STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

MR 91

Problems in the theory of programming languages

by

J.W. de Bakker

Lecture given at the 1967 International Congress
for Logic, Methodology and Philosophy of Science

Amsterdam, August 29, 1967

The Mathematical Centre at Amsterdam, founded the 11th of February, 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications, and is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) and the Central Organization for Applied Scientific Research in the Netherlands (T.N.O.), by the Municipality of Amsterdam and by several industries.

1. INTRODUCTION

The theory of programming languages is usually divided into three parts (see e.g. Zemanek [46]):

a. Syntax.

It is investigated which formal systems can be used for the definition of grammars of programming languages. A grammar is a set of rules that defines which sequences of symbols over a given alphabet form a program in the language concerned. Two important requirements which should be fulfilled by such a system are: It should be powerful enough to allow formal expression of all syntactical rules, and it should define the structure of a program in such a way that efficient translation is possible.

b. Semantics.

Problems are investigated which deal with the meaning of programs. The ultimate goal is the development of a theory that leads to a formal definition of the semantics of programming languages and that can provide an answer to questions such as: "Are two given programs equivalent?", or "Is a compiler for a certain language correct?", or "Does a given program solve a certain problem?".

c. Pragmatics.

Here the object of study is the relation between the language and its user. Hence, the important question in this area is: "Which concepts should be included in a language to allow the programmer efficient, compact and elegant formulation of his problem?".

It is clear that for practical purposes, pragmatic problems are the most important. Consequently, most of the efforts in programming languages have been spent in this field. However, as far as we know, no theory of pragmatics has been developed as yet. Theoretical considerations have up to now mainly been concerned with syntax. We mention only: the theory of context free languages with their various specializations and generalizations, the production language of Floyd, and the syntax-directed compilers. In our talk, we shall not deal with these investigations but shall restrict ourselves to semantic problems and shall try to give an impression of the work that has been done in this field.

2. SEMANTICS AND THE GENERAL THEORY OF COMPUTATION

For the development of semantic theories about programming languages, it is clearly desirable to have available a "general theory of computation" which can provide a background or framework to which semantics can be related. However, such a general theory of computation is only in a rudimentary stage. There are several ways of approaching such a theory. A survey of the situation, as it existed several years ago, has been given by McCarthy [26]. In our opinion, no decisive progress has been made since then. We shall now discuss a few approaches in somewhat more detail.

a. The theory of computability, i.e. the theory of Turing machines, recursive functions etc. It was already said by McCarthy that this theory has as yet only resulted in the statement of the essential limits which are imposed upon a theory of computation. Its relevance for a theory of algorithmic processes, as they occur in the practical use of computers, is very limited. However, it should be mentioned that in the past few years, research has started into real-time aspects of Turing machines, i.e., investigations which take into account the time factor, e.g. expressed by the number of operations that are required for a certain calculation. This new branch of the theory of Turing machines might eventually lead to results which are of interest for the theory of computation.
Among the many formalisms that have been proposed for studies of computability, and that have all been proved to be equivalent, there is one system that we want to mention separately, namely the theory of "graphschemata". It was proved by Rosza Peter [33] that these graphschemata are equivalent to recursive functions. However, it is probable that the formalism of graphschemata shows the closest connection to the methods that are used in practice for the description of computer algorithms. This follows from the fact that graphschemata are nothing but flow diagrams obeying certain restrictions. Investigations in this area have been reported by Kaluzhnin [18] and Thiele [39]. Related is the work of Böhm and Jacopini [5], who exhibit a number of components, from which, in a sense, each flow diagram can be made up (they need some extra formalism, for which we refer to their paper).

b. Automata theory. Here the situation is the same as above. Although automata theory has led to many results of mathematical interest, again no generally accepted system, directly useful for a theory of computation, has come forward. We think that the following quotation from Hao Wang [40] is still valid:

"Although there are various elegant formulations of Turing machines, they are still radically different from existing computers. To approach the latter, we should use fixed word lengths, random access addresses, accumulator, and permit internal modifications of the programs. Alternatively, we could, for example, modify computers to allow more flexibility in word lengths. Too much energy has been spent on oversimplified models, so that a theory of machines and a theory of computation which have extensive practical applications have not been born yet".

We shall give here a few examples of several automaton-like models that have been proposed in the past few years. No attempt is made at completeness, but we wish to give only an impression of the great variety that exists in this field:

b1. One of the first proposals was made by Kaphengst in his paper: "Eine Abstrakte Programmgesteuerte Rechenmachine" [19]. This paper introduces concepts such as register, instruction and instruction counter, etc., in an abstract machine which is then proved to be equivalent to recursive functions.

b2. A paper by Raymond: "Etude générale des structures de calculatrices à prefixes et à piles" [35]. Emphasis is laid here upon a study of the memory of a computer.

b3. A paper by De Backer and Verbeek: "Study of Analog, Digital and Hybrid Computers, using automata theory" [1]. In this article the notion of error in a computation plays an important role.

b4. "A theory of computer instructions", by Maurer [24]. This paper covers many aspects of existing computers: It treats the notions of memory, registers, input/output, and instructions. It appears to be an interesting contribution to a theory of computing that is more concerned with hardware aspects.

b5. The stack automata, as introduced by Ginsburg, Greibach and Harrison [14]. Here the purpose is to simulate techniques which are used in the translation of programming languages.

b6. The theory of "Random Access, Stored Program Machines", as introduced by Elgot and Robinson [9,10]. We shall return to this later, since it has played a role in the formal definition of PL/I.

c. McCarthy's mathematical theory of computation [25,26,27].
This theory is not directly related to either the theory of computability or to automata theory. McCarthy's papers "A basis for a mathematical theory of computation", "Towards a mathematical Theory of computation", and "Problems in the theory of computation", have become well known and have influenced work on the semantics of programming languages, as we shall see below.

d. Proofs about programs.
We shall make here some remarks on investigations, related to theories of computation, which are in some way concerned with proofs about programs. First of all, it is obvious that a theory intended to lead to proofs about programs, will be limited by unsolvability results from logic. We mention only the classic example concerning the impossibility of an algorithm which decides for each arbitrary program whether it will get into an infinite loop. Another difficulty that arises when one wants to develop a theory that can prove the correctness of a program, is the following:
Suppose that one wishes to prove that a given program P, written in some programming language, gives a correct description of a certain process Q. This problem only makes sense if Q can be precisely stated by means of some other formalism, e.g. some part of mathematics. Often, however, the only precise way of stating process Q is by exhibiting some program that describes it. Clearly, in these cases a proof of the correctness of this description will be very difficult or even impossible.
We now mention a few investigations that deal with proofs about programs:

d1. Well known is the work of Yanov [45], who introduced the "logical schemes of algorithms" and derived several equivalence results about them.

d2. Less well known is the work that has been done by Igarashi, namely his papers "An axiomatic approach to the equivalence problems of algorithms with applications" [16], and "A formalization of languages and the related problems in a Gentzen-type formal system" [17]. See also [15].

d3. McCarthy [25] has used his technique of recursion induction for some proofs on Algolic (i.e., written in a small subset of ALGOL 60) programs. Later on, we shall mention another type of proof due to him.

d4. Naur [31] has proposed a method to be used for the proof of algorithms, by the technique of what he calls "general snapshots", i.e., expressions of static conditions existing whenever the execution of the algorithm reaches particular points.

d5. In his Ph.D. thesis [11], Evans has proved the correctness of two translation algorithms. Some references to other work in this area which we found in his paper, are: Cooper [7] and London [23].

3. SEMANTIC DEFINITION OF PROGRAMMING LANGUAGES

After having tried to give an impression of the background which is available for a theory of semantics, we shall now deal with one of the main goals of a semantic theory, namely the development of a system for the formal definition of programming languages. We first state some reasons for such a formal definition:

a. First of all, the wish to provide the compiler-writer with a complete, precise and unambiguous definition of the language which he has to translate. Such a definition should e.g. make it clear which parts of the language are not fully specified, so that the compiler-writer knows where he has to give his own interpretation. Experience has shown that it is almost impossible to avoid ambiguities in the definition of a programming language by means of a natural language, such as English.

b. One might require of a formal definition that it can be used as a basis for the development of a compiler. The formal definition should then be designed in such a way that it reflects in some sense the structure of the compiler. It should be remarked that it is often difficult to combine requirements a and b.

c. Recently, suggestions have been made for the introduction of programming languages which allow the programmer to include modifications or extensions of the language in his program. It is clear that it is necessary in such a situation to provide the programmer with a formalism in which he can state these modifications to the language.

d. Finally, a formal definition of a programming language should provide insight into theoretical properties of this language. It should lead to a vocabulary which can be used for discussions about the language. One might expect of such theoretical investigations e.g. the detection of incompatible, contradictory of ambiguous concepts or constructions in the language. It might also be used as a source of inspiration for new useful concepts, which would not have originated directly from practical considerations.

We shall now discuss some systems which have been proposed for the formal definition of programming languages. In will appear that the situation is the same as with the theory of computation; i.e., almost every author has his own system; there is as yet no generally accepted method, nor any indication of a convergence in opinion towards such a method. In September 1964, a conference on "Formal Language Description Languages" was held, organized by the technical committee on programming languages of the International Federation for Information Processing. The proceedings of this conference [36] show clearly how much the ideas of the several authors diverge.

First of all we mention the methods that are based upon the λ-calculus. Landin [20,21,22] is the main representative of this group. Böhm [3,4] uses both the λ-calculus and the combinatory logic of Curry. He calls his system CUCH, derived from CUrry and CHurch. The λ-calculus also plays an important role in the work of Strachey [38]. It appears that the λ-calculus allows an elegant definition of the locality concept;

the definition of assignment statements and goto statements causes more difficulties.

Well known is the state vector approach of McCarthy [28]. In principle, the components of the state vector are: the current values of the variables that occur in the program, and the number of the statement which is to be executed. The semantics of a program is defined by a recursive function that describes how the state vector changes as a result of the statements that occur in the program. McCarthy admits that the structure of the state vector will have to become more complicated if recursion occurs in the program. Also, the meaning of e.g. declarations and procedures cannot be defined directly in terms of this state vector.

McCarthy has applied his formalism also to give a proof of the correctness of a simple compiler for arithmetic expression, [29].

Again, however, he says that in order to apply the technique to proofs concerning the correctness of translation of e.g. sequences of assignment statements or goto statements, "a complete revision of the formalism will be required".

Wirth [42] lets the semantic description of a programming language run parallel to its syntactic definition. Whenever a syntactic rule is applied during the analysis of a program, a corresponding semantic rule is applied which changes the values of zero or more entities in a so-called environment. The semantic rules are formalized in a language which is said to correspond closely to the elementary operations of a computer. It is assumed that the concepts of this elementary language do not need further formal definition. He demonstrates his system by means of a formal definition of the programming language EULER, based upon a generalization of ALGOL 60.

Feldman [12] has introduced a "Formal Semantic Language", which he has designed for the purpose of constructing compilers. For these practical purposes, FSL has proven to be of much use. However, we feel that FSL is too complicated a language to be considered a solution to the problem of the formalization of semantics.

Finally, we mention some systems which give only some principles for semantic description, from which it is not yet possible to form an opinion as to their applicability to a complete formal definition of a programming language: the papers of Steel [37], Garwick [13], and Nivat and Nolin [32].

Complete formal definitions have been given of PL/I [34] and of ALGOL 60 [2]. We shall return to the definition of ALGOL 60 below. The definition of PL/I is due to a group at the IBM Laboratory in Vienna. We quote from the introduction to their report:
"The method adopted is based on the definition of an abstract machine which is characterized by the set of its states and its state transition function. A PL/I program defines an initial state of the machine, and the subsequent behaviour of the machine is said to define the interpretation of the PL/I program ...
The basis for the development of the method are the publications of McCarthy, Landin and Elgot. Especially, the notions of instruction and computation are similar to those given by Elgot. The notion of Abstract syntax is due to McCarthy".

For completeness sake, we mention the announcement of a paper by Christensen and Mitchell [16], which will give a partly formalized definition of NICOL II, a version of PL/I.

4. A FORMAL DEFINITION OF ALGOL 60

In our thesis [2], we have investigated a method for the formal definition of programming languages, and applied this method to a complete formal definition of ALGOL 60. The system is based upon two papers by van Wijngaarden [43,44]. We give here only a sketch of its principles; for details we refer to our paper. The method consists essentially of a combination of Markov algorithms and context free grammars. The definition of a language is given by means of a list of rules, which are either of syntactical nature, in which case they have the form of a production rule of a context free grammar, or of semantical nature.

Then they have the structure of a substitution rule, as used in Markov algorithms. In these substitution rules, use is made of the metalinguistic variables, as defined in the syntactical rules.(A combination of syntactical and semantical elements in one rule is also possible; we shall not treat this feature here.)
As an example, we exhibit the definition of the greatest common divisor of two integers, written in "unary" notation, by means of the Euclidean algorithm.

<integer>::= 1 | <integer> 1
(<integer1>,<integer1>) → <integer1>
(<integer1><integer2>,<integer1>) → (<integer1>,<integer2>)
(<integer1>,<integer1><integer2>) → (<integer1>,<integer2>)

Note the occurrence of so-called "indices" within the metalinguistic variables. The function of these indices is the following: If, in a certain rule, one of its possible productions is substituted for an indexed metalinguistic variable, then the same substitutions must be made in all places in this rule where this metalinguistic variable occurs with the same index.
An abstract machine is introduced, called the processor, which applies the rules described above, to an input sequence (in the example given above, the processor might be asked to evaluate e.g. (111,11). When the processor has to establish whether a substitution rule is applicable to an input sequence, it uses a well-defined parsing scheme. Details of the way parsing is performed are omitted here.
A further important property of the system is the following: Whenever the value of a certain input sequence has been determined, this value is added - in the form of a new substitution rule - to the already existing list of rules. Consequently, the list of rules is continuously growing, according as more input sequences are evaluated. This last feature, i.e. the growing of the list of substitution rules, is essential for the definition of a programming language such as ALGOL 60. The definition of ALGOL 60, as given in [2], consists of a list of about 800 rules, of syntactical, semantical (or mixed) type. If the processor evaluates an ALGOL 60 program, this is performed essentially by successive evaluation of the declarations and statements that constitute the

program concerned. E.g. evaluation of the assignment statement a:= 3, will lead to the extension of the already existing list of rules with the substitution rule a → 3. We cannot deal here with the way in which declarations, procedures, goto statements etc. are treated. Their treatment is explained extensively in our paper. We now give a summary of its contents: First a detailed description is given of the system of which we have sketched some principles above. Next we investigate some theoretical properties of the system, namely its relation to the theory of computability, and to a few aspects of the theory of phrase structure languages. The processor is defined by means of an ALGOL 60 program, and this program is demonstrated by a large number of examples. Then follows the definition of ALGOL 60, by means of about 800 rules, and a commentary upon this definition.

Our system has proved capable of giving a complete formal definition of ALGOL 60, from the definition of integer arithmetic to the definition of e.g. the procedure concept. However, it cannot be used directly as a basis for a compiler for the language.

5. CONCLUSIONS

From the research which has been performed up to now in the semantics of programming languages, it can be concluded that, for the treatment of the more difficult concepts, present-day mathematics is only of limited use. It appears that concepts, as nowadays current in programming languages, often have no direct counterparts in mathematics. We give a few examples: One would expect that a simple concept such as the arithmetic expression, would be clear to everyone who knows some high-school algebra. However, already in this simple case anomalies are caused by the possibility of side effects in a language such as ALGOL 60, so that e.g. a+b is not necessarily equal to b+a. More difficult is the concept of locality and the related problems of storage allocation. Although the locality concept is related to the idea of bound variables, this does not help much if one wants to investigate concepts like own dynamic arrays. The name-value relation in its simplest form is known in logic. However, the general reference structure, as present in the proposal for ALGOL 67, is again, as far as we know, without a direct counterpart. Simple data structures, such as vectors, matrices or rectangular arrays in general, or trees, are well known. This does not hold for more complicated structures, such as the records proposed by Hoare [41]. Function designators are at first sight nothing but functions, as known in mathematics. However, a mathematician will not be confronted with the question: "What happens to the value of the function if a jump to a point outside is performed?". We know of no concept in mathematics that can be related to goto statements. We might remark here that a complete formal definition of the meaning of goto statements, at least in our system and in several others as well, is one of the most difficult tasks. Some authors consider the goto statement as a relic from the days of machine coding, and propose to abolish it (McKeeman [30]) or at least to diminish its use (Dijkstra [8]). Finally we mention the notion of parallel processing, which has hardly been investigated at all in computability and automata theory.

McCarthy once expressed the hope that mathematical logic will be as fruitful for the science of computation as analysis has been for physics. We hope to have given an impression of the results which have been obtained in this direction and of the many open problems which still remain to be studied.

REFERENCES

1. W. de Backer and L. Verbeek — Study of Analog, Digital and Hybrid Computers using Automata Theory. I.C.C. Bulletin, 1966, vol. 5, pp.215-245.

2. J.W. de Bakker — Formal definition of programming languages, with an application to the definition of ALGOL 60. Mathematical Centre Tracts 16, Amsterdam, Mathematisch Centrum, 1967.

3. C. Böhm — The CUCH as a formal and description language. [36], pp. 179-197.

4. C. Böhm — Introduction to CUCH. Automata Theory (Ed. E.R. Caianiello). New York, Academic Press, 1966.

5. C. Böhm and G. Jacopini — Flow Diagrams, Turing Machines and Languages with only two Formation Rules. Comm. ACM, vol. 9, 1966, pp. 366-372.

6. C. Christensen and R.W. Mitchell — Reference Manual for the NICOL II programming language. (to appear as a report of Computer Associates, Wakefield, U.S.A.).

7. D.C. Cooper — The equivalence of certain computations. Computation Center, Carnegie Institute of Technology, 1965.

8. E.W. Dijkstra — Programming considered as a human activity. Proc. IFIP Congress 1965, vol. 1 (Ed. A. Kalenich). Washington, Spartan Books, 1965, pp. 213-219.

9. C.C. Elgot — Machine species and their computation languages. [36], pp. 160-179.

10. C.C. Elgot and A. Robinson — Random-access, stored program machines, an approach to programming languages. J. ACM, 1964, vol. 11, pp. 365-399.

11. A. Evans, Jr. — Syntax Analysis by a Production Language. Ph.D. thesis, Carnegie Institute of Technology, 1965.

12. J. Feldman — A formal semantics for computer languages and its application in a compiler-compiler. Comm. ACM, 1966, vol. 9, pp. 3-9.

13. J.V. Garwick — The definition of programming languages by their compilers. [36], pp. 139-147.

14. S. Ginsburg, S.A. Greibach and M.A. Harrison — Stack Automata and Compiling. J. ACM, 1967, vol. 14, pp. 172-201.

15. S. Igarashi — On the logical schemes of algorithms. Information Processing in Japan, 1963, vol. 3, pp. 12-18.

16. S. Igarashi — An axiomatic approach to the equivalence problems of algorithms with applications. Ph.D. thesis, University of Tokyo, 1964.

17. S. Igarashi — A formalization of the description of languages and the related problems in a Gentzen type formal system. RAAG Research Notes, Third Series, no. 80, 1964.

18. L.A. Kaluzhnin — Algorithmization of Mathematical Problems. Problems of Cybernetics, 1961, vol. 2, pp. 371-392.

19. H. Kaphengst — Eine abstrakte programmgesteuerte Rechenmaschine. Z. Math. Logik und Grundlagen der Mathematik, 1959, vol. 5, pp. 366-379.

20. P.J. Landin — The mechanical evaluation of expressions. Comp. J., 1964, vol. 6, pp. 308-320.

21. P.J. Landin — A formal description of ALGOL 60. [36], pp. 266-294.

22. P.J. Landin — A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM, 1965, vol. 8, pp. 89-101, pp. 158-165.

23. R.L. London — A computer program for discovering and proving sequential recognition rules for well-formed formulas defined by a Backus normal form grammar. Ph.D. thesis, Carnegie Institute of Technology, 1964.

24. W.D. Maurer — A theory of computer instructions. J. ACM, 1966, vol. 13, pp. 226-236.

25. J. McCarthy — Towards a mathematical theory of computation. Proc. IFIP Congress 1962, (Ed. C.M. Popplewell), Amsterdam, North-Holland, 1963, pp. 21-28.

26. J. McCarthy — A basis for a mathematical theory of computation. Computer Programming and Formal Systems (Ed. P. Braffort and D. Hirschberg). Amsterdam, North-Holland, 1963, pp. 33-69.

27. J. McCarthy — Problems in the theory of computation. Proc. IFIP Congress 1965, vol. 1 (Ed. A. Kalenich), Washington, Spartan Books, 1965, pp. 219-222.

28. J. McCarthy — A formal description of a subset of ALGOL. [36], pp. 1-12.

29. J. McCarthy and J. Painter — Correctness of a compiler for arithmetic expressions. Technical Report CS 38, Computer Science Dept., Stanford University, 1966.

30. W.M. McKeeman — An approach to computer language design. Technical Report CS 48, Computer Science Dept., Stanford University, 1966.

31. P. Naur — Proof of Algorithms by General Snapshots. B.I.T., 1966, vol. 6, pp. 310-317.

32. M. Nivat and N. Nolin — Contribution to the definition of ALGOL semantics. [36], pp. 148-159.

33. R. Peter — Graphschemata und rekursive Funktionen. Dialectica, 1958, vol. 12, pp. 373-393.

34. PL/I Definition Group of the Vienna Laboratory — Formal Definition of PL/I. IBM Technical Report TR 25.071, 1966.

35. F.H. Raymond — Etude générale des structures de calculatrices à préfixes et à piles, I. Chiffres, 1966, vol. 9, pp. 235-277.

36. T.B. Steel, Jr. (Ed.) — Formal Language Description Languages for Computer Programming. Proceedings IFIP Working Conference, Vienna, 1964. Amsterdam, North-Holland, 1966.

37. T.B. Steel, Jr. — A formalization of semantics for programming language description. [36], pp. 25-36.

38. C. Strachey — Towards a formal semantics.
[36], pp. 198-220.

39. H. Thiele — Wissenschaftstheoretischen Untersuchungen in Algorithmische Sprachen.
Berlin, VEB, 1966.

40. H. Wang — Machines, sets and the decision problem.
Formal Systems and Recursive Functions (Ed. J.N. Crossley and M.A.E. Dummett).
Amsterdam, North-Holland, p. 306.

41. N. Wirth and C.A.R. Hoare — A contribution to the development of ALGOL.
Comm. ACM, 1966, vol. 9, pp. 413-432.

42. N. Wirth and H. Weber — EULER, a Generalization of ALGOL, and its Formal Definition.
Comm. ACM, 1966, vol. 9, pp. 13-23, pp. 89-99.

43. A. van Wijngaarden — Generalized ALGOL.
Proc. ICC Symposium on Symbolic Languages in Data Processing.
New York, Gordon and Breach, 1962, pp. 409-419.
Also in
Annual Review in Automatic Programming, R. Goodman (Ed.), vol. 3, pp. 17-26.
New York, Pergamon Press, 1963.

44. A. van Wijngaarden — Recursive definition of syntax and semantics.
[36], pp. 13-24.

45. Y.I. Yanov — The logical schemes of algorithms.
Problems of Cybernetics, 1960, vol. 1, pp. 82-140.

46. H. Zemanek — Semiotics and Programming Languages.
Comm. ACM, 1966, vol. 9, pp. 139-143.