calculator. … Such functional, value and control tapes as are required are then computed, coded and punched. (Harvard 1946, 50)

Similar accounts of how to prepare problems for automatic computation were given by the developers of the other automatic computing machines of the early 1940s. These machines were not intended to revolutionize the way in which large-scale computations were organized. Within the overall pattern of a division of labour established for more than a century, automatic computers replaced a certain class of rather mechanical human labour but left the overall structure of the workflow involved in planning and carrying out calculations unchanged.

This continuity of practice should come as no surprise. All the developers of the automatic machines of the early 1940s had had extensive experience in carrying out or supervising large-scale manual computation. What was new was the need to be completely explicit about the operations that the machine would carry out. It turned out that the instructions given to a machine needed to be structured in rather different ways from those given to human computers, and the best ways to do this only emerged with experience. The fundamental notion was that of a sequence of instructions, each specifying a single arithmetical operation to be carried out. These sequences were then arranged in larger structures, such as loops to repeat the instructions in a sequence, and conditional branching to choose between alternative sequences or terminate a loop. This was the level of program structure that von Neumann and Goldstine's flow diagrams depicted and that the first machine codes could express. These codes also allowed instructions to be at least partially rewritten as a program was running, a feature which greatly increased the flexibility and usefulness of the new machines.

This historical survey provides the background against which to read Newell and Simon's mid-1950s comments about programming technique. The assumptions behind the organization of large-scale manual computation were precisely that all the steps in a calculation could be foreseen, planned and laid out in detail before work started. Newell and Simon's proposals for an alternative, more flexible approach to programming depended heavily on the use of subroutines, and to appreciate the novelty of their ideas it is useful first to briefly examine the role of subroutines in the "programming as planning" approach.

4 Subroutines

It is often necessary in large computations to carry out certain processes more than once, but at irregular intervals. To avoid having to repeatedly write out the instructions for such processes, the early automatic computers included diverse

mechanisms enabling subcomputations to be executed when required. For example, the Harvard (1946) machine had dedicated units, described as "electromechanical tables", to compute certain trigonometric and exponential functions, and an interpolation unit which read tabular data from tape and calculated intermediate values.

As this terminology suggests, these developments represented a translation of the use of mathematical tables in manual computation into the world of automatic machinery. Tables hold the precomputed values of commonly occurring functions, and when the value of such a function was required, a human computer would interrupt work on the main calculation sheet, take the appropriate volume of tables down off the shelf, look up the required value, and copy it into the appropriate place on the worksheet. Interpolation was used to obtain values for arguments that fell between those printed in the table.[16]

Some early computer designs allowed for the explicit storage of precomputed values but, as the Harvard machine's designers realized, there was an alternative. The speed of the new automatic computers meant that they could be programmed to compute function values when required. This idea was soon generalized so that any repeated sequence of instructions could be easily reused, not only those computing familiar mathematical functions. This gave programmers something new to do: they could look out for repetitions of code sequences and turn them into subroutines. In 1946, for example, mathematicians Haskell Curry and Willa Wyatt planned an interpolation routine for ENIAC, and recommended that programmers look for sequences of operations that could be isolated as repeatable "stages" in the computation (Curry and Wyatt 1946).

The provision and use of mathematical tables implies yet another division of labour, between the calculations carried out by the original compilers of a table and the later work of computers using the published table. This distinction was quickly institutionalized in the programming world in the form of "subroutine libraries". The Harvard programmers set up a "tape library" holding "such control tapes as are of general application" (Harvard 1946, 50), and in a January 1947 report on EDVAC programming Samuel Lubkin gave an example of a "standard subroutine" to compute square roots "in the form it would take in a library of subroutines". In March of that year, Jean Bartik's programming group was contracted by the US Army's Ballistic Research Laboratory to develop a set of exponential and trigonometric subroutines for ENIAC. The library concept and techniques for writing and using library subroutines were more widely disseminated in a 1951 programming textbook by Maurice Wilkes, David Wheeler and Stanley Gill which made, as its subtitle promised, "special reference to the EDSAC and the use of a library of subroutines".[17]

[16] For a comprehensive history of mathematical tables and their use, see Campbell-Kelly et al (2003).

[17] For the EDVAC subroutine, see Lubkin (1947), 20, 28. Lubkin's rather casual use of the term "library" suggests that it already had some currency within the EDVAC group. The work of Bartik's group is discussed in Bartik (2013) and Haigh et al (2016). Wilkes et al (1951) describe the EDSAC subroutine library in great detail, including what became known as the "Wheeler jump", an innovative and influential coding technique for transferring control between the main routine and what were termed "closed subroutines".

However, nobody writing about programming treated the identification of new subroutines, as opposed to the use of those already provided in a library, in any detail, and they were rather uncommon in practice. Of the 30 "stages" in Curry and Wyatt's interpolation program, only four were identified as being reusable. In the Monte Carlo programs run on ENIAC in 1948, there was only one subroutine (to compute a pseudo-random number) in approximately 800 program instructions. Von Neumann and Goldstine did not include subroutine identification as a step in their program planning process, and the flow diagram notation itself did not provide an obvious way represent a subroutine explicitly. Wilkes, Wheeler and Gill included a chapter on "library subroutines and their use in constructing programs", but gave few details on how to identify and write new subroutines.

Finally, the emphatic distinction made between "master routines" and subroutines meant that calling hierarchies were rather flat. Typically, a master routine would call a small number of subroutines, but it was rather rare for one subroutine to call another.[18] In manual computation, looking up a value in a table is an exception from the normal process of working through a computation sheet. This exceptionality reappears in the role that subroutines play in the programming as planning approach. The techniques widely used for subroutine call and return further meant that recursive calls, where a subroutine calls itself, were not possible.

None of these things are necessary features of subroutines, which soon came to be used much more ubiquitously and flexibly. The first steps in this development took place as part of an explicit attempt to address perceived shortcomings of the "programming as planning" approach.

5 Programming Complex Processes

Newell and Simon (1956a) thought it a problem that computers had to be "instructed in minute detail on everything they do". I have called this the "programming as planning" approach, and suggested that it is the result of the translation of established practices in manual computation into the new situation created by the automation of the work carried out by (human) computers. Newell and Simon argued that a new approach was necessary to program what they called "complex information processes".

Complex programs, in Newell and Simon's sense, should be distinguished from those that are merely large. In 1956, Herbert Benington gave a talk on the "production of large computer systems", drawing on his experience managing software development for the SAGE air-defence system.[19] Benington outlined a process which started with an "operation plan" and went through three levels of increasingly detailed specifications before reaching the stage of coding.

[18] In Wilkes et al (1951), some library subroutines, such as those to carry our integration, call "auxiliary subroutines" which represent the function being integrated. The three-level terminology of master, sub-, and auxiliary routines suggests a rather stereotyped and limiting approach, though of course the use of subroutines in actual EDSAC programming practice may have been more flexible.

[19] See Benington (1956). Background on the SAGE project can be found in Hughes (1998) and Benington (1983), which also reprints the text of the 1956 talk.

Documentation was described as "an immense, expensive task" and included the operational, program, and coding specifications as well as detailed flowcharts and coded program listings. The SAGE system was described as a "real-time control system" consisting of 100,000 instructions and 80 component subprograms, each dealing with a "logically independent subfunction" of the system. The subprograms operated on shared data tables, and were coordinated by a centralized control program. Benington commented that

> … only the most thorough testing of the entire program ensures that system threads have been carefully worked out, that incompatibilities are discovered, and that all contingencies are accounted for. (Benington 1983, 356)

Benington's paper illustrates how the "programming as planning" approach had evolved as the uses of computers had expanded. Despite the unprecedented size and complexity of the SAGE system, its development started from the assumption that the system's behaviour could be comprehensively specified in advance, and that the job of the control program was simply to ensure that these plans were correctly carried out. In direct contrast to this, Newell and Simon were interested in systems whose behaviour could not be specified in advance, and where the control program would have to exhibit a degree of flexibility.

Their approach, first used in the Logic Theory Machine (LT) discussed above, built on the earlier work of Selfridge (1955) and Dinneen (1955), who had proposed a two-level architecture to address the pattern recognition problem. Some simple routines carried out what we would now call image processing tasks, and a higher-level routine attempted to use these routines to classify the original images. The novelty was that the classification process could not be specified in advance. Rather, the top-level program employed heuristics in an attempt to *find* a plausible classification of the presented image.

Newell and Simon extended Selfridge and Dinneen's work in many ways, in particular by clearly defining the strategies and heuristics applicable to theorem proving and, later, more general problem solving, but the structure of LT preserved the two-level approach. The key features of "complex information processes" that they identified included the following (Newell and Simon 1956a, b):

1. They contain a large number of subprocesses. These subprocesses need not be complex themselves, but can be "simple and easily understood".
2. The subprocesses can be highly diverse. They need not be recognizable steps in a complete solution of the complex task and, individually, they are not "central or, usually, even necessary" to the overall task.
3. Complexity arises from the way in which the subprocesses are combined, the "pattern in which [they] operate". The application of the subprocesses is highly contingent, and will depend on information from the environment and also the outcomes of previous subprocess execution.
4. This overall progress of the computation is controlled by a "master process". The master routine makes use of heuristics: its course of action is not fixed in advance, but is supposed to respond dynamically to the evolving information

> available. "Large numbers of the processes have the function of determining the conditions under which other processes will operate."

LT was not written in machine code, a level of notation that Newell and Simon found "not at all suitable for human thinking or communication about a complex system." Instead, drawing upon current research on programming systems, they designed a pseudo-code which would allow them to express the features of complex information processes more naturally. In 1956, they called their pseudo-code the "logic language" (LL). They began by writing LT in LL and simulating its behaviour by hand, and then implemented LL to run on the Johnniac.

In some respects, LL resembled other pseudo-codes and languages of the mid-1950s, and in particular its exact contemporary, Fortran. Programs consisted of instructions which were executed in sequence except where a conditional branch instruction caused a deviation from the default sequence. Instead of the numbers and arrays required for numerical programming, however, LL supported the manipulation of lists holding the elements (such as logical connectives and variables) making up the expressions of the propositional calculus.

LL was radically different from Fortran, however, in the way it addressed one of the key features of complex information processes, namely the definition and flexible manipulation of large numbers of subprocesses. As Newell and Simon noted:

> Most current computing programs … call for the systematic use of a small number of relatively simple subroutines that are only slightly dependent on conditions. (Newell and Simon 1956b, 61)

This is the type of problem that Fortran was designed for. In keeping with its orientation towards mathematical applications, subroutines were referred to as "functions", and a small number of routines for evaluating some simple mathematical functions were built in to the system. Additional functions could not be defined in Fortran, but were coded separately and then added to the master tape.

> Library subroutines exist on the master FORTRAN tape in relocatable binary form. Placing a new subroutine on that tape involves (1) producing the routine in the form of relocatable binary cards, and (2) transcribing those cards on to the master tape by means of a program furnished for that purpose. (IBM 1956, 40)

Only with the arrival of Fortran II (IBM 1958) did the language permit the in-line definition of functions and subroutines. These could call each other freely, but recursive calls were not possible. By contrast, LL could express complex "organizations of processes that are often hierarchical, iterative and recursive".

LL did not make a strong distinction between instructions and subroutines. It was organized around the notion of an "information process" (IP), in effect a function or subroutine whose "inputs and outputs are comprised of symbolic information". LL defined a set of 44 elementary IPs, and new IPs could be defined in terms of these by

giving a routine description rather similar to a Fortran II function definition. Newell and Simon emphasized the potentially hierarchical nature of this process:

> [w]e can define a set of new IP's in terms of the elementary IP's, then another set of IP's in terms of both the elementary and defined IP's, and so on; thus creating a whole hierarchy of IP's and their corresponding routines. (Newell and Simon 1956b, 65)

There were very few restrictions on the extent to which one IP could call others. In particular, recursive calls where an IP called itself directly or indirectly were possible.

LL embodied a vision of program structure that was completely different from that of the "programming as planning" tradition that lay behind the Fortran proposals:

> A FORTRAN source program consists of a sequence of FORTRAN statements. (IBM 1956, 7)

Bearing in mind that Fortran functions were placed on the master tape, this definition implies that functions were not even considered part of a Fortran program, at least not until Fortran II. In contrast to this, an LL program is not thought of as a sequence of instructions with occasional calls to subroutines. Rather, the subroutine hierarchy is constitutive of the program, with only an executive routine to determine how subroutines are to be called:

> a program – e.g., LT – is a system of subroutines … organized in a roughly hierarchical fashion. There is a "master routine," each instruction of which is defined by another routine; the instructions in these subroutines, in turn, are defined by yet other subroutines, and so on. Eventually, primitive instructions are reached, and their defining subroutines, which are in machine language, are executed. (Newell and Shaw 1957, 234–5)

A program is a system of subroutines, and even primitive instructions are thought of as being defined by subroutines. LL thus turned the traditional program structure, where a main program consisting of a series of instructions occasionally calls subroutines from a library, inside out. Everything in LL was a subroutine, and at run-time the most promiscuous calling structures could arise.

> The number of levels in the main part of LT is about 10, ignoring some of the recursions which sometimes add another four or five levels. (Newell and Shaw 1957, 238)

This vision of program structure directly reflected—indeed, no doubt evolved in parallel with—Newell and Simon's analysis of the requirements of complex information processing. Some years later, Simon reflected more generally on the properties of complex systems, concluding that such systems are necessarily hierarchical, and it is very tempting to see in this a reflection of the technical subroutine hierarchies found in LT and similar systems.[20] Newell and Simon found the seeds of this approach to hierarchical organization in Selfridge and Dinneen's

[20] See Simon (1962). I thank one of the anonymous referees for drawing my attention to this paper.

work, but this earlier program was rather simple, consisting of a main routine which called a small number of simple, primarily computational, subroutines. The ubiquity of subroutines and the use of recursion were new in LL.

6 Towards a New Language Style

As Newell and Simon moved on from LT to the more ambitious General Problem Solver, LL evolved in parallel, in the process acquiring the more general name of "Information Processing Language" (IPL). LL itself became retrospectively known as IPL-I. The most mature version of the language, IPL-V, was by 1960 available on the IBM 650, 704 and 709.[21] But despite its distinctive philosophical approach and its various language innovations, IPL has never been seen as defining a new language style, nor as a functional programming language.

Instead, some of IPL's more significant features were incorporated into other languages. After the Dartmouth conference, Nathaniel Rochester suggested to IBM programmer Herbert Gelernter that he follow up suggestions made by Marvin Minsky and develop a geometry theorem proving program. Gelernter and his colleagues soon decided that their program would use lists to store data, and they planned to translate IPL from its Johnniac implementation to the IBM 704 computer. However, John McCarthy, at this point a consultant to the project, advised against this and instead suggested that they use Fortran as a base language.[22]

What resulted was FLPL, a "Fortran-compiled List-Processing Language", which added to Fortran a set of subroutines to enable the manipulation of lists.

> Our list-processing functions merely serve to increase the "vocabulary" of the language so that list manipulation processes may be described within the Fortran framework as are ordinary computer processes. We are thus able to enjoy the same ease of programming, ease of modification, and extensive debugging aids available to programmers of standard numerical problems. (Gelernter et al. 1960, 93)

As Gelernter pointed out, one of the attractions of this strategy was that Fortran was simply a more mature and better supported language than IPL. From a technical point of view, the major advantage of using Fortran instead of IPL was that it provided an "algebraic" notation.

> The algebraic structure of certain statements in this language corresponds closely to the structure of [a list], making possible the generation and manipulation of complex list expressions with a single statement. (Gelernter et al. 1960, 87)

[21] GPS was first described in Newell et al (1958). The history of IPL is sketched in Newell and Tonge (1960).

[22] See Gelernter et al. (1960, 88).

One of the goals of the Fortran project had been to allow programmers to use conventional mathematical notation in programs. As well as allowing programs to include expressions like X + Y, the language supported the notation of nested function calls. Thus the expression

> XCARF(XCDRF(LPTS)) → β1, the name of the second point on LPTS. (Gelernter et al. 1960, 94)

allows the two functions XCARF and XCDRF to be called in a single expression, with the result of the XCDRF function call being provided as the argument to XCARF. In IPL, in contrast, these two function calls would be written as two separate statements.

FLPL therefore represents an important step on the way to the functional language style, showing how the basic subroutines for list manipulation could fit with the algebraic notation for repeated function application that Fortran provided. However, other aspects of Fortran, in particular its lack of support for recursive subroutine calls, limited the capabilities of FLPL.

As well as consulting on the IBM project, McCarthy was involved in a number of projects of his own, several of which inspired him to develop new programming techniques.[23] The first of these was the conditional expression, "invented … in connection with a set of chess legal move routines" written in Fortran in 1957-8. McCarthy found Fortran's IF statement

> very awkward to use, and it was natural to invent a function XIF(M, N1, N2) whose value was N1 or N2 according to whether the expression M was zero or not. (McCarthy 1981, 176)

In the summer of 1958, McCarthy went back to IBM and worked on a program to differentiate algebraic expressions. Unlike the chess move routines, this program did use lists, and McCarthy used his new conditional expressions to define recursive functions. The program also required the ability to apply a function to every element in a list, and McCarthy defined a function that he called *maplist* to do this. *Maplist* had to be able to take a function as an argument, and it was at this point that the lambda calculus entered the scene.

> To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church (1941). I didn't understand the rest of the book, so I wasn't tempted to try to implement his more general mechanism for defining functions. Church used higher order functionals instead of using conditional expressions. Conditional expressions are much more readily implemented on computers. (McCarthy 1981, 176, citation in original)

McCarthy (1959) wrote up his ideas about conditional expressions and recursive functions as part of a proposal to include them in the emerging Algol language definition. As FLPL supported neither concept, however, he was unable to implement the differentiation program immediately. There wasn't any great

[23] The following paragraphs draw on the account given in McCarthy (1981).

enthusiasm on his or IBM's part to attempt further modifications of Fortran, and instead, when he moved to MIT after the summer, he began the implementation of a new language, Lisp. Far from being an implementation of the lambda-calculus, Lisp programs were initially expressed

> in an informal notation called M-expressions intended to resemble FORTRAN as much as possible. Besides FORTRAN-like assignment statements and **gotos**, the language allowed conditional expressions and the basic functions of LISP. Allowing recursive function definitions required no new notation from the function definitions allowed in FORTRAN I [sic] – only the removal of the restriction – as I recall, unstated in the FORTRAN manual – forbidding recursive definitions. (McCarthy 1981, 177)

Although the M-expressions were defined in McCarthy (1960), the first publication to mention Lisp, users of the language quickly adopted an alternative parenthesis-heavy notation that was easier to process mechanically. This paper is frequently cited as the original reference on Lisp, but its main purpose was to give a theoretical description of how lists could be defined recursively and it did not describe the complete Lisp language.[24]

Lisp, then, was the product of a complex historical process in which we can recognize at least the following steps.

1. The use of lists and recursive functions in Newell and Simon's LL and IPL languages.
2. The use of Fortran's algebraic notation in FLPL to ease the writing of complex sequences of list-processing subroutines.
3. McCarthy's development of the conditional expression.
4. His use of conditional expressions to define recursive functions.
5. His reformulation of the list data structure using concepts from recursive function theory.
6. His use of lambda notation for the very specific purpose of providing functional arguments to functions.

Many different motivations are visible here. Newell and Simon had developed a style of programming that used lists and recursion as a solution to the problem of programming complex problems. In contrast, FLPL's use of algebraic expressions and McCarthy's preference for conditional expressions seem to reflect rather more personal stylistic preferences. Implementation issues play an important role, and Fortran in particular was simultaneously an inspiration and an obstacle. Finally, theoretical results from logic did contribute, but as a resource to be drawn upon in a rather piecemeal fashion rather than something which straightforwardly provided a framework for the new language.

[24] See Priestley (2011, 220–223) for more on this. In this connection, it is interesting to note the comment made by Turner (2012): "The theoretical model behind LISP was Kleene's theory of first order recursive functions". He added in a footnote: "McCarthy made these statements, or very similar ones, in a contribution from the floor at the 1982 ACM symposium on Lisp and functional programming in Pittsburgh. No written version of this exists, as far as [I] know".

7 Conclusions

The starting point of this paper was Backus's distinction between the von Neumann and functional styles of programming language, and the account he gave of the origins of the two styles. Backus suggested that a programming language style reflects an underlying computational model: in the case of the von Neumann style the underlying model is physical, but in the case of the functional style it is mathematical.

This account might be taken to suggest that, historically, programming languages and language styles emerge from underlying computational models through processes of derivation, evolution, and articulation. The papers by Hudak (1989) and Turner (2012) suggest that this may indeed be how the community of functional programming researchers understand the history of that style, but Backus's own historiography was rather more complex: in 1978, as well as articulating the view that languages derive from computational models, he gave an account of the origins of Fortran that did not at all present it as "basically [a] high level, complex version […] of the von Neumann computer".

The account given in this paper of the historical origins of the two styles suggests that they have their roots not in differing computational models, but in different visions of the programming process. I have argued that the von Neumann, or "programming as planning", style can best be seen as a translation into the age of automatically sequenced machines of existing practices of planning large-scale manual computations.[25] The origins of the functional style, however, can be traced back to Newell and Simon's realization in 1954 that programming a certain class of complex programs would require a radically different approach to design and program structure.

At the level of detailed program structure, one of the most striking differences between the two approaches is in their use of subroutines. The "programming as planning" approach established, or reinforced, the belief that problems could be specified, ideally mathematically, before being coded. The tension implicit between this approach and the desirable property of code reuse was accommodated by the increasing use of extensive subroutine libraries. Subroutines remained exceptional, however, interruptions to the "main flow" of the program. Languages placed limits on the depth of calling hierarchies, and the issue of recursive subroutine calls remained controversial.

In contrast, Newell, Shaw and Simon proposed a programming style that made extensive use of user-defined and problem-specific subroutines, and allowed maximum flexibility in the way they could be combined. They implemented this style in the series of IPL languages, but despite their innovative features, these languages were in many ways conventionally structured, and have never featured prominently in the genealogy of functional programming languages.

[25] Rather than following Backus in seeing the von Neumann languages as reflections of the underlying computer, it might be more accurate to describe the "von Neumann architecture" as having been developed to support a particular way of structuring and expressing computations.

Section 6 outlined the process whereby Lisp, sometimes described as the "first functional programming language", was developed in the context of practical programming in artificial intelligence in the late 1950s, a context that owed much to the Newell and Simon's pioneering achievements, in particular the Logic Theory Machine. Throughout this period, McCarthy does not appear to have been inspired to create a language based on a single theoretical foundation. His work is rather an extended passage of *bricolage*, with important ideas being taken from sources as diverse as IPL, Fortran, and recursive function theory (with a little bit of notation borrowed from the lambda calculus), all the while being informed by his desire to find the best way to express solutions to a variety of programming problems.

Gelernter and McCarthy's work in these years is a clear example of the process described in Sect. 1, whereby individual stylistic choices—such as the decision to use Fortran's algebraic expressions to express complex list-processing functions rather than IPL's step-by-step approach—later become constitutive elements of a new language. As Lisp historian Herbert Stoyan put it,

> it was McCarthy who, as the first, seemed to have developed the idea of using functional terms (in the form of "function calls" or "subroutine calls") for every partial step of a program. This idea emerged more as a stylistic decision, proved to be sound and became the basis for a proper way of programming – functional programming. (Stoyan 1984, 299)

Similarly, McCarthy's development of conditional expressions began as a personal choice based on the fact that he didn't like the form of Fortran's IF statement, but they later became the fundamental way of expressing conditional execution in Lisp.

As a result of these complex origins, Lisp was, as Backus noted, far from being simply a functional language, and in fact it allowed programmers to work in either the conventional or the applicative style. Interestingly, the choice between these alternatives was very closely related to the different Lisp programming styles identified by Black (1964), and in particular to the decision on whether or not to use the language's so-called "program feature". Lisp's heterogeneity should not be attributed to a failure on McCarthy's part to implement a more thorough-going theoretical programme, however; as Stoyan put it,

> around the end of the fifties, nobody, including McCarthy himself, seriously based his programming on the concept of mathematical function. It is quite certain that McCarthy for a long time associated programming with the design of stepwise executed "algorithms". (Stoyan 1984, 299).

As functional programming researchers have recognized in their historical reflections, then, Lisp does not fit well into the view that the development of functional programming consists in ever more refined articulations of the lambda calculus. It falls outside the scope of this paper to consider the process by which the "functional programing paradigm" was subsequently articulated around the idea of the lambda calculus, and came to adopt Lisp as its figurehead, or more generally how programming language paradigms become individuated. It is certainly a plausible hypothesis that the identification of a suitable abstract model of

computation plays an important role in this process, but the lesson that this paper draws from history is rather that the role of programming problems and practices in the emergence of novel programming languages and language styles should not be underestimated.

Acknowledgements This paper originated in talks delivered at the CiE 2012 conference in Cambridge and the "Jornadas sobre Inteligencia Artificial y sociedad comtemporánea: El cometido de la información" at the University of A Coruña, in Ferrol, in March 2016. Thanks to John Tucker and Wenceslao J. Gonzalez for invitations to deliver these talks. A preliminary draft was discussed at a workshop on the Early Digital, held at the University of Siegen in 2016, and I would like to thank the workshop participants, and also David Nofre and the two anonymous referees, for many helpful comments and conversations.

References

Backus, J. (1973). Programming language semantics and closed applicative languages. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on principles of programming languages* (pp. 71–86).

Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM, 21*(8), 613–641.

Backus, J. (1981). The history of FORTRAN I, II, and III. *In Wexelblat, 1981,* 25–45.

Bartik, J. J. (2013). *Pioneer programmer*. Kirksville: Truman State University Press.

Benington, H. D. (1956). Production of large computer programs. In *Symposium on advanced programming methods for digital computers*, ONR Symposium Report ACR-15, 15–28. (Office of Naval Research, Washington, DC, June 28-9, 1956.)

Benington, H. D. (1983). Production of large computer programs. *Annals of the History of Computing, 5*(4), 350–361.

Black, F. (1964). Styles of programming in LISP. In E. C. Berkeley & D. G. Bobrow (Eds.), *The programming language LISP: Its operation and applications* (pp. 96–107). Cambridge: MIT Press.

Boden, M. A. (2006). *Mind as machine: A history of cognitive science* (Vol. 2). Oxford: Oxford University Press.

Campbell-Kelly, M., Croarken, M., Flood, R., & Robson, E. (2003). *The history of mathematical tables: From Sumer to spreadsheets*. Oxford: Oxford University Press.

Church, A. (1941). *The calculi of lambda-conversion*. (Annals of Mathematics Series, Number 6, Princeton: Princeton University Press.)

Curry, H. B. (1949). *On the composition of programs for automatic computing*. (Naval Ordnance Laboratory Memorandum 9805, 26 January 1949).

Curry, H. B., & Wyatt, W. A. (1946). *A study of inverse interpolation of the Eniac*. (Ballistic Research Laboratory, Aberdeen Proving Ground, MD. Report No. 615. 19 August, 1946.)

Dinneen, G. P. (1955). Programming pattern recognition. In *Proceedings of the Western Joint Computer Conference* (pp. 94–100).

Feigenbaum, E. A., & Feldman, J. (Eds.). (1963). *Computers and thought*. New York: McGraw-Hill.

Floyd, R. W. (1979). The paradigms of programming. *Communications of the ACM, 22*(8), 455–460.

Gelernter, H., Hansen, J. R., & Gerberich, C. L. (1960). A Fortran-compiled list-processing language. *Journal of the ACM, 7*(2), 87–101.

Goldstine, H. H., von Neumann, J. (1947). *Planning and coding of problems for an electronic computing instrument*. Part II, Volume 1. (Institute of Advanced Study, NJ. 1 April, 1947.)

Grattan-Guinness, I. (1990). Work for the hairdressers: The production of de Prony's logarithmic and trigonometric tables. *Annals of the History of Computing, 12*(3), 177–185.

Grier, D. A. (2005). *When computers were human*. Princeton: Princeton University Press.

Haigh, T., Priestley, M., & Rope, C. (2016). *ENIAC in action: Making and remaking the modern computer*. Cambridge: MIT Press.

Harvard. (1946). *A manual of operation for the Automatic Sequence Controlled Calculator*. Cambridge: Harvard University Press.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys, 21*(3), 359–411.

Hughes, T. P. (1998). *Rescuing prometheus*. New York: Pantheon.
IBM. (1956). *The FORTRAN automatic coding system for the IBM 704 EDPM: Programmer's reference manual*. (Applied Science Division and Programming Research Dept., IBM: New York, NY, October 15, 1956.)
IBM. (1958). *Reference manual: FORTRAN II for the IBM 704 data processing system*. New York: IBM.
Kernighan, B., & Plauger, P. J. (1974). *The elements of programming style*. Reading: Addison-Wesley, Second edition: 1978.
Lubkin, S. (1947). *Proposed programming for the EDVAC*. (Moore School of Electrical Engineering, Office of the Director Records, 1931–1948, UPD 8.4, University of Pennsylvania Archives and Records, box 8.)
McCarthy, J. (1959). Letter to the editor. *Communications of the ACM, 2*(8), 2–3.
McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine: Part I. *Communications of the ACM, 3*(4), 184–195.
McCarthy, J. (1981). History of LISP. *In Wexelblat, 1981,* 173–194.
McCarthy, J., Minsky, M. L., Rochester, N., & Shannon C. E. (1955) *A proposal for the Dartmouth summer research project on artificial intelligence*. (Dartmouth College, 31 August, 1955.)
McCorduck, P. (2004). *Machines who think*. (A. K. Peters, Natick, MA.)
Merton, R. C., & Scholes, M. S. (1995). Fischer black. *The Journal of Finance, 50*(5), 1359–1370.
Newell, A. (1954). *The chess machine: An example of dealing with a complex task by adaptation*. (RAND Corporation, report P-620, 28 December, 1954.)
Newell, A., & Shaw, J. C. (1957). Programming the logic theory machine. In *Proceedings of the western joint computer conference: Techniques for reliability* (pp. 230–240).
Newell, A., Shaw, J. C., & Simon, H. A. (1958). Report on a general problem-solving program. (RAND Corporation, report P-1584, 30 December, 1958. Revised 9 Feb 1959.)
Newell, A., & Simon, H. A. (1956a). *Current developments in complex information processing*. (RAND Corporation, report P-850, 1 May, 1956.)
Newell, A., & Simon, H. A. (1956b). The Logic Theory Machine: A complex information processing system. *IRE Transactions on Information Theory, 2*(3), 61–79.
Newell, A., & Tonge, F. M. (1960). An introduction to Information Processing Language V. *Communications of the ACM, 3*(4), 205–211.
Priestley, M. (2008). *Logic and the development of programming languages, 1930–1975*. (PhD thesis, University of London).
Priestley, M. (2011). *A science of operations: Machines, logic and the invention of programming*. London: Springer.
Rosen, S. (1967). *Programming systems and languages*. New York: McGraw-Hill.
Sakoda, J. M. (1974). Structured programming in FORTRAN. *ACM SIGSOC Bulletin, 6*(1), 12–16.
Sammet, J. (1969). *Programming languages: History and fundamentals*. Upper Saddle River: Prentice-Hall.
Sammet, J. (1971). Application of extensible languages to specialized application languages. *ACM SIGPLAN Notices, 6*(12), 141–143.
Sammet, J (1972). An overview of programming languages for specialized application areas. In *AFIPS '72 (Spring) Proceedings of the May 16–18, 1972, Spring joint computer conference* (pp. 299–311).
Selfridge, O. G. (1955). Pattern recognition and modern computers. In *Proceedings of the western joint computer conference* (pp. 91–93).
Simon, H. A. (1962). The architecture of complexity. *Proceedings of the American Philosophical Society, 106*(6), 467–482.
Stoyan, H. (1984). Early LISP History (1956–1959). In *Proceedings of the 1984 ACM symposium on LISP and functional programming* (pp. 299–310).
Turner, D. A. (2012). Some history of functional programming languages. In *13th International symposium on trends in functional programming*. (Paper available at https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf. Accessed February 28, 2017.)
Van Gelder, A. (1977). Structured programming in Cobol: An approach for application programmers. *Communications of the ACM, 20*(1), 2–12.
Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. (Paper available at https://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf. Accessed February 28, 2017.)
Wexelblat, R. L. (Ed.). (1981). *History of programming languages*. New York: Academic Press.
Wilkes, M. V., Wheeler, D. J., & Gill, S. (1951). *The preparation of programs for an electronic digital computer*. Reading: Addison-Wesley.