9169

RA

Reprinted from: ADVANCES IN INFORMATION SYSTEMS SCIENCE, Vol. 2
(Plenum Press, 1969)

D *Chapter 3*

SEMANTICS OF PROGRAMMING LANGUAGES

J. W. de Bakker
Mathematical Center
Amsterdam, the Netherlands

MR 109

1. INTRODUCTION

This chapter is a survey of the research on the semantics of programming languages. We feel that it is neither feasible nor desirable for our aim here to make an attempt at a rigorous definition of the term "semantics." This would require discussion both of the various proposals in the literature for such a definition, and of the relation of the semantics of programming languages to semantics as studied in linguistics, mathematical logic, and philosophy. Therefore we restrict ourselves to a description, mainly of allusive nature, of the sort of problems which are considered in semantics.

Semantics is concerned with meaning. To be more specific: Semantics of programming languages is concerned with the study of the meaning of the constituent concepts of these languages, of their mutual relationships, and of their applications in individual programs.

We add a few comments on this description:

1. If formal methods are used in these studies, one might prefer to speak of formal semantics. Since all investigations considered in this paper employ some kind of formalism, we omit this qualification.

2. Our survey is almost exclusively devoted to research on machine-independent, general-purpose languages such as FORTRAN, ALGOL 60, or PL/I. No separate attention is paid to special-purpose languages, e.g., for simulation, list processing, real-time processes, etc., nor to languages for on-line communication with computers. Only those concepts of assembly languages are considered which are also present in machine-independent languages (e.g., iteration).

RA

3. An alternative to the phrase "the meaning of the constituent concepts" is "the relation between the meaning and the symbolic representation of the constituent concepts." This alternative is rejected, since it suggests more involvement with syntactic problems than is actually present in a considerable part of the work to be discussed.

4. An alternative to "meaning" is "effect upon some processor (either a human, an abstract machine, or a real computer)." This would exclude various approaches to semantics which are not of a constructive type, e.g., axiomatic methods, and is therefore also rejected.

5. Occasionally, in programming literature a distinction is made between "algorithm" and "program" (see, e.g., Knuth ([73])). Algorithm is then a more general term, referring to an effective process for obtaining the solution of a certain problem, whereas a program is the precise description of such a process in terms of some programming language. The present paper does not adhere to this terminology; in our opinion, an attempt at consistent use would raise more problems than it would solve.

The theory of programming languages is concerned with, in addition to semantics, syntax and pragmatics. It may be useful to add a short description of these two other fields:

1. *Syntax* is concerned with the study of formal systems to be used for the definition of grammars of programming languages. A grammar is a set of rules prescribing which sequences of symbols over a given alphabet constitute a program in the language concerned. It should define the structure of a program in such a way that efficient translation is possible.

2. In *pragmatics* one studies the relation between the language and its interpreter. If the interpreter is a human being, say, a programmer, one investigates the applications of the various concepts in the language to the problem he has to solve. One may also think of interpretation by a machine, which leads to the "mechanical pragmatics" of Gorn ([61]). A general discussion of pragmatics and its relation to syntax and semantics is given by Zemanek ([140]).

By far the largest part of the research in the theory of programming languages deals with syntactic problems. Formal systems for syntax definition are considered from an abstract point of view in the rapidly growing theory of context-free languages and their generalizations (Ginsburg ([59]), Aho and Ullman ([2])); the practical application of these systems to the construction of compilers is discussed in the extensive survey of Feldman and Gries ([55]).

Compared with syntax, semantics is a somewhat neglected subject. This is probably due to the fact that it has as yet no direct applications to practical problems in programming which are of the same importance as those of syntax for compiler building. However, there are a number of important long-range goals for a semantic theory:

1. It should provide a framework in which properties of individual programs can be investigated and proofs about these properties can be obtained, the ultimate goal being the proof that a given program solves a certain problem.
2. It should lead to methods for the complete formal definition of languages, to be used as a reference by compiler writer and programmer, and for standardization purposes.
3. It should provide a theoretical framework for the design and comparison of languages.
4. In combination with formal studies of machine languages, it should be applied to the construction of compilers. Investigations of the meaning of the language concepts may be of use in comparing alternative meaning-preserving implementations. The relation between optimization of source programs and object programs should be studied. The final goal is again to prove the correctness of compilers.

Some further remarks on proofs about programs and on motivations for formal definition are made in the introductions of Sections 2 and 3, respectively.

We have divided our survey of the research on semantics into two parts. The first (Section 2) is devoted primarily to the discussion of investigations concerning one or more basic concepts in programming, whereas in the second part (Section 3) the emphasis is on research dealing with complete languages. Generally speaking, in Section 2 programming concepts are considered as mathematical objects, without paying much attention to their symbolic representation, i.e., to syntactic problems. In Section 3 the relation between syntax and semantics often plays an important role.

Section 2 begins with some general remarks on the research on basic programming concepts; moreover, the relevance of computability theory for semantics is discussed briefly. In Section 2.2 we treat the important results on program schemata of Yanov and of Luckham, Park, and Paterson. Section 2.3 deals with axiomatic characterizations of assignment, conditions, and **goto** statements. It is based largely on the work of Igarashi. A discussion of some aspects of McCarthy's theory of computation follows in Section 2.4. In Section 2.5 we have collected various investigations dealing with flow diagrams.

The majority of the research reviewed in Section 3 deals with methods for the formal definition of programming languages. After a discussion of the motivations for formal language definition, we give a survey of the methods which have been used or proposed for this purpose, viz., the methods of van Wijngaarden and Caracciolo, based on extended Markov algorithms, the state vector approach of McCarthy and its applications to proofs about compilers, the Vienna method developed for the formal definition of PL/I, and the λ-calculus approach of Landin, Strachey, and others. A brief discussion is given of some other methods, viz., the system used for the definition of ALGOL 68, the proposal of Hoare for an axiomatic method, compiler-oriented methods, and the semantics of context-free languages of Knuth.

The preceding summary was intended to give an impression of the scope of the present chapter. Clearly, semantics has many relations with other fields in programming and mathematics. We mention only: the theory of syntax, techniques for compiler construction, automata theory, mathematical models of computers and various abstract machines, mathematical logic, in particular computability theory, graph theory, mathematical linguistics, etc. Many instances of the relation of semantics to these fields can be found in the literature to be reviewed. However, a systematic treatment of them would be a formidable task, exceeding by far the scope of this survey.

Some topics which might be considered to belong to semantics, but which are not discussed separately below, are mentioned in Section 2.1. Finally, we have, apart from a few exceptions, omitted discussion of the rather extensive Russian literature on semantics. For this we refer to the survey paper of Ershov and Lyapunov ([53]).

2. BASIC CONCEPTS

2.1. Introduction

2.1.1. General Remarks

Section 2 is devoted to investigations concerning one or more basic concepts in programming languages.

Which concepts one considers as fundamental is to some extent a matter of taste. It will depend heavily on his experience in using or designing languages, and on the type of problems one has to solve. The following list is a first approximation. It has no pretense of completeness, but is intended as a minimal set, to which other elements may be added if required. In addition, not all concepts listed are independent of each other. No special meaning should be attached to the order of the list.

1. Real and integer arithmetic; operations on other simple types, Boolean, string, or character; constants.
2. Expressions; evaluation in relation to the value, type, and scope of their variables; name-value relation.
3. Data structures: simple types, vectors, arrays, trees, structures, records, files, lists, rings; pointers and references; relation to storage allocation.
4. Conditional constructions; generalization to selection from *n*-tuples.
5. Sequencing, labels, **goto** statements and repetitive clauses, iteration versus recursion.
6. Parallel computation.
7. Assignment.
8. Procedures and functions, parameter mechanisms, side effects, recursion.
9. Blocks, locality.
10. Declarations, relation to locality, introduction of new data structures or operators, initialization.
11. Input/output.

In order to keep the size of this chapter within reasonable bounds, it was necessary to make a selection from the concepts in this list. As a first criterion, we chose the distinction between imperative and descriptional features of languages, and decided to give preference to discussion of the former. Consequently we have omitted separate treatment of the large number of investigations dealing with various data structures. (It should be noted that this does not imply that we pay no attention whatever to data structures. A great variety of them occur in the languages to be treated in Section 3, and a substantial part of the literature reviewed there is concerned with these structures. What we do omit is discussion of papers which are exclusively devoted to them.) The same criterion applies largely to input/output, which is therefore also not considered in Section 2.

Parallel computation, though an imperative feature, is not treated for another reason. It has only fairly recently appeared as a concept in programming languages. As a result of this, it has been investigated mainly from a pragmatic point of view, i.e., by discussion of examples of the sort of problems in which it can be applied. However, such pragmatic considerations are outside the scope of our chapter.

Occasionally, we have departed from our rule on the division of the material between Sections 2 and 3. For instance, a number of articles dealing with concepts as indicated in the first two entries of the list, are included in Section 3.5, since their treatment could be combined there with that of a particular technique for formal language definition.

We now make some general remarks on the results to be reported in this section. It may be of interest to note that the systems to be discussed are almost entirely "processor-independent," i.e., direct use of (abstract) machines is avoided. This in contrast to the methods of Section 3, the majority of which makes extensive use of such machines.

It will appear that many different approaches are used; there is no general framework in which all results can be stated in a unified and systematic manner. A complete synthesis may be a long way off. However, many of the results are closely related to each other, and in several cases clarification of their relation seems quite a promising subject for further research.

There is one common feature shared by most of the systems, however great their differences be otherwise, viz., use of various notions of equivalence between (parts of) programs. The results in this direction may be considered as first steps toward a solution of the problem of reducing a program to an equivalent one which is simpler according to some standard. Which standard is to be applied depends on the circumstances. There are the usual requirements on minimizing execution time or storage space. However, we feel that another criterion will become increasingly important in the future, viz., whether the program is in a form which is suited for obtaining proofs about it. These may either show that the program satifies certain conditions, e.g., termination for given input, or, more ambitiously, that it solves a given problem. It should be added that only very little is known as yet on general techniques for proving the correctness of programs. When one considers the tremendous amount of time and effort spent on debugging programs, it is surprising how relatively little attention has been paid in programming research to the development of such techniques.

For a more extensive discussion of proofs about programs and of motivations and goals for a theory of semantics in general, the reader should first of all consult McCarthy ([98,99]). Cooper ([39]) is a more recent survey paper on proofs about programs (see also Section 2.4).

2.1.2. Semantics and Computability Theory

Computability theory, i.e., the theory of Turing machines, recursive functions, etc. (Davis ([43])), has yielded a number of results which determine the essential limits imposed upon the theory of semantics. Apart from this, however, its relevance for semantics is rather limited. Most of the basic concepts of programming languages, as listed above, have no direct counterparts in one of the various systems of computability theory. Therefore it does not provide much help in investigating the properties of these concepts. Those features of programming which do have some counterpart in computability theory, such as sequencing, have, generally speaking, not been studied independently in it. Moreover, computability theory is concerned with (undecidability theorems on) "mass problems," rather than with the study of individual algorithms; again, these results are not very useful for the sort of problems one considers in programming theory.

However, some qualifications are in order with respect to our rather negative judgment on the applicability of computability theory to programming. A link between the two theories may result from the growing interest in the quantitative aspects of Turing machines, e.g., investigations on bounds for the number of operations needed for a certain calculation (for references see, e.g., Aho and Ullman ([2]). Though most of this work is still outside the scope of this paper, there are a number of related investigations which are no doubt of importance for programming. We mention Meyer and Ritchie ([104,105]). They are interested in the derivation of bounds for the running time of loop programs, i.e., sequences of, possibly nested, repetitive clauses and simple assignment statements of the form $x := 0$, $x := x + 1$, and $x := y$. The functions defined by loop programs are shown to coincide with primitive recursive functions. A bounding function for the running time of a loop program is given, depending only on the number of its instructions and on the depth of nesting of its loops.

Concepts from programming and computability theory are also related to each other, though on a highly abstract level, by Eilenberg and Elgot ([47]), who use iteration to give an algebraic characterization of recursive functions.

Applications of (extended) Markov algorithms and of the λ-calculus can be found in the literature reviewed in Sections 3.2 and 3.5, respectively. Post's canonical systems are applied by Donovan and Ledgard ([45]).

2.2. Program Schemata

2.2.1. Introduction

In this section we discuss two papers which are of fundamental importance for the semantics of basic programming concepts, that of Yanov ([137]) (a short summary was given by Yanov ([135,136])), which is essentially an investigation of the sequencing concept in its relation to the values of the conditions which determine the flow of control, and that of Luckham *et al.* ([91]), who have extended Yanov's work in the sense that they also take into account the notion of assignment.

Both papers may be viewed as studies of properties of flow diagrams, and might therefore have been discussed in Section 2.5. However, since we want to give a somewhat more detailed explanation of them, we have introduced a separate section for this purpose. Even this will not allow us to give more than a first impression of the principal ideas and results of the two papers. For a more comprehensive account of the first part of Yanov's paper ([137]) we refer to Fels ([56]) (the reader should be warned that this contains some errors in its comments on Yanov's main theorem). Some further references are noted at the end of Section 2.2.2.

2.2.2. Yanov's Program Schemata

Yanov's starting point is the following observation: The application of an algorithm to one of its arguments determines uniquely a sequence of elementary actions. In general, different arguments result in different sequences. However, it is always possible to find a finite set of predicates, representing properties of the arguments, such that the sequence of elementary actions to be performed for a given argument may be considered as a function of the values of these predicates for that argument. As a tool for investigating this idea, Yanov introduces the notion of a program schema.

Let $A = \{A_1, A_2, \ldots, A_m\}$ be a finite set of symbols denoting the elementary actions. Elements of A are called operators.

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a finite set of propositional variables, from which predicates can be formed by means of the logical operators $\neg$, $\wedge$, and $\vee$. The identically true (false) predicate is denoted by 1 (0).

We do not give the formal rules for the construction of program schemata from A and P, but illustrate this process by two examples in an ALGOL-like notation.

Example 1:

```
L₁: A₁; if p₁ then goto L₁.
```

Example 2:

```
    if ¬ p₁ ∧ p₂ then goto L₁;
L₂: A₁;
    if ¬ p₂ then goto L₂;
    if 1 then goto L₃;
L₁: A₂;
L₃:
```

It appears that program schemata look somewhat like ALGOL 60 programs. Their constituents are operators (all operators in a schema are required to be different) and conditional **goto** statements, the conditions of which are predicates over P; both of these may be labeled. Would some interpretation be provided for the operators, e.g., as assignment statements, then ordinary ALGOL 60 programs would result. However, such interpretation is omitted on purpose. Consequently, a program schema cannot be executed in the usual way, since it is not known how the values of the conditions—assuming that for a given argument initial values are given—change during the execution of the schema. Now Yanov's central idea is to give these changing values of the conditions in advance. A program schema is then considered as a function of the sequence of values of its constituent conditions. A more precise explanation follows.

Let $p_1, p_2, \ldots, p_k$ be the propositional variables occurring in a given schema. An evaluation of these variables is defined as an ordered k-tuple of zeros and ones (corresponding to the values "false" and "true"). An evaluation sequence is a sequence of such k-tuples. A program schema as function of an evaluation sequence is executed as follows: The first element of the evaluation sequence is considered, the corresponding elements of this k-tuple are assigned to the propositional variables of the schema, and from this the values of its conditions are determined. One then starts to execute the schema in the ordinary ALGOL way. However, as soon as an operator is met the next element of the evaluation sequence is considered, the propositional variables are assigned the corresponding elements of this second k-tuple, the new values of the conditions are determined, and the execution is continued until another operator is met, after which the third k-tuple is considered, etc. The execution terminates, if ever, with the execution of the last "statement" of the schema. (The transition to the next element of the evaluation sequence each time an operator is met reflects the possible change in the properties of the argument being transformed as a result of the execution of this operator.) We have not yet said what is meant by the value of a program schema when applied in this way to an evaluation sequence. It is defined to be the sequence of operators as encountered successively during the execution of the schema.

We illustrate these definitions by means of example 2 above. The number of propositional variables in this schema is two; hence evaluation sequences for it are sequences of pairs. Application of the schema to the sequence $(1, 1)$, $(1, 0)$, $(1, 1)$, $\ldots$ yields the value A_1A_1; application to the sequence $(0, 1)$, $(0, 1)$, $\ldots$ yields the value A_2.

Next we define what is meant by the equivalence of two schemata:

Two program schemata over A and P are equivalent if and only if they have the same value for each evaluation sequence.

Note that this is a very strong notion of equivalence. The corresponding definition for programs would not only require that the final values of their variables be identical, but in addition that these values have been obtained by performing the same elementary actions in the same order.

The equivalence problem for program schemata is shown to be decidable. An effective procedure is given which reduces each program schema to a canonical form, and it is proved that two schemata are equivalent if and only if their canonical forms satisfy a certain effectively verifiable condition.

Yanov also gives an axiomatic characterization of equivalence. He introduces a set of axioms and rules of inference, and proves that if two schemes S_1 and S_2 are equivalent by the definition given above, then $S_1 \sim S_2$ can be derived in this system. We give two examples of his axioms:

> **if** $p_1 \wedge p_2$ **then goto** $L_1 \sim$
>
> **if** $\neg p_1$ **then goto** L_2; **if** p_2 **then goto** L_1; L_2:

and

> **if** p_1 **then goto** L_1; S_1; L_1: **if** p_1 **then goto** L_2; S_2; L_2:
>
> $\sim$
>
> **if** p_1 **then goto** L_1; S_1; **if** p_1 **then goto** L_2; S_2; L_2: L_1:

(Here S_1 and S_2 stand for arbitrary program schemata. For the second axiom also see Section 2.4.3.)

Finally, Yanov introduces a matrix notation for program schemata and studies its relation to the linear notation.

Rutledge (116) has given a simplified proof of Yanov's main result, viz., the decidability of the equivalence problem. Moreover, it is shown that the same problem for an extended notion of program schema—in which the requirement that all operators of the schema be different is omitted—is just the equivalence problem for finite automata. The relation between program schemata and finite automata was also noted by Igarashi (64).

Yanov's work has been extended in several directions in the Russian research on semantics; we mention Ershov (52) and Yanov (138). Further references are given by Ershov and Lyapunov (53).

2.2.3. The Formalized Computer Programs of Luckham et al. (91)

In a paper by Luckham *et al.* (91) a natural extension of Yanov's program schemata is considered which amounts to the introduction of variables and assignment. The elementary actions of a schema are no longer left completely unspecified, but are assumed to be of the form $x_i := f_j^n (x_1, x_2, \ldots, x_n)$, and conditions are of the form $p_k(x_l)$. An example of such a schema is

```
$L_1$: $x_1 := f_1^1(x_1)$;
    **if** $p_1(x_1)$ **then goto** $L_3$;
$L_2$: $x_1 := f_1^1(x_1)$;
    **if** $p_1(x_1)$ **then goto** $L_1$;
    **if** 1 **then goto** $L_2$;
$L_3$: $x_2 := f_1^1(x_1)$;
    $x_1 := f_1^1(x_1)$
```

An interpretation I of a program schema (from now on taken in the extended sense) is established as follows:

1. A domain D is selected.
2. To each variable x_i occurring in the schema there is assigned an element of D (to be considered as its initial value).
3. To each f_j^n there is assigned an n-ary function $D^n \to D$.
4. To each p_i there is assigned a function $D \to \{0, 1\}$.

An interpreted program schema P_I can be executed in the usual way. If the execution terminates, the value of P_I, denoted by val(P_I), is defined to be the vector of the final values of the variables occurring in P.

The following notions of equivalence are introduced:

1. $P \equiv P'$ if and only if for all interpretations I, val(P_I) = val(P_I'), whenever either value is defined.

2. $P \equiv_F P'$ if and only if for all interpretations I on finite domains, val(P_I) = val(P_I'), whenever either value is defined. (An example is given to show that $P \equiv_F P'$ does not imply $P \equiv P'$.)

3. $P \simeq P'$ if and only if for all interpretations I, val(P_I) = val(P_I'), whenever both values are defined.

4. A relation $\sim$ between program schemata is called "reasonable" if for all schemata P and P' the following two conditions are satisfied: (a) $P \equiv P'$ implies $P \sim P'$, (b) $P \sim P'$ implies $P \simeq P'$.

Whereas the equivalence problem of the previous section was decidable, this is no longer the case for the various notions of equivalence introduced here. A number of undecidability results on multihead automata are derived

—ultimately based on the undecidability of the halting problem for Turing machines—which are then applied, by simulating automata by schemata, to the equivalence problems for schemata. (Incidentally, this provides one of the very few examples of the application of automata theory to semantics.) The main results are the following: Let P_0 be the schema L: **goto** L; let P_1 be $x_1 := f_1^1(x_3)$; $x_2 := f_1^1(x_3)$; and let P_2 be the example given above. Then the following relations are not partially decidable (i.e., not recursively enumerable) for arbitrary P:

1. $P \equiv P_0$.
2. $P \equiv_F P_0$.
3. $P \equiv_F P_1$.
4. $P \simeq P_1$.
5. $P \not\simeq P_1$.
6. $P \sim P_2$, for each reasonable $\sim$.

These results imply that all hope for a general simplification algorithm for programs which use at least the three concepts of assignment, conditions, and goto statements, is in vain.

In the last section of the paper by Luckham *et al.* ([91]) some subclasses of program schemata are considered for which the equivalence problem (with respect to $\equiv$) is decidable. For instance, the following result is mentioned (a proof is given by Paterson ([110])): Let a schema be called monadic if all functions occurring in it are functions of one variable only. The equivalence problem for monadic schemata with nonintersecting loops is decidable.

2.3. The Axiomatic Approach

2.3.1. Introduction

The main representative of the axiomatic approach to semantics is Igarashi ([66]) (for an introduction see Igarashi ([67])). Since this paper is not easily accessible, it will be treated in somewhat greater detail. Igarashi's axiom systems for assignment, conditions, and **goto** statements are discussed in Sections 2.3.2, 2.3.3, and 2.3.4 respectively. Igarashi's paper ([66]) contains also a sketch for an axiomatic treatment of input/output and of arrays, and some further applications. Another paper of Igarashi ([65]), is partly preparatory to the later paper ([66]) and partly concerned with an axiomatic approach to syntactic problems.

Some general reflections on the advantages of an axiomatic treatment of concepts in programming, with the emphasis on its application to the formal definition of languages, can be found in Hoare ([63]) (also see Section 3.6).

We shall not enter here into a discussion of the respective merits of the axiomatic method versus various other systems, in particular those of a constructive nature. However, we feel that it is fair to say that it deserves more attention than has been paid to it up to now.

2.3.2. Axioms for Assignment

We cannot treat Igarashi's axiom system for assignment in full, but shall concentrate on its essential features. Therefore some definitions will be given only somewhat loosely.

Let $V = \{x, y, z, \ldots\}$ be a set of variables. (Only simple variables—in the sense of ALGOL 60—are considered.)

Let $F = \{f, g, \ldots\}$ be a set of functions. Specification of the nature of F is omitted here. When we want to indicate that f depends on the variable x, we write $f(x)$; $f(g)$ then denotes the result of substituting g for all occurrences of x in f (with the usual precautions).

An assignment statement has the form $x := f$, for some $x \in V$ and $f \in F$. It is convenient to consider the dummy statement, denoted by θ, also as an assignment statement.

A, B, C, $\ldots$ denote arbitrary sequences of assignment statements, separated by semicolons, and $\lambda(A)$ is the set of all variables constituting the left-hand sides of the statements in A; $\varrho(A)$ is the set of all variables occurring as arguments of the functions in the right-hand sides of the statements in A, and $\tau(A) = \lambda(A) \cup \varrho(A)$. We use $A(x)$ and $A(g)$ similarly to $f(x)$ and $f(g)$, and Φ denotes the empty set.

Well-formed formulas of the axiomatic theory are expressions of the form $A \underset{X}{\sim} B$, with $X \subset V$. Such a formula may be understood intuitively as: The sequences A and B have the same effect upon the variables from the set X. $A \underset{V}{\sim} B$ is abbreviated to $A \sim B$.

We can now formulate the axiom system. It consists of six axioms and four rules of inference.

Axioms:

Ia_1: $x{:=}x \sim \theta$.

Ia_2: $A;\theta \sim A$,

$\theta;A \sim A$.

Ia_3: If $X \cap \lambda(A) = \Phi$, then $A \underset{X}{\sim} \theta$.

Ia_4: If $\lambda(A(x)) \cap \tau(x{:=}f) = \Phi$, then
$x{:=}f;\ A(x);\ x{:=}g(x) \sim A(f);\ x{:=}g(f)$.

Ia_5: If $x \neq y$, $\lambda(A(x)) \cap \tau(x{:=}f) = \Phi$, and $x \notin \tau(f)$, then
$x{:=}f;\ A(x);\ y{:=}g(x) \sim x{:=}f;\ A(f);\ y{:=}g(f)$.

Ia_6: If $f = g$, then $x{:=}f \sim x{:=}g$.

Rules of inference:

Ia_7: If $A \underset{\varrho(C) \cup \varrho(D) \cup X}{\sim} B$, and $C \underset{X}{\sim} D$, then $A;C \underset{X}{\sim} B;D$.

Ia_8: If $A \underset{X}{\sim} B$ and $A \underset{Y}{\sim} B$, then $A \underset{X \cup Y}{\sim} B$.

Ia_9: If $A \underset{X \cup Y}{\sim} B$, then $A \underset{X}{\sim} B$.

Ia_{10}: Symmetry and transitivity of $\underset{X}{\sim}$.

Remarks: (1) $f = g$ in Ia_6 means that the equality of f and g can be established in some suitable underlying system, depending on further specification of F. (2) The system $\{Ia_1, Ia_2, \ldots, Ia_{10}\}$ is called $\mathscr{I}a$.

As an example of the application of $\mathscr{I}a$, we consider the following two sequences:

A is $s{:=}n \times s;\ n{:=}n-1;\ s{:=}n! \times s;\ n{:=}0$, and
B is $s{:=}n! \times s;\ n{:=}0$.

The equivalence of A and B for $n > 0$ was used in an example of McCarthy ([99]) (see also Section 2.4.4). We derive $A \sim B$ from $\mathscr{I}a$:

(1) $n{:=}n-1;\ s{:=}n! \times s;\ n{:=}0 \sim$
$s{:=}(n-1)! \times s;\ n{:=}0$,Ia_4,
(2) $A \sim s{:=}n \times s;\ s{:=}(n-1)! \times s;\ n{:=}0$,(1),Ia_7,
(3) $s{:=}n \times s;\ s{:=}(n-1)! \times s \sim s{:=}(n-1)! \times n \times s$, ,$Ia_4$,
(4) $A \sim s{:=}n! \times s;\ n{:=}0$,(2),(3),Ia_7, definition of $n!$

Hence $A \sim B$.

Igarashi's main result on the system $\mathscr{I}a$ is the proof of its completeness. A formal definition of the effect of a sequence of assignment statements upon a variable is given which describes the usual meaning of assignment. (For a special case of this definition see below.) The completeness theorem asserts that two sequences A and B have the same effect upon the variables from X if and only if $A \underset{X}{\sim} B$ is a theorem of $\mathscr{I}a$.

A further analysis of the axiomatics of a simple type of assignment statement is given in de Bakker ([10]). Statements are considered the right-hand sides of which consist only of variables, such as $x := y$, $z := t$, etc. We abbreviate these to xy, zt, etc. The following axiom system is introduced (A, B, ..., now stand for such simple sequences):

Axioms:

Ba_1: $xy;yx \sim xy$.

Ba_2: $xy;xz \sim xz$, provided that $x \neq z$.

Ba_3: $xy;zx \sim xy;zy$.

Ba_4: $xy;zy \sim zy;xy$.

Rules of inference:

Ba_5: If there exist x, y, z, t $(x \neq y)$ such that $A;xz \sim B;xz$ and $A;yt \sim B;yt$, then $A \sim B$.

Ba_6: If $A \sim B$, then $A;C \sim B;C$ and $C;A \sim C;B$.

Ba_7: Symmetry and transitivity of $\sim$.

The system $\{Ba_1, Ba_2, \ldots, Ba_7\}$ is denoted by $\mathscr{B}a$. It was developed with the view to a more detailed investigation of the relations between the various axioms. Some results de Bakker ([10]) are:

1. The axioms of $\mathscr{B}a$ are independent.

2. The effect of A upon x is described by the function $E(x,A)$, defined recursively by (a) $E(x,xz) = z$, and $E(x,yz) = x$, (b) $E(x,A;B) = E(E(x,B),A)$. The completeness theorem can then be formulated as: $A \sim B$ if and only if for all $x \in V$: $E(x,A) = E(x,B)$. (Our proof differs somewhat from Igarashi's.)

3. It is investigated to see whether it is possible to replace the four axioms of $\mathscr{B}a$ by a smaller set, such that the resulting system remains equipollent with $\mathscr{B}a$ (i.e., the same equivalences can be derived from it). A typical theorem is the following: ($(A)^n$ denotes the sequence $A;A;\ldots;A$ (n times A, $n \geq 1$)). Let $\mathscr{C}a_n^{(i)}$ be defined by:

$$\mathscr{C}a_n^{(i)} = \{Ba_2, Ba_5, Ba_6, Ba_7\} \cup \{Ca_n^{(i)}\}, \qquad i = 1, 2$$

$Ca_n^{(1)} = (xy;zx;yz)^n \sim zy;xy$, and $Ca_n^{(2)} = (xy;zx;yz)^n \sim zy;xz$. Then the following hold: (a) For each $n \geq 1$, $\mathscr{C}a_n^{(1)}$ is equipollent with $\mathscr{B}a$. (b) $\mathscr{C}a_1^{(2)}$ is equipollent with $\mathscr{B}a$. (c) $\mathscr{C}a_{2m}^{(2)}$ $(m \geq 1)$ is not equipollent with $\mathscr{B}a$. The equipollence of $\mathscr{C}a_{2m+1}^{(2)}$ $(m \geq 1)$ with $\mathscr{B}a$ is an open problem.

A third axiomatic characterization of assignment is studied by Kaplan ([71]). It is based upon the properties of state vector functions associated with assignment introduced by McCarthy ([99]). A discussion of these ideas is given in Section 2.4.3.

2.3.3. *Axioms for Conditional Expressions*

Axiom systems for conditional expressions have been given by McCarthy ([98]) and Igarashi ([66]). We shall present both systems; it will turn out that they are equipollent: McCarthy's axioms can be derived from Igarashi's system and *vice versa.*

Let $P = \{p, q, r, \dots\}$ be a set of propositional variables; P^* the set of propositions which can be constructed from P by means of the operators $\neg$, $\wedge$, and $\vee$; and 1 (0) the identically true (false) proposition.

Let X be an arbitrary set.

The set $C(P, X)$ of conditional expressions over P and X is defined by:

(a) $X \subset C(P, X)$,

(b) If $\pi \in C(P, P^*)$, $\alpha \in C(P, X)$, and $\beta \in C(P, X)$, then $(\pi \to \alpha, \beta) \in C(P, X)$.

An expression $(\pi \to \alpha, \beta)$ can be interpreted as the ALGOL 60 expression **if** π **then** α **else** β; i.e., if π has the value 1 (0) then the value of the expression is the value of α (β). Two conditional expressions are called equivalent if and only if they have the same values for all values of the propositional variables occurring in them. (We omit discussion of the case—considered in detail by McCarthy ([98])—that the value of a propositional variable is undefined.)

An axiomatic characterization of equivalence for conditional expressions is now introduced. First we give McCarthy's system. Let $C_1(P, X)$ be defined as $C(P, X)$, except that in clause (b) above we replace $\pi \in C(P, P^*)$ by $\pi \in C(P, P)$. Well-formed formulas of the theory are of the form $\alpha \sim \beta$, with α and $\beta \in C_1(P, X)$.

The axioms are:

M_1: $(p \to \alpha,\alpha) \sim \alpha$,

M_2: $(1 \to \alpha,\beta) \sim \alpha$,

M_3: $(0 \to \alpha,\beta) \sim \beta$,

M_4: $(p \to (p \to \alpha,\beta),(p \to \gamma,\delta)) \sim (p \to \alpha,\delta)$,

M_5: $((p \to q,r) \to \alpha,\beta) \sim (p \to (q \to \alpha,\beta),(r \to \alpha,\beta))$,

M_6: $(p \to (q \to \alpha,\beta),(q \to \gamma,\delta)) \sim (q \to (p \to \alpha,\gamma),(p \to \beta,\delta))$.

The rules of inference are stated here only informally. They assert that equivalence is preserved by the systematic substitution of an element of $C_1(P, P)$ for a propositional variable, and also by replacement of an occurrence of a subexpression by an equivalent subexpression.

The system $\{M_1, M_2, \ldots, M_6\}$ is called $\mathscr{M}_1$. It provides a complete characterization of conditional expressions: Two expressions α and β from $C_1(P, X)$ are equivalent by the above given definition if and only if $\alpha \sim \beta$ is a theorem of $\mathscr{M}_1$. This is proved by reducing each $\alpha \in C_1(P, X)$ to a canonical form $\alpha' \sim \alpha$, and by showing that α and β are equivalent if and only if $\alpha' = \beta'$.

Next we treat Igarashi's system. Let $C_2(P, X)$ be defined as $C(P, X)$, except that in clause (b) above we replace $\pi \in C(P, P^*)$ by $\pi \in P^*$. Well-formed formulas are of the form $\alpha \sim \beta$, with α and $\beta \in C_2(P, X)$.

The axioms are:

Ic_1: $(p \rightarrow \alpha,\alpha) \sim \alpha$,

Ic_2: $(1 \rightarrow \alpha,\beta) \sim \alpha$,

Ic_3: $(p \rightarrow \alpha,\beta) \sim (\neg p \rightarrow \beta,\alpha)$,

Ic_4: $(p \rightarrow (q \rightarrow \alpha,\beta),\gamma) \sim (p \wedge q \rightarrow \alpha, (p \wedge \neg q \rightarrow \beta,\gamma))$,

Ic_5: $(p \rightarrow \alpha, (q \rightarrow \beta,\gamma)) \sim (p \rightarrow \alpha, (\neg p \wedge q \rightarrow \beta,\gamma))$.

Statement of the rules of inference is omitted. It is assumed that the usual rules for $\neg$, $\wedge$, and $\vee$ hold. (Note that this was not necessary in $\mathscr{M}_1$.) The system $\{Ic_1, Ic_2, \ldots, Ic_5\}$ is called $\mathscr{I}c_1$. It is a complete axiom system for $C_2(P, X)$. This is also proved *via* the introduction of a canonical form, which differs, however, from McCarthy's.

The systems $\mathscr{M}_1$ and $\mathscr{I}c_1$ cannot be compared directly, since they refer to the different sets $C_1(P, X)$ and $C_2(P, X)$. In order to clarify the relation between the two systems, it is necessary to extend both. Let $\mathscr{M} = \mathscr{M}_1 \cup \{M_7, M_8, M_9, M_{10}\}$, where

M_7: $(p \rightarrow 1,0) \sim p$,

M_8: $(p \rightarrow 0,1) \sim \neg p$,

M_9: $(p \rightarrow q,0) \sim p \wedge q$,

M_{10}: $(p \rightarrow 1,q) \sim p \vee q$,

and $\mathscr{I}c = \mathscr{I}c_1 \cup \{Ic_6\}$, where

Ic_6: $(p \rightarrow q,r) \sim (p \wedge q) \vee (\neg p \wedge r)$.

Then the following theorem holds: The systems $\mathscr{M}$ and $\mathscr{I}c$ are equipollent axiom systems, and both provide a complete characterization of $C(P, X)$. That $\mathscr{M}$ can be derived from $\mathscr{I}c$ was shown by Igarashi ([66]). For a proof of the reverse result we refer to de Bakker ([11]).

In order to deal with the relation between conditional expressions and functions, both authors have introduced the following axiom:

$$M_{11}: f((p \rightarrow \alpha,\beta)) \sim (p \rightarrow f(\alpha), f(\beta)).$$

Another rule of both systems is:

M_{12}: Suppose that the equivalence of α and β can be shown under the assumption that p is true. Then $(p \rightarrow \alpha,\gamma) \sim (p \rightarrow \beta,\gamma)$.

Igarashi also considers a combination of conditions and assignment. He gives a complete axiom system $\mathscr{I}ac$ for sequences of possibly conditional assignment statements, consisting essentially of $\mathscr{I}a$ and $\mathscr{I}c$, to which are added:

$$Iac_1: x{:=}f;\ (p(x) \rightarrow A,B) \sim (p(f) \rightarrow x{:=}f;A,\ x{:=}f;B)$$

$$Iac_2: (p \rightarrow A,B);\ C \sim (p \rightarrow A;C,\ B;C).$$

This concludes our discussion of McCarthy's and Igarashi's work on conditions. We add a few remarks on other papers on this subject.

The axiom Ic_6 defines the meaning of $(p \rightarrow q,r)$ in Boolean algebras. De Bakker ([9]) investigates a related function $(p \rightarrow q,r)^*$ in the setting of distributive, relatively complemented lattices. The function $(p \rightarrow q,r)^*$ is the relative complement of p in the interval $(p \wedge q,\ p \vee r)$. It satisfies M_1, M_4, M_5, and M_6; moreover, it can be used in the definition of such lattices. Related results (on distributive, relatively complemented lattices with zero) are given by Dicker ([44]).

Rennie ([114]) gives an elaboration of McCarthy's system. The imperative features of conditional statements are considered and a normal form is sketched for sequences of such statements which seems similar to the form given by Igarashi.

Caracciolo ([23,25]) extends some of McCarthy's axioms, viz., M_1, M_4, and M_6, to constructions which select from n objects (cf. the case construction of ALGOL 68). This suggests relations with n-valued logic and set-theoretic notions. A beginning is made of a study of these ideas.

Some aspects of the relation between conditions and assignment are treated by Munteanu ([106]).

Finally, we mention Wittman and Ingerman ([130]), who introduce the notion of threshold selection and prove its equivalence to Boolean selection, i.e., to conditional expressions.

2.3.4. Axioms for **goto** *Statements*

Yanov's axiom system, mentioned in Section 2.2.2, provides the first example of an axiomatic treatment of **goto** statements in relation to conditional expressions.

Igarashi's axioms for **goto** statements are added to the system $\mathcal{I}ac$ described above. Let us call the resulting system $\mathcal{I}acg$. As might be expected, it no longer has the general completeness property of $\mathcal{I}a$, $\mathcal{I}c$, or $\mathcal{I}ac$. What remains is essentially the following: $\mathcal{I}acg$ is complete for the equivalence of two programs of which it is known in advance that they have a common supremum for the number of elementary operations (unspecified here) to be performed during their execution. (Note that the problem of finding such a supremum is in general undecidable.)

We shall not present the entire system $\mathcal{I}acg$, but restrict ourselves to an example of an equivalence which can be derived from it. Consider the following programs P_1 and P_2 (this time we write the conditions in the usual ALGOL 60 notation):

P_1 is

$i{:=}0$; $L{:}i{:=}i+1$; A; **if** $i<n$ **then goto** L; $i{:=}i+1$

P_2 is

$i{:=}1$; $L{:}A$; $i{:=}i+1$; **if** $i\leq n$ **then goto** L

A is an arbitrary sequence of (conditional) assignment and **goto** statements, provided that it contains no jumps to L. For the proof of $P_1 \sim P_2$ three rules of $\mathcal{I}acg$ are needed:

1. An axiom of $\mathcal{I}acg$, here loosely formulated as

   $\ldots L{:}\,B$; $M{:}\ldots$; **goto** L; $\ldots \sim$

   $\ldots L{:}\,B$; $M{:}\ldots$; B'; **goto** M; $\ldots$

(B is an arbitrary sequence; B' is derived from it by suitably renaming its labels in order to avoid "clash of labels.")

2. An equivalence which can easily be derived from $\mathcal{I}acg$:

   if p **then begin** ...; **goto** L **end else** ... $\sim$

   if p **then begin** ...; **goto** L **end**; ...

3. A rule which allows the introduction of superfluous labels and systematic renaming of labels.

Then:

$$\begin{aligned}P_1 &\sim i{:=}0;\ L{:}i{:=}i+1;\ M{:}A;\ \textbf{if}\ i<n\ \textbf{then goto}\ L;\ i{:=}i+1\\ &\sim i{:=}0;\ L{:}i{:=}i+1;\ M{:}A;\ \textbf{if}\ i<n\ \textbf{then begin}\ i{:=}i+1;\ \textbf{goto}\\ &\quad M\ \textbf{end};i{:=}i+1\end{aligned}$$

by rule 3 and rule 1, and

$$\begin{aligned}P_2 &\sim i{:=}0;\ i{:=}i+1;\ L{:}A;\ i{:=}i+1;\ \textbf{if}\ i\le n\ \textbf{then goto}\ L\\ &\sim i{:=}0;\ i{:=}i+1;\ L{:}A;\ i{:=}i+1;\ \textbf{if}\ i\le n\ \textbf{then goto}\ L\ \textbf{else}\ \theta\\ &\sim i{:=}0;\ i{:=}i+1;\ L{:}A;\ \textbf{if}\ i+1\le n\ \textbf{then}\\ &\quad \textbf{begin}\ i{:=}i+1;\ \textbf{goto}\ L\ \textbf{end else begin}\ i{:=}i+1;\ \theta\ \textbf{end}\\ &\sim i{:=}0;\ i{:=}i+1;\ L{:}A;\ \textbf{if}\ i<n\ \textbf{then}\\ &\quad \textbf{begin}\ i{:=}i+1;\ \textbf{goto}\ L\ \textbf{end};\ i{:=}i+1\end{aligned}$$

by Ia_4, rule 2, Iac_1, and rule 2, respectively. (Remember that θ denotes the empty statement.) $P_1 \sim P_2$ now follows from rule 3.

The connection between **goto** statements and conditions is also studied, though not from an axiomatic point of view, by Engeler (50). In particular, he investigates the relation between termination properties of programs and the provability of formulas in an infinitary language, viz., a language allowing countably long disjunctions.

2.4. McCarthy's Theory of Computation

2.4.1. Introduction

Of central importance for the theory of semantics are the papers of McCarthy (97–101), in particular the papers published in 1963 (98,99). They contain a rich variety of ideas on possible approaches to the mathematical investigation of basic programming concepts, and detailed studies of a number of the proposed methods. We can discuss only a selection from McCarthy's work; reasons of space prohibit a more comprehensive treatment.

The present section is based on the 1963 pagers (98,99), and deals with: recursive functions; the proof technique of recursion induction; state vectors and state vector functions—in particular, in relation to assignment statements; and recursion induction on state vector functions. Part of McCarthy's formal system for conditional expressions was treated in Section 2.3.3.

Discussion of his ideas on the formal definition of languages and applications to proofs about compilers is deferred to Section 3.3.

The list of topics which are considered in more or less detail in McCarthy's papers (98,99) but which are not discussed here includes some general reflections on motivations and goals for a theory of computation, computable functionals and the λ-and label mechanism, recursive definitions of sets, relations to other formalisms and to mathematical logic, and computer-checked proofs of programs (cf. also McCarthy (97)).

2.4.2. Recursion Induction

The formalism for conditional expressions (Section 2.3.3) can be used for the definition of functions; e.g., the function abs(x) can be defined by

$$\text{abs}(x) = \textbf{if } x \geq 0 \textbf{ then } x \textbf{ else } -x.$$

If the function being defined occurs on the right-hand side of such an equation, e.g., in the definition of the factorial function

$$f(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \times f(n-1) \tag{1}$$

then the definition and corresponding function are called recursive. (Note that this use of the term recursive, though current in the field of programming, does not coincide with that in recursive function theory as studied in mathematical logic.)

A problem which presents itself immediately with such definitions is that of convergence. In Eq. (1) it can be seen that the process of determining $f(n)$ terminates only for $n \geq 0$. However, a formal theory of convergence is not available. Therefore we shall assume below that the functions considered are convergent for the relevant arguments.

The general form of the equation for the recursive definition of a function is

$$f(x_1, x_2, \ldots, x_n) = E(f, x_1, x_2, \ldots, x_n) \tag{2}$$

where the right-hand side is a conditional expression in which f occurs. Suppose that we can show that two functions g and h both satisfy Eq. (2) for appropriate arguments. Then we say that the equivalence of g and h (for these arguments) has been proved by recursion induction.

This technique was introduced by McCarthy (98) and illustrated by several examples, both of numerical and nonnumerical (involving LISP functions) type. McCarthy (99) also applies it to state vector functions (see Sections 2.4.3 and 2.4.4). Further applications are contained in the papers on proofs about the correctness of compilers by McCarthy and Painter ([102]), Painter ([109]), and Kaplan ([70]) (Section 3.3).

Recursion induction has also been studied by Cooper ([38,39]). In the first paper ([38]) he considers three definitions of the factorial function, which are generalized by abstracting from the special properties of the functions involved in the definitions (such as multiplication). The three definitions are proved equivalent, and a particular case of the generalized function is shown to be a function which reverses the order of symbols of a list. Next he proves the equivalence of two functions for evaluating an approximation to an integral. Examination of the strategies used in these proofs has led to the discovery of a new proof rule which can be applied in situations where no proofs by recursion induction have been found. The other paper by Cooper ([39]) is a survey of research on proofs about programs. It contains some general reflections on the principles of such proofs. In addition, the work done at the Carnegie Institute of Technology on proofs about compilers (London ([85,86]), Earley ([46]), and Evans ([54])), Cooper's previous paper ([38]), and some plans for future work are discussed.

As an illustration of the nature of a proof by recursion induction, and of the application of some of the axioms of Section 2.3.3, we give an example of such a proof, taken from Cooper ([39]).

Let $f(n)$ be defined by Eq. (1), and let $g(n) = h(n, 0, 1)$, with

$$h(n, m, a) = \textbf{if } n{=}m \textbf{ then } a \textbf{ else } h(n, m{+}1, (m{+}1){\times}a) \tag{3}$$

We prove that $f(n) = g(n)$. Two auxiliary functions are introduced. Let $h'(n, m, a)$ be defined by

$$h'(n, m, a) = \textbf{if } n{=}m \textbf{ then } a \textbf{ else } n{\times}h'(n{-}1, m, a) \tag{4}$$

It is clear that $h'(n, 0, 1)$, regarded as function of n, satisfies Eq. (1); hence $f(n) = h'(n, 0, 1)$. Let $h''(n, m, a)$ be defined by

$$h''(n, m, a) = \textbf{if } n{=}m \textbf{ then } a \textbf{ else } h'(n, m{+}1, (m{+}1){\times}a) \tag{5}$$

Using Eqs. (5) and (4), M_{12} (Section 2.3.3), M_{11}, and Eq. (5), we obtain

$$\begin{aligned}
h''(n, m, a) &= \textbf{if } n{=}m \textbf{ then } a \textbf{ else}\\
&\quad\ \textbf{if } n{=}m{+}1 \textbf{ then } (m{+}1){\times}a \textbf{ else } n{\times}h'(n{-}1, m{+}1, (m{+}1){\times}a)\\
&= \textbf{if } n{=}m \textbf{ then } a \textbf{ else}\\
&\quad\ \textbf{if } n{=}m{+}1 \textbf{ then } n{\times}a \textbf{ else } n{\times}h'(n{-}1, m{+}1, (m{+}1){\times}a)\\
&= \textbf{if } n{=}m \textbf{ then } a \textbf{ else}\\
&\quad\ n{\times}(\textbf{if } n{-}1{=}m \textbf{ then } a \textbf{ else } h'(n{-}1, m{+}1, (m{+}1){\times}a))\\
&= \textbf{if } n{=}m \textbf{ then } a \textbf{ else } n{\times}h''(n{-}1, m, a)
\end{aligned}$$

It follows that $h''(n, m, a)$ satisfies Eq. (4); hence, by recursion induction, $h''(n, m, a) = h'(n, m, a)$. Replacing h'' by h' in Eq. (5), we see that h' satisfies Eq. (3); thus, again by recursion induction, $h'(n, m, a) = h(n, m, a)$. Since $f(n) = h'(n, 0, 1) = h(n, 0, 1) = g(n)$, the required result follows.

2.4.3. Recursive Functions of State Vectors

In this section we discuss McCarthy's method for investigating properties of programs by associating them with recursive functions.

In general, a program manipulates a number of variables. The current values of these variables constitute the so-called state vector, which will be denoted by ξ. In order to obtain a functional representation of a program, one associates it with a function σ, such that $\xi' = \sigma(\xi)$ is the state vector resulting after execution of the program for the initial state vector ξ.

We first give an explanation of some of the basic properties of state vector functions by means of examples of programs in a simple language, viz., consisting of sequences of (possibly labeled) assignment and conditional **goto** statements. Let S_1 and S_2 be two such sequences, with the restriction that they have only one entrance and exit, i.e., they neither contain labels which are referred to from "outside," nor **goto** statements referring to labels outside. Suppose that already associated with S_1 and S_2 are the functions $\sigma_1(\xi)$ and $\sigma_2(\xi)$. Then

1. With $S_1;S_2$ is associated $\sigma(\xi) = \sigma_2(\sigma_1(\xi))$.
2. With **if** p **then goto** L; S_1; L: S_2 is associated
   $\sigma(\xi) =$ **if** $p(\xi)$ **then** $\sigma_2(\xi)$ **else** $\sigma_2(\sigma_1(\xi))$.
3. With L: S_1; **if** p **then goto** L is associated
   $\sigma(\xi) =$ **if** $p(\xi)$ **then** $\sigma(\sigma_1(\xi))$ **else** $\sigma_1(\xi)$.

From the third example it follows that the state vector function is recursive in the case that the program contains a loop.

The rules given in these examples are not intended as a general scheme for the construction of state vector functions. Because of the restrictions on S_1 and S_2, they are by no means sufficient to treat all sequencing structures. In the general case the construction of the state vector function σ corresponding to a given program leads to a set of mutually-dependent recursive functions with σ as one of its elements. Some specific examples of such constructions are given by McCarthy ([99]). The general problem is considered by de Bakker ([11]), Böhm ([16]), Luckham *et al.* ([91]), and Strachey ([123]). An alternative approach, adopted by McCarthy ([101]), Kaplan ([70]), and Painter ([109]), is to extend the state vector with a statement counter, the current value of which is the number of the statement to be executed. This technique has proved useful when state vector functions are applied in the formal definition of languages (Section 3.3). However, the first solution should presumably be preferred in applications concerning proofs about individual programs.

As a further illustration of the state vector concept, we show that a special case of one of Yanov's axioms (Section 2.2.2) can be derived from rules 1 and 2 and from the axioms for conditional expressions.

Let P_1 be the program

$$\textbf{if } p \textbf{ then goto } L_1;\ S_1;\ L_1\text{: } \textbf{if } p \textbf{ then goto } L_2;\ S_2;\ L_2\text{:}$$

and let P_2 be

$$\textbf{if } p \textbf{ then goto } L_1;\ S_1;\ \textbf{if } p \textbf{ then goto } L_2;\ S_2;\ L_2\text{: } L_1\text{:}$$

Yanov's axiom asserts that P_1 and P_2 are equivalent. We prove this for the special case that S_1 and S_2 satisfy the above-mentioned restrictions, by showing that P_1 and P_2 have equivalent state vector functions σ' and σ''.

By rule 2, to the program **if** p **then goto** L_i; S_i; L_i: $(i = 1, 2)$ corresponds **if** $p(\xi)$ **then** ξ **else** $\sigma_i(\xi)$. Composition of these two yields for the function σ' of P_1:

$$\begin{aligned}\sigma'(\xi) = {} & \textbf{if } p(\textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else } \sigma_1(\xi)) \\ & \textbf{then if } p(\xi) \textbf{ then } \xi \textbf{ else } \sigma_1(\xi) \\ & \textbf{else } \sigma_2(\textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else } \sigma_1(\xi))\end{aligned}$$

Application of M_{11} gives

$$\begin{aligned}\sigma'(\xi) = {} & \textbf{if if } p(\xi) \textbf{ then } p(\xi) \textbf{ else } p(\sigma_1(\xi)) \\ & \textbf{then if } p(\xi) \textbf{ then } \xi \textbf{ else } \sigma_1(\xi) \textbf{ else if } p(\xi) \textbf{ then } \sigma_2(\xi) \textbf{ else } \sigma_2(\sigma_1(\xi))\end{aligned}$$

By M_4, M_5, and M_6 this can be reduced to

$$\sigma'(\xi) = \textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else if } p(\sigma_1(\xi)) \textbf{ then } \sigma_1(\xi) \textbf{ else } \sigma_2(\sigma_1(\xi))$$

By rules 1 and 2, to the program S_1; **if** p **then goto** L_2; S_2; L_2: there corresponds

$$\bar{\sigma}(\xi) = \textbf{if } p(\sigma_1(\xi)) \textbf{ then } \sigma_1(\xi) \textbf{ else } \sigma_2(\sigma_1(\xi))$$

Hence

$$\begin{aligned}\sigma''(\xi) &= \textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else } \bar{\sigma}(\xi) \\ &= \textbf{if } p(\xi) \textbf{ then } \xi \textbf{ else if } p(\sigma_1(\xi)) \textbf{ then } \sigma_1(\xi) \textbf{ else } \sigma_2(\sigma_1(\xi))\end{aligned}$$

We conclude that $\sigma' = \sigma''$.

A systematic investigation of the relation between the properties of conditions and state vector functions, and of Yanov's axioms, is given by de Bakker ([11]).

We have not yet discussed how assignment statements are treated. For this purpose McCarthy has introduced two functions, α and γ. Here $\alpha(x, f, \xi)$ reflects the effect of the assignment statement $x := f$ executed at the moment that the current state vector is ξ. It delivers a new state vector which is equal to ξ except for its x component, which has now become the value of the function f. The function $\gamma(x, \xi)$ delivers the current value of the variable x in the state vector ξ. The relations between the functions α and γ are characterized by the following rules:

1. $\gamma(x, \alpha(y, f, \xi)) = \textbf{if } x{=}y \textbf{ then } f \textbf{ else } \gamma(x, \xi)$.
2. $\alpha(x, \gamma(x, \xi), \xi) = \xi$.
3. $\alpha(x, f, \alpha(y, g, \xi)) = \textbf{if } x{=}y \textbf{ then } \alpha(x, f, \xi) \textbf{ else } \alpha(y, g, \alpha(x, f, \xi))$.

As an example of the application of these rules, we consider the equivalence $x{:=}y;\, y{:=}x \sim x{:=}y$ (axiom Ba_1 of Section 2.3.2). With $x{:=}y$ is associated $\alpha(x, \gamma(y, \xi), \xi)$. By the given rules this is equivalent to $\alpha(y, \gamma(x, \alpha(x, \gamma(y, \xi), \xi)), \alpha(x, \gamma(y, \xi), \xi))$, i.e., to the function associated with $x{:=}y;\, y{:=}x$.

The three rules are studied in detail by Kaplan ([71]). An axiomatic theory is developed with these rules as axioms, to which some rules of inference are added. A formalism is introduced for interpreting the well-formed formulas of the theory; moreover, it is defined what it means for a formula to be true in a given interpretation. Then the following completeness theorem is proved: A formula is a theorem of the axiomatic theory if and only if it is true in all interpretations.

2.4.4. Recursion Induction on State Vector Functions

As a final example of McCarthy's work, we discuss an application of the combination of the ideas from the two previous sections. Consider the following two programs P_1 and P_2, with corresponding functions σ_1 and σ_2:

P_1 is

L_1: **if** $n{=}0$ **then goto** L_2; $s{:=}n \times s$; $n{:=}n-1$; **goto** L_1; L_2:

P_2 is

L_1: $s{:=}n! \times s$; $n{:=}0$; L_2:

Let $\bar{\sigma}$ be the function corresponding to $s{:=}n \times s$; $n{:=}n-1$. (Note that $\bar{\sigma}$

can be expressed in terms of α and γ.) Then for σ_1 we have

$$\sigma_1(\xi) = \textbf{if } n{=}0 \textbf{ then } \xi \textbf{ else } \sigma_1(\bar{\sigma}(\xi)) \tag{6}$$

From the properties of α and γ, and the definition of the factorial, it follows that if $\gamma(n, \xi) = 0$, then $\sigma_2(\xi) = \xi$, and if $\gamma(n, \xi) \neq 0$, then P_2 is equivalent to

$$L_1\text{: } s{:=}n{\times}s;\ n := n-1;\ s{:=}n!{\times}s;\ n{:=}0;\ L_2\text{:}$$

(For this equivalence see also an example of Section 2.3.2.) We conclude that

$$\sigma_2(\xi) = \textbf{if } n{=}0 \textbf{ then } \xi \textbf{ else } \sigma_2(\bar{\sigma}(\xi)) \tag{7}$$

Comparison of Eqs. (6) and (7) yields, by recursion induction, that $\sigma_1 = \sigma_2$.

McCarthy has also introduced a method for directly applying recursion induction to programs, i.e., omitting the intermediate use of state vector functions. He in fact applied it to obtain the equivalence given above of P_1 and P_2 (99). Since we have not discussed this method, we had to use state vector functions in the derivation of $P_1 \sim P_2$.

2.5. Flow Diagrams

2.5.1. General Properties

Ever since the first years of computing, flow diagrams have been used for the representation of programs (their use goes back to Goldstine and von Neumann (60)). As is well known, they are especially suitable for providing an overall picture of the global properties of a program—in particular, with respect to its sequencing structure.

Although flow diagrams have been used for practical purposes for a long time, theoretical investigation of their properties has started only relatively late. The first treatment which might be called abstract of flow diagrams seems to be due to Kaluzhnin (69) (see also Fels (56)). Kaluzhnin introduces the notion of a graph schema, defined as follows:

Let there be given a set $A = \{A_1, A_2, \ldots, A_n\}$, the elements of which are called operators, and a set $F = \{F_1, F_2, \ldots, F_m\}$, the elements of which are called discriminators.

A finite, labeled, directed graph (Berge (14)) is called an A–F-graph schema if it satisfies the following conditions:

1. There is precisely one node, called the entrance, to which no arrow leads and away from which exactly one arrow leads.

2. There is precisely one node, called the exit, away from which no arrow leads.

3. With the exception of the entrance and exit nodes, from each node of the graph there leads either one or two arrows. In the first case it is called an operator node, and is labeled by an element of A. In the second case it is called a discriminator node, and is labeled by an element of F. The arrows leading away from a discriminator node are marked, e.g., by 0 and 1.

An interpretation of an A–F-graph schema is defined by the selection of a domain D, and by interpreting the elements of A as functions from D to D and the elements of F as functions from D to $\{0, 1\}$. An interpreted graph schema defines a partial function from D to D in the usual way: Starting with the entrance node a path is followed through the graph in the arrow direction. When an operator is met the function corresponding to it in the given interpretation is evaluated for the argument considered (the result becomes the argument for the next function) and the arrow leaving this node is followed. A discriminator node determines a choice from the two arrows leading from it. It selects the arrow marked 0 (1) if the value of the corresponding function for the argument considered is 0 (1). The evaluation of the function determined by the graph schema terminates, if ever, when the exit node is reached. (A more formal description of the evaluation of the function determined by an interpreted graph schema may be found in the paper by Kaluzhnin (69) and in most of the papers to be discussed presently (13,51,107,126).

After the definition of graph schemata some of their properties are studied. A rule is given for obtaining, for each Markov algorithm, an interpreted schema which defines the same function. The graph-schema formalism provides a convenient means of defining substitution or composition operations—this in contrast, e.g., with Markov algorithms. The substitution of a given graph schema for an operator node A_i in another schema is defined as follows: Replace each occurrence of A_i in the second schema by the graph which results after deleting the entrance and exit nodes of the first schema. Note, however, that substitution cannot be defined in the same way for discriminator nodes.

We have given a rather detailed account of the main ideas of Kaluzhnin (69) because they return in some form in many of the other papers on properties of flow charts, with the discussion of which we now proceed.

As already mentioned, an important application of flow charts is their use in the study of the sequencing concept. It should be clear, however, that there is no essential difference between the representation of the sequencing structure of a program by means of a flow diagram, or a linear notation based on **goto** statements. (On the other hand, the repetitive effect of the recursive use of procedures is in general not directly representable by means of a flow diagram. This question is discussed by Cooper ([38]) and McCarthy ([99]).) Which of the two approaches—sometimes characterized as geometric versus algebraic—is preferred is mainly a matter of convenience. For the investigation of the global properties of a sequencing structure a flow-chart representation may be more appropriate, whereas local aspects—e.g., considered in the axiom systems of Yanov and Igarashi—are more concisely representable in terms of a linear notation.

The close connection between properties of flow charts and of **goto** statements implies that a number of investigations treated in the previous sections may just as well be viewed as dealing with flow charts. This holds, e.g., for Yanov's work, the equivalence results of which are immediately applicable to graph schemata, or for the results on the formalized computer programs of Luckham *et al.* ([91]), the operator nodes of which must be interpreted as assignment statements. In addition, flow charts were in fact used by McCarthy ([99]) in his explanation of the association of state vector functions with programs.

There is a fairly extensive literature dealing with flow charts, with varying degrees of relevance for the semantics of programming languages. No attempt at completeness will be made in our discussion of them, but we shall indicate briefly the various directions which may be distinguished in these investigations, and give some representative references.

First we mention some papers which are concerned with the relation between graph schemata and notions from logic—in particular, computability theory. Péter ([111,112]) gives the first proof of the equivalence of the class of functions defined by means of graph schemata and that of partial recursive functions. Asser ([6]) proves the equivalence of graph schemata with his function-algorithms ([7]). Kunze ([76]) studies a certain extension of graph schemata which allows operations on only a part of the object being transformed. Another proof of the equivalence of graph schemata with partial recursive functions is given by Ershov ([51]). The same equivalence result is again proved by Basu ([13]), who also gives an algorithm for the transformation of flow diagrams to a linear notation. Thiele ([126]) considers (extended) graph schemata, the operator and discriminator nodes of which are interpreted by functions and predicates as studied in the first-order predicate calculus, and investigates to what extent the results of the predicate calculus can be extended to these structures. Our judgment on the relevance of the sort of results meant here is twofold. On the one hand, they are not as yet directly applicable to semantics; the remarks of the first part of Section 2.1.2 are largely pertinent here. However, graph schemata are much more suitable for the representation of properties of programs than, e.g., Turing machines. Hence it may well be that further development of the logical investigations of graph schemata will lead to results which are indeed interpretable as results on programming concepts. Clearly, major progress in semantics would be achieved, if, e.g., the work of Thiele were extended in such a way that theorems of the predicate calculus could be interpreted in some manner as theorems on properties of programs.

In a second group of papers properties of graph schemata are investigated by means of graph theory. Examples are Karp ([72]), who uses graph theory to determine redundancies in programs (detection of nodes which cannot be reached from the entrance node, or from which the exit node cannot be reached), and Schurmann ([117–119]), who gives algorithms to determine the number of certain cycles in graphs, which cycles correspond to loop structures in programs. Use is made in these papers of the connection matrices of the graphs considered. This is also done by Krider ([75]) and Hain and Hain ([62]), who are mainly concerned with the actual drawing of flow charts.

In our opinion, the purely graph-theoretic approach to flow-chart investigation is useful only for a limited class of problems. In order to obtain deeper results, the graph structure must be investigated together with the properties of the predicates associated with its discriminator nodes, and with the effects of the operators upon these predicates, which effects determine the flow of control. For most purposes the graph structure alone is too poor a model of the flow diagram.

Probability aspects of graph models of computations are studied in a number of papers, e.g., ([93–95,113]). These investigations are only remotely related to programming concepts (an exception is, of course, the notion of parallelism, which is, however, not considered in the present chapter), especially since many of them pertain also to operating systems, e.g., in a timesharing environment. Discussion of them is therefore omitted.

Narasimhan ([107]) is more directly concerned with programming languages. An extensive formalism is introduced with flow diagrams as basic components. Rules are given for the substitution of a flow diagram for nodes in another diagram. These substitutions may be considered as subroutine calls. Compared with other work, an extension is introduced which amounts to the treatment of subroutines with parameters. The formalism also allows parameters referring to other flow diagrams. For such nested calls of subroutines, the term hierarchical computation is used. The system is envisaged as a general framework in which properties of languages and computers can be phrased. In fact, the paper even purports to develop "a unified metatheory of programming languages and computers." This highly ambitious goal has certainly not been achieved. All that can be said is that the proposed formalism may be of some use for the investigation of the transfer of information between different flow diagrams, although this will have to be borne out by further elaboration.

A more interesting paper is that by Böhm and Jacopini ([18]). The problem is considered whether it is possible to decompose each flow diagram into a finite number of base diagrams. By means of a counterexample it is shown that this is not the case. It is necessary to introduce an extension of the flow-chart formalism which amounts to the following: Whenever an argument is subjected to a test by one of the discriminator nodes of the flow chart it is supplied with an indication of the result of this test. A mechanism for inspecting or deleting these indications is also introduced. It is then shown how, in the extended formalism, each flow diagram can be decomposed using either three or two base diagrams. Related is the work of Cooper([40]). Meaning-preserving transformations of graphs are introduced, and necessary and sufficient conditions are given which must be satisfied by a graph to allow reduction to some normal form by means of these transformations. A comment by Cooper on the relation of his work to that of Böhm and Jacopini is given in ([41]). In order to clarify the relation between the results of Böhm and Jacopini ([18]), Cooper ([40]) and various approaches discussed in the previous sections (e.g., is it possible to express these results as properties of the functions associated, in McCarthy's sense, with programs?) we expect that the method of Floyd ([58]), to be treated in the next section, will be useful.

2.5.2. *Floyd's Method*

Flow diagrams are used as a tool for assigning meaning to programs in an important paper by Floyd ([58]).

Consider a flow diagram with commands associated with its nodes. (The distinction of the previous section between operator and discriminator nodes is not made here; commands include both cases.) With each constituent arrow of the flow diagram a proposition is associated which states the condition to be satisfied by the variables manipulated by the program (i.e., the state vector) in order that the flow of control take the arrow concerned (similar to these propositions are the "general snapshots" proposed independently by Naur ([108])). A verification condition for a command is a relation which holds between the propositions associated with the incoming and outgoing arrows of the command, respectively.

This relation is to be defined such that if it is satisfied, and if the proposition associated with the arrow along which the command is entered is true, then the proposition associated with the arrow from which the exit, after execution of the command, is taken, is also true.

An axiomatic treatment is given of the general requirements which must be met by the verification conditions in order to obtain a complete and consistent theory. These requirements are illustrated by the definition of the verification conditions for the statements of a particular flow-chart language. It is shown for instance that for the assignment statement $x := f(x, y_1, \ldots, y_n)$ the condition must have the following form:

'Let $P(x, y_1, \ldots, y_n)$ and $Q(x, y_1, \ldots, y_n)$ be the propositions associated with the incoming and outgoing arrows of this statement, respectively. Then the verification condition is: If there exists x_0 such that $P(x_0, y_1, \ldots, y_n)$ holds and such that $x = f(x_0, y_1, \ldots, y_n)$, then $Q(x, y_1, \ldots, y_n)$ holds.

Next, verification conditions for some typical ALGOL-like commands, such as conditional, **goto** and **for** statements, are derived, and the locality aspect of declarations is treated. A method for dealing with assignment statements inside expressions is given—which amounts to the introduction of a processor with a pushdown stack—as an illustration of the side-effect feature of procedures.

As a final application, a technique for proofs about termination of programs is proposed: Associate with each arrow of the flow chart—besides the already-mentioned propositions—also a state vector function with values in a well-ordered set. Then show that for each command the value of the function associated with the incoming arrow is greater than the value of the function associated with the outgoing arrow, where the two arrows are such that their associated propositions satisfy the verification condition for this command.

2.6. *Concluding Remarks*

In this section we make a few concluding remarks on the research on basic programming concepts discussed in the preceding sections. For this purpose we consider again the list of Section 2.1.1. It appears that the majority of the investigations concentrates upon concepts 4, 5, and 7, i.e., conditional constructions, sequencing, and assignment. A rearrangement of the references dealing with one more of these concepts may be useful: assignment ([10,21,71]); conditions ([9,23,25,44,114,130]); assignment and conditions ([106]); conditions and sequencing ([11,16–18,38–40,47,50,52,53,64,116,135–138]); and assignment, conditions, and sequencing ([20,58,63,65–67,77–81,91,98,99,104,105, [108,123,125]) (a few references have been included in this scheme which were not yet mentioned, but are to be treated in Section 3).

If one bears in mind that, apart from a few exceptions, all these papers use different methods and formalisms, one will appreciate the remark of Section 2.1 concerning the difficulty of incorporating the various approaches into one unified system, allowing a systematic exposition of the interrelationships of the proposed methods and the results obtained. It seems likely that decisive progress will be achieved only if a number of unifying notions will have been found, having the same effect on semantics as, e.g., the theory of phrase-structure grammars has had on syntax. However, although we feel that a complete synthesis is not within direct reach, there are a number of less ambitious goals for future research which may contribute towards unification, and which seem not too difficult. Possible candidates for such investigations are:

1. A study of the relation between the various axiomatic characterizations of assignment.
2. An analysis of the relation between the functional approach to sequencing (*via* systems of recursive functions), and the imperative approach (*via* **goto** statements).
3. An analysis of the relation between the algebraic properties of sequencing (axioms of Yanov and Igarashi), and the various methods based on flow charts.
4. More abstractly (and more difficult), an investigation of the different notions of equivalence between programs and the corresponding equivalence-preserving transformations.

3. FORMAL DEFINITION OF PROGRAMMING LANGUAGES*

3.1. Introduction

After having given in Section 2 a survey of the investigations of basic concepts in programming, we devote Section 3 to a discussion of the research which has been done on complete programming languages. Most of this research is concerned with the formal definition of (syntax and) semantics of programming languages.

* Section 3 owes much to our participation in the discussions of the IFIP Working Group 2.2 on formal language description languages. However, all opinions expressed in this section are our own and do not necessarily reflect the opinions of the Group.

The work reported in the previous section was inspired mainly by the wish to investigate the mathematical properties of concepts in programming and to obtain proofs about these properties. On the other hand, research in methods for formally defining languages is motivated primarily by other reasons of a more practical nature. Of these we mention the following:

1. First of all, the wish to provide the compiler writer with a complete, precise, and unambiguous definition of the language for which he must construct a compiler. Such a definition should, for instance, make it clear which parts of the language are not fully specified, so that the implementor knows where he may choose his own interpretation. As an example of an implementation-dependent feature of most programming languages, take real arithmetic, the implementation of which will differ with the various machines for which it is intended. However, the formal description may well state some basic requirements which must be satisfied by all implementations.

2. A formal description method for languages can also be of use in their design. It should lead to a vocabulary for discussions about concepts in the language. One might expect of it the detection of incompatible, contradictory, or ambiguous constructions, or it might be used as a source of inspiration for new concepts which would not have originated directly from practical considerations. These applications of a formal description method will in particular arise when a new language is designed on the base of an already existing method. For instance, the design of ALGOL 68 has been influenced by the previously developed method for its syntactic description (cf. Section 3.6); in addition, there has been much interaction between the design of CPL and the theoretical investigations of semantics by Landin, Strachey, and others (Section 3.5).

3. People who want to write or understand programs may want to consult the formal definition of the language in those cases where their usual reference document does not provide a sufficiently clear answer to their problem. Experience has shown that such situations, where the manual which describes the language does not give satisfactory information, often occur.

4. A formal definition of a language may be used for standardizing this language. A need for the availability of a formal definition method has been expressed several times by people who are concerned with the standardization of programming languages, e.g., by Steel ([122]).

5. Comparison of different languages may be facilitated when they have been described formally using the same method. It should then be possible to establish which concepts in the languages are essentially the

same, and for the remaining ones what relationships, if any, they have with each other.

6. Finally, one might expect of a formal definition of a language that it can be used to give proofs about properties of the language; one may distinguish here among proofs about general concepts in the language, about individual programs, and about compilers for it. This application is usually considered to be of less importance than the ones mentioned above, since it is recognized that more or less complete descriptions of full languages are too complicated to be used as a tool for giving proofs. However, some systems for formally describing languages have indeed been used for proofs, e.g., on the correctness of a small compiler (Section 3.3).

After these introductory remarks on possible reasons for formal language definition we shall devote the remainder of this chapter to a survey of the several systems which have been used up to now. The book edited by Steel ([121]), which contains the proceedings of a conference on formal language description languages held in 1964, may serve as a further introduction to the field of language definition. The principles of many of the systems currently in use are presented in this book (an important exception is the Vienna work on the definition of PL/I). It also contains, especially in the discussions, a great deal of information on how the various authors motivate their approaches.

3.2. The Markov-Algorithm Approach

Markov algorithms have become well known in programming. They were introduced by Markov ([92]) (for an introductory exposition see Mendelson ([103])) for the investigation of problems in computability theory, mainly leading to theorems on undecidability, and may be compared from this point of view, for instance, with Turing machines. However, the transformation scheme as present in Markov algorithms has found several applications in practical programming, the first of which seems to be due to Yngve in his design of COMIT ([139]). Most of these applications are in the field of languages for symbol manipulation, for which we refer to Bobrow ([15]) (see also Christensen ([35]) and Itturiaga ([68])).

Obviously, Markov algorithms need extensions with other concepts in order to be useful for practical purposes. For instance, in many cases some **goto** mechanism is added. Other extensions will appear presently in our discussion of the use of Markov algorithms for formally defining languages.

The starting points for the use of Markov algorithms for language definition are the papers of van Wijngaarden ([131,132]), Caracciolo ([26]), and

Caracciolo and Wolkenstein [34]. It was noticed only afterward that these papers had in common the introduction of an essentially similar extension to the Markov-algorithm concept, which may be summarized as follows: In the ordinary Markov algorithm one has transformation rules, the left- and right-hand sides of which are sequences of symbols over some given alphabet. In order to establish whether a rule is applicable to a given sequence, the sequence is scanned for the occurrence of a sub-sequence which is identical to the left-hand side of the transformation rule. If such a sub-sequence does occur, then its first occurrence is replaced by the right-hand side of the rule concerned. In the extended version the transformation rules consist in general not only of symbols from the given alphabet (to be called terminal symbols), but also of metalinguistic variables, for instance, in Backus notation. An example of such an extended rule is

$$\langle\text{unsigned integer}\rangle\ 0 + \langle\text{digit}\rangle \rightarrow \langle\text{unsigned integer}\rangle\ \langle\text{digit}\rangle \tag{8}$$

The corresponding extension of the concept of applicability, and of the transformation determined by an applicable rule, is then: Consider a transformation rule and a terminal sequence. The rule is applicable to the sequence considered if its left-hand side satisfies the following condition: There exist productions of the metavariables occurring in it such that the sequence which results after substituting these productions for these metavariables is identical with a sub-sequence of the considered terminal sequence (In case there is more than one possibility for such productions, the first one in a suitably defined order is chosen.) This sub-sequence is then replaced by the sequence which results from the right-hand side when the same substitutions of productions for metavariables have been performed there as in the left-hand side.

Example: Rule (8) is applicable to the sequence $210 + 5$, provided that the usual definitions of ⟨unsigned integer⟩ and ⟨digit⟩ are available, and it transforms this sequence to 215.

In this description of the proposed extension of the Markov algorithms [26,34,131,132] (essentially the same system was proposed subsequently by Cohen and Wegstein [37]) some modifications have been introduced in order to bring out the common features. We now treat some other ideas of these papers.

Caracciolo [26] and Caracciolo and Wolkenstein [34], instead of using metalinguistic variables in the transformation rules, used in fact a somewhat more general approach: The rules may contain names of arbitrary sets, provided that it is effectively decidable for each given sequence of symbols whether it belongs to this set or not. Moreover, a further extension was considered in which the right-hand side of a transformation rule does not simply determine a replacement—in case of applicability of the rule to a given sequence—but in general the application of some recursive function to this sequence. (Essentially the same extension of Markov algorithms was studied previously by Asser and Vuckovič (7).)

In order to establish whether a transformation rule is applicable to some sequence, it may be necessary to determine whether some part of this sequence is a production of a metavariable occurring in the left-hand side of the rule. In Caracciolo's system a separate set of formulas is supposed to be given for this purpose. On the other hand, in van Wijngaarden's work (131,132) these questions are settled by consulting the same list of transformation rules, i.e., these rules contain all relevant "syntactic" information. Details of the precise way in which this is done are omitted here.

We now consider some applications of the systems of van Wijngaarden and Caracciolo to language design and definition.

Van Wijngaarden (131) was concerned both with the design of a language called generalized ALGOL and with its formal description. Some ideas on this generalization of ALGOL were taken up by Wirth (128) and Wirth and Weber (129), although in the latter a completely different method for formal definition was used. In van Wijngaarden (132) the emphasis was laid on the formal definition of ALGOL 60. The ideas of this paper were used as the base for our de Bakker's (8), where an almost complete definition of ALGOL 60 was given (the only feature not treated being real arithmetic), consisting of about 800 transformation rules. The meaning of an ALGOL 60 program is determined by the way in which it is transformed by these rules. Here another extension of the Markov-algorithm scheme not yet discussed is of importance, viz., the possibility of having a dynamically growing list of rules. The execution of a particular ALGOL 60 program will lead to the extension of the list of language-defining rules with rules which reflect the meaning of this specific program. For instance, the occurrence of the assignment statement $a := 3$ in a program causes the creation of a new rule $a \to 3$ (omitting some details on locality), which will be applied each time the value of a is needed subsequently in the execution of the program. De Bakker (8) also gives a precise definition of the formal system used, illustrated by several examples, and an implementation of an abstract machine for interpreting it.

The system of Caracciolo has been applied to a large class of problems. However, many of these have been described only in reports as yet unpublished. The list of applications includes:

1. The design of a language for symbol manipulation, called PANON I B (27,33,34).

2. The definition of a storage allocation mechanism for FORTRAN, concerning the notions of COMMON, DIMENSION, and EQUIVALENCE, by Aguzzi and Pinzani (1).

3. The formal definition of a machine tool language by Caracciolo and Camera (29) (also see Caracciolo (24)).

4. The definition of the record and file manipulation in COBOL (84).

5. The definition of ALGOL 60 (31). The main difference between this and de Bakker (8) is that Caracciolo's system does not include the idea of a growing list of rules. The relevant information originating during the execution of the program, i.e., the sequence being transformed, is kept available by including it in some way in this sequence. Moreover, the treatment of **goto** statements and locality is simpler than that in de Bakker's system (8).

6. The definition of SIMULA (32).

7. The definition of the de Bakker's formal system (8) and some principles of the system which has been used for the formal definition of PL/I, (Section 3.4) by Caracciolo (28) and Caracciolo and Carlucci (30).

3.3. McCarthy's Ideas on Formal Definition

McCarthy's main paper on the formal definition of languages was given at the 1964 IFIP Working Conference (101) (the basic ideas were already described in the last part of a paper to the the 1962 IFIP Congress (99); some comments were added in a paper to the 1965 IFIP Congress (100).

The first important concept of McCarthy's system is that of abstract syntax, as opposed to the usual notion of concrete syntax. The concrete syntax of a language, e.g., given by a context-free grammar in Backus notation, prescribes which sequences of symbols constitute valid constructions in the language. The abstract syntax, on the other hand, makes no commitments about the way in which such constructions are represented by sequences of symbols, but for each type of construction names of a predicate and of functions are given, where the predicate is such that it is true precisely for a construction of the given type, and the functions are used for decomposing the construction in its relevant parts. For instance, in an abstract syntax one is not interested in whether an infix or prefix notation is used in binary arithmetic expressions; it is only necessary to be able to recognize it as an arithmetic expression and to have functions for obtaining its operator and its first and second operands.

Abstract syntax is especially useful in combination with McCarthy's proposal for defining the semantics of a programming language: This is done by means of a state vector function (Section 2.4.3), called "lang," say, such that $\xi' = \text{lang}(\pi, \xi)$ gives the state ξ' which results from applying the program π to the state ξ. The crucial point here is to decide what information should be included in the state vector. In Section 2.4.3 it was taken to be the set of current values of the variables occurring in the program. In McCarthy ([101]) and in the papers to be discussed presently it also includes a statement counter, the current value of which is the number of the statement to be executed. These components are sufficient for the definition of languages containing only some simple concepts such as arithmetic and Boolean expressions, assignment and **goto** statements. For the treatment of richer languages, which also include concepts as locality, procedures, declarations, etc., many more components must be introduced, as was done, e.g., in the system for the formal definition of PL/I (Section 3.4).

As an illustration of the use of state vector functions for formal definition, we consider the treatment of the assignment statement. Let the predicate assignment(s), and the functions left(s) and right(s) be available in the abstract syntax, and let *sn* be the name of the statement counter, with the current value *n*. Furthermore, we assume that a function value (t, ξ) has already been defined, delivering the value of the expression *t* for the state ξ. The meaning of an assignment statement, using the function α of Section 2.4.3, is then described by:

...**if** assignment(s) **then**

$\alpha(sn, n+1, \alpha(\text{left(s)}, \text{value(right(s)}, \xi), \xi))\ldots$

The formalism sketched above, though not yet elaborate enough for the definition of complete languages, has found interesting applications in proofs about compilers, e.g., by McCarthy and Painter ([102]), Painter ([109]), and Kaplan ([70]). The former paper ([102]) can be considered as preparatory to the latter two ([70,109]), where a proof is given of the correctness of a compiler for a language including arithmetic and Boolean expressions, assignment, conditional, and **goto** statements, and some I/O. The notion of a correct compiler is defined as follows: First the semantics of the source language and the object language (of an idealized machine with a small set of instructions resembling actual assembly operations) is given by the method described. In order to construct the object program produced by the compiler, it is necessary to also provide a synthetic (abstract) syntax of the machine language, i.e., a set of rules for composing a program, as opposed to the analytic abstract syntax which gives a means for taking a program apart. The connection between the analytic and synthetic syntax of a language is defined by certain "regularity conditions." Next a function is postulated which establishes a 1:1 mapping of the variables of the source program onto a set of registers used in the object program. Then the compiler is correct if for each source program, the corresponding object program produced by it satisfies the following requirement:

Let the variables of the source program and the registers corresponding to them by the postulated function be assigned the same initial values. Then the final values of the variables in the source program and the registers of the object program are the same, these values being obtained by application of the semantic function of the source language and object language, respectively.

The main tool in the proofs is recursion induction or some of its variants. The proofs by both Painter ([109]) and Kaplan ([70]) are long and complicated. In our opinion, they provide a good illustration of the need for some fundamental theorems in the theory of semantics which would make it unnecessary to start from scratch, as it were, every time one wants to give a proof in this field.

3.4. The Vienna Method

One of the major achievements in the area of language definition is the formal definition of PL/I, as given by the PL/I definition group of the IBM Laboratory in Vienna.

The method used, although developed with view to PL/I, consists in fact of a number of concepts quite generally applicable. Hence it can also be used with other languages; this has been illustrated by employing it for the definition of ALGOL 60.

The principles of the method are explained by Lucas *et al.* ([90]). It is based on the definition of an abstract machine which is characterized by the set of its states and its state transition function (cf. Elgot ([48]) and Elgot and Robinson ([49])). A given program defines an initial state of the machine; the subsequent behavior of the machine is said to define the interpretation of the program. To be more precise, let Λ be the state transition function and ξ_0 the initial state. Then the behavior of the machine is the sequence $\xi_0, \xi_1, \ldots, \xi_i, \xi_{i+1}, \ldots$, with $\xi_{i+1} = \Lambda(\xi_i)$. The sequence terminates, if ever, if an element of a given set of final states is reached.

The interpreting machine is used in this way to attach a meaning to programs and their constituent expressions. However, it proved to be in-convenient to have the machine operate directly upon the sequences of symbols which constitute these expressions, as prescribed by the syntax. To circumvent this difficulty, an intermediate stage has been introduced. McCarthy's abstract syntax is used to define programs as abstract objects, and these abstract objects are the entities manipulated by the interpreting machine. Hence, before the machine can be applied to a given program text, i.e., a sequence of symbols produced by the concrete syntax, the text must first be translated into the corresponding abstract object. For PL/I this translation is described by Alber and Oliva (3); a general discussion of the relation between abstract and concrete syntax can be found in the paper by Lucas *et al.* (90).

Next we explain some features of the formalism used for the description of the operations of the interpreting machine. A general class of objects is introduced, of which both the abstract objects representing programs and the states of the machine are subclasses. These objects may be considered as tree-structured entities: An object is either elementary or it is composed of a finite number of immediate components, each of which is again an object. The immediate components of an object are named by means of selectors. The application of a selector to a nonelementary object yields the immediate component with this selector as its name. To a nonelementary immediate component again a selector can be applied, etc. In this way, by successive application of selectors, all components of a given object can be obtained.

Several operations on objects and their selectors are defined. The most important of these is the so called μ-operator. Given an object A and a composite selector (i.e., the functional product of a number of selectors) and an object B, application of μ results in a new object, viz., A, where the component to which the composite selector points has been replaced by B.

Some general schemes for the definitions of subclasses of the class of objects are introduced, e.g., for the definition of objects, the immediate components of which satisfy certain predicates. These schemes can be used, e.g., in the definition of the abstract syntax of a language.

The formalism for dealing with general objects is then applied in the description of the properties of the interpreting machine. It was already mentioned that the states of the machine are special cases of these objects. Each state has a so-called control part as one of its immediate components. The control part of a given state is used by the machine to determine the operations to be executed in order to obtain the successor state of this state, i.e., it contains the relevant information for the state transition function. (During the execution of a program the situation may arise that one has to perform a set of actions in unspecified order—e.g., the order of evaluation of primaries in an expression is not defined in ALGOL 60; in order to be able to describe such cases, the state transition function has been extended in that it does not, in general, define one successor state, but a set of successor states; hence, we have in fact a nondeterministic machine.)

The interpreting machine is illustrated by Lucas *et al.* ([90]) by applying it to the definition of a simple language with some arithmetic, conditions, assignment, procedures, and blocks. In addition to the control part, several other state components are necessary to deal with these concepts. We mention: The environment component, which is used to associate identifiers with unique names (this is necessary to treat the scope problem), and the dump component, which has a stacklike structure and is used to represent the dynamic nesting of blocks, procedure, and function activations.

The short explanation given above of the general principles of the Vienna method must suffice here; for further information we refer to the paper by Lucas *et al.* ([90]).

As already noted, the most important application of the method is the definition of PL/I. This is described in a number of reports. The main document is by Walk *et al.* ([127]), and gives the abstract syntax and interpretation of PL/I. The translation of concrete PL/I programs into abstract programs is described by Alber and Oliva ([3]) (for the concrete syntax of PL/I see Alber *et al.* ([4])). The PL/I compile time facilities are not considered by Walk *et al.* ([127]) but are treated separately by Fleck and Neuhold ([57]). An informal explanation of the paper of Walk *et al.* ([127]) is given by Lucas *et al.* ([89]).

The Vienna group has also started to exploit the formal definition of PL/I in investigations on properties of the language, especially concerning problems of implementation. Lucas ([87]) considers two interpretations of the PL/I block concept, one based on the environment mechanism of Walk *et al.* ([127]) and the other based on a chaining mechanism ([5]), and proves the equivalence of these two interpretations. A further study in this area has resulted in the detection of a deeply hidden error in a PL/I compiler ([88]).

Finally, we mention reports ([82,83]) in which the Vienna method is applied to the formal definition of ALGOL 60.

3.5. The λ-Calculus Approach

Several authors have based their investigations on Church's λ-calculus ([36]). Its introduction into programming is due to McCarthy ([96]), who did not, however, further pursue its use in his theory of computation.

Semantic theories in which the λ-calculus does play a central role have been developed by Landin ([77–81]) and Strachey ([123,125]). Related is the work of Böhm ([16]) and Böhm and Gross ([17]), who use both the combinatory logic of Curry (Curry and Feys ([42]) or Rosenbloom ([115])) and the λ-calculus in the system CUCH. We shall now give a short explanation of some of the main ideas of these authors.

In the first paper cited ([77]) Landin uses the λ-calculus for modeling expressions (as opposed to statements) occurring in programming languages. To be more precise, he observes that an expression is usually constructed from its components in three ways: by forming λ-expressions (e.g., in the case that the expression contains bound variables, or when an auxiliary function is used in its definition), by forming an operator/operand combination, or by forming a list of expressions. In order to investigate this general structure of expressions, the notion of "applicative expression" is introduced, and an abstract machine is described for evaluating applicative expressions in a given environment. Similar problems are considered by Burge ([19]).

In the next paper cited ([78]) (to which Landin's paper ([80]) at the 1964 IFIP Working Conference is an introduction) Landin also takes the imperative features of languages into account. Jumps are taken care of by treating them—apart from one important difference—as procedure calls. To deal with assignment, both the notion of applicative expression and the structure of the evaluating machine are extended. The system is applied to a definition of ALGOL 60 semantics, essentially by exhibiting how to model constructs of ALGOL 60 by means of extended applicative expressions. A formal description of the correspondence is also given; "abstract ALGOL" is introduced as an intermediate step, and a set of formulas is given to translate this both into concrete ALGOL 60, i.e., sequences of symbols, and into extended applicative expressions. Landin applies his formal system in a discussion of alternatives to various ALGOL 60 concepts, e.g., regarding its parameter mechanisms (call by name or value versus call by reference) and some variations on the own concept.

Strachey's work on semantics ([123,125]) has been developed in parallel with the design of the programming language CPL ([12,22,124]). The main difference with Landin's system is that he does not propose any extension of the λ-calculus, but an (as yet only informally described) method for mapping the constructs of the language into pure λ-notation. Central concepts in Strachey's system are the "left-hand value" and "right-hand value" of an expression. These terms correspond to the values of the left part (i.e., the address) and right part of an assignment statement, but they are generalized and also used in other situations, such as for the specification of parameters. An abstract store is used, which is a function from left-hand values to right-hand values. A method is then described of associating (compositions of) functions with (sequences of) commands. In the general case, when a loop structure is present in a sequence of commands, the association will lead to a set of mutually-dependent recursive functions.

Some extensions of Strachey's ([123]) work have been investigated by Burstall ([20]), who has also made a detailed study of assignment ([21]), partly based on Landins work ([81]).

Finally, we add a few remarks on the CUCH system. Böhm and Gross ([17]) give an explanation of its principles, whereas Böhm ([16]) is more concerned with applying it to the description of concepts in programming. CUCH is introduced as a language, the expressions of which allow different interpretations, with the provision, however, that the same interpretation be given to expressions which are convertible into each other (in the sense of the λ-calculus). Some possible interpretations of CUCH expressions are then proposed, partly dealing with notions which are often taken for granted in other systems, such as integer arithmetic (in this respect CUCH may be considered to be complementary to, e.g., the work of Landin), but also with more advanced concepts, such as the representation of flow diagrams by systems of functions.

3.6. *Other Methods*

In this section we deal with some other methods which have been used (or proposed) for formal definition.

First of all we consider the definition of ALGOL 68 ([134]). Clearly, the method applied here is not as completely formal as those discussed in the previous sections, since use is made in it of the English language. However, it is also clear that the definition is considerably closer to a completely formal definition than, e.g., that of ALGOL 60. As to the syntactic part of the description, the formal system used is much more powerful than Backus notation; thus, less English is needed here (and much richer structures can be defined). To be more precise, whereas in Chomsky's classification Backus notation corresponds to grammars of type 2, the system used for the definition of the syntax of ALGOL 68, viz., the van Wijngaarden grammars, is of type 0 ([120]), i.e., it has the same power as Turing machines or their equivalents. (From this it follows that all use of English in the definition of the syntax, as present, e.g., in the definition of the class of "proper programs" by means of the context conditions, might have been avoided.

However, this possibility is only of theoretical interest; a fully formalized syntax would be very large and difficult to read.)

In the definition of the semantics of ALGOL 68 no (explicit) formal system, but only English, has been used. A hypothetical computer is introduced, and the meaning of the various constructions of the language is defined by stating which actions it performs in their "elaboration." Since English is used in these definitions in a very precise and rigorous way, they may to some extent be viewed to be of a formal, though not symbolized, nature. However, in order to be able to apply this "formal" system to other languages, it is necessary to determine first which of its features are independent of the specific properties of ALGOL 68. This separation of concepts of the describing formalism from concepts of the language described must be awaited before the method can be considered as a candidate for the formal description of other languages.

An axiomatic approach to formal definition has been proposed by Hoare ([63]). He first gives some general arguments in favor of such an approach, and then illustrates it by an axiomatic definition of several basic features of programming languages, such as integer and real arithmetic (cf. van Wijngaarden ([133])), expressions, procedures, assignment, and jumps. Of this general arguments we mention the following: The definition of constructs in a language by means of a set of axioms may be considered as an implicit one, stating only their essential properties. This is precisely what is needed for standardization of the language, since it provides the compiler writer on the one hand with a set of conditions to be satisfied by his implementation (which can be viewed as one of the possible constructive models of the axiom system), whereas on the other hand it leaves him sufficient freedom to adopt the implementation to a particular machine. Several other arguments, e.g., on the advantages of the axiomatic method in the design and comparison of languages, are also given. Although we are not always convinced that these only hold for the axiomatic approach, as opposed to formal methods in general (e.g., in the (constructive) Vienna method there is also a very careful distinction between the rules to be satisfied by each implementation and those which leave open a choice to the implementor), it may well be true that by means of a set of axioms a more concise and elegant definition can be given than by a constructive method. The examples of Hoare have indeed a simple structure; however, he has not yet applied his method to the definition of a complete language. We think that this should be done first before a fair comparison with other systems can be made.

A number of methods for the specification of semantics have resulted from the research on the automatic construction of compilers. There exists a fairly general agreement that these methods, however large their practical use may be, do not provide a solution to the problem of formal definition. First of all, they are of course heavily influenced by present-day compiler techniques, and it is felt that these should be kept apart from the definition of the meaning of a language, since they are in a sense extraneous to it. One should first provide a compiler-independent definition of the semantics of a language, and then one or more compilers can be constructed which satisfy this definition. A second argument is the following: Compiler-oriented specification of semantics is usually directed toward a (possibly somewhat abstracted) real machine; hence one might argue that the definition of a language by means of a compiler is not complete unless some formal description of the machine is given as well. However, the construction of such a description, if feasible at all, is an independent problem, and the definition of the language should not be burdened by it. (It should be remarked that the distinction between the compiler-oriented methods and other formal techniques is not always as clear-cut as suggested here. For instance, in the method of Wirth and Weber (129) a very simple machine is used, the operations of which may well be considered as self-explanatory.)

Because of the arguments mentioned above we feel that a discussion of the compiler-oriented methods is outside the scope of this chapter; they are reviewed in great detail by Feldman and Gries (55).

A particular aspect of some compiler-oriented methods, in which the meaning of a program is specified in parallel with the construction of its parsing tree, is abstracted and generalized by Knuth (74). Consider a language, the syntax of which is defined by means of a context-free grammar. To the elements of the vocabulary (terminals and nonterminals) of the grammar are assigned "attributes," and with each production rule there are associated functions of the attributes. The attributes are of two types, either "synthesized" or "inherited." Consider a derivation tree of a word in the language and an element of the vocabulary labeling a node in the tree. If an attribute of this element is defined—by means of the associated functions—in terms of the attributes of its descendants only, it is called synthesized; if it depends only on the attributes of its ancestors, it is called "inherited." In general, a combination of synthesized and inherited attributes may lead to circular definitions. An algorithm is given yielding a necessary and sufficient condition for detecting circularities. The system is illustrated by means of a simple language describing the operations of Turing machines. (In our opinion, the semantic definition in this example is not quite sufficient: It yields essentially a translation of a Turing machine program into another program resembling the conventional scheme with states and a state transition function. However, the definition does not include a direct formal description of the way in which the machine operates upon the tape.)

Again, a judgment on the merits of Knuth's method must be reserved until it has been applied to a complete programming language.

3.7. Summary

The various language-definition methods of the previous sections may be briefly summarized as follows:

In the Markov-algorithm method programs are interpreted by means of an abstract machine supplied with a list of transformation rules; the machine operates directly upon sequences of symbols.

In McCarthy's method programs are considered as abstract objects to which a meaning is attached by means of state vector functions.

In the Vienna method program texts are translated into abstract objects which are interpreted by an abstract machine characterized by its states and state transition function.

In Landin's method programs are translated *via* an intermediate stage of abstract objects into extended λ-expressions, and these expressions are evaluated by an abstract machine.

In Strachey's method programs are translated into pure λ-expressions.

In the ALGOL 68 method the meaning of a program, i.e., a sequence of symbols produced by the syntax, is defined by the actions which are performed in its elaboration by a hypothetical computer.

In the axiomatic method the meaning of a program is derived from a list of axioms which characterize implicitly the properties of the constructions in the language concerned.

In the compiler-oriented methods programs are translated into assembly language programs for machines which more or less resemble present-day computers.

In Knuth's method meaning is attached to a program by associating semantic functions with the nodes in its derivation tree.

There is apparently a great variety in these systems. In our opinion, it is not yet possible to determine which one of the methods should be preferred, or to predict which method, if any, will prevail in the future. First a set of criteria must be developed for comparing the different systems. However, there has as yet been no systematic research in this area leading to a general framework in which the respective merits of the various methods can be assessed. Therefore we restrict ourselves to a few tentative remarks on a number of points which should be taken into account in the judgement of a formal definition method; no attempt will be made at a systematic evaluation of the methods reviewed by means of these criteria.

1. A first criterion is the scope of the description method. Is it applicable to all programming languages or only to a certain subclass of them? The relevance of this question is limited by the lack of a suitable definition of the notion of programming language; this renders "the class of all programming languages" a rather vague entity. In principle, the answer is easy: As soon as the formalism can describe Turing machines, it has enough descriptive power. In practice, one may have some confidence that a system which has proved capable of defining a language of the size of ALGOL 60, say, will also be applicable to the definition of a reasonably large class of other languages (see, e.g., the definition of a machine tool language by the Markov algorithm method ([29])). However, for the definition of the more esoteric special-purpose languages corresponding special methods may well be preferable.

2. A general criterion of great importance concerns the (inevitably vague) notions of readability, transparency, conciseness, and elegance of the description. One should distinguish here between the method and its applications to specific languages. A very simple method will lead to a very complicated definition. (No one has ever tried to use Turing machines for language definition.) On the other hand, one requires the method to be substantially simpler than the language to be described. Clearly, a compromise must be found between these two extremes. Another way of putting this is as follows: When one uses a metalanguage for the definition of a language, one expects it to be unnecessary to introduce a metametalanguage for the definition of the metalanguage, etc. Concerning these problems, one should compare the remarks of Landin and Böhm on the self-defining capabilities of their systems. The method of de Bakker ([8]), used on the one hand for the definition of ALGOL 60, is also defined by means of an ALGOL 60 program. For a discussion of the apparent circularities in this approach we refer to de Bakker ([8]).

A number of criteria can be derived directly from the various possible uses of a formal definition, as listed in Section 3.1.

3. Some properties of a definition method which are of interest to the compiler writer are: Is it possible to leave the definition of certain parts of the language either completely open, or to give only a partial definition of them, i.e., a definition which states some basic properties of the concept

concerned, but does not give a full specification? Does the definition provide information on the division of the actions to be performed at compile time or at run time? (This is related to the question of where the borderline is drawn between syntax and semantics in the method under discussion.) What are the properties of the method with respect to the problem of establishing whether the constructions used in the implementation satisfy the definition?

4. Questions related to language design are, e.g.: How much insight is gained into the properties of the language concepts by formally describing them? Is it possible to reflect independent concepts by more or less independent parts of the description? Are small changes in language concepts expressible by small changes in the description? Does the formal description help in the clarification of the interactions between the constituent concepts of the language, e.g., with respect to the detection of concepts which are overlapping, incompatible, or which lead to ambiguous constructions?

5. For the people who want to write or understand programs, the main criterion will be the readability, etc., of the definition. To the use of a formal description in the comparison of languages, the criteria mentioned under (4) are applicable. In order to qualify as a language standard, a description must find the right balance between complete and partial definition.

6. Generally applicable criteria concerning the possibility of obtaining proofs about or in a certain description method cannot be given. One might give some preference to methods which rely on functions and functional composition, since these often provide a convenient means for the phrasing of mathematical arguments. Ultimately, however, only on the base of experience can it be decided whether a system is useful for the derivation of proofs.

Programming languages are highly complex structures, and it is only to be expected that this is reflected in their formal definition. However, we feel that there exists a fairly general agreement that the present methods are not yet satisfactory with respect to these criteria (in particular criterion 2) and that future research should continue in trying to improve them.

Conclusions

The theory of semantics of programming languages is only in its initial stage. Before the goals mentioned in Section 1 will be attained much work needs to be done, on deeper investigation of the foundations of programming concepts, on further development of language definition techniques, and on the application of semantics to language design and translation.

We hope that this chapter has given an impression of what has already been achieved in semantic research, and in this way may have contributed a little to its advancement.

REFERENCES

1. G. Aguzzi and R. Pinzani, On a Formalization of a Storage Allocation Mechanism for FORTRAN, Instituto di Elaborazione dell'Informazione, Report II-70, Pisa (June 1968).
2. A. V. Aho and J. D. Ullman, The Theory of Languages, *Math. Systems Theory* **2**, 97–125 (1968).
3. K. Alber and P. Oliva, Translation of PL/I into Abstract Syntax, Technical Report TR 25.086, IBM Laboratory, Vienna (June 1968).
4. K. Alber, P. Oliva, and G. Urschler, Concrete Syntax of PL/I, Technical Report TR 25.084, IBM Laboratory, Vienna (June 1968).
5. C. D. Allen, D. Beech, J. E. Nicholls, and R. Rowe, An Abstract Interpreter of PL/I, Tech. Note 3004, IBM Hursley Laboratories (November 1966).
6. G. Asser, Funktionen-Algorithmen und Graphschemata, *Z. Math. Logik Grundl. Math.* **7**, 20–27 (1961).
7. G. Asser and V. Vucković, Funktionen-Algorithmen, *Z. Math. Logik Grundl. Math.* **7** 1–8 (1961).
8. J. W. de Bakker, "Formal Definition of Programming Languages," Mathematical Center Tracts, Vol. 16, Mathematisch Centrum, Amsterdam (1967).
9. J. W. de Bakker, On Convex Sublattices of Distributive Lattices, Report ZW 1967-003, Mathematisch Centrum, Amsterdam (May 1967).
10. J. W. de Bakker, Axiomatics of Simple Assignment Statements, Report MR 94, Mathematisch Centrum, Amsterdam (June 1968).
11. J. W. de Bakker, Some Remarks on McCarthy's Theory of Computation, in preparation.
12. D. W. Barron, J. N. Buxton, D. F. Harthley, E. Nixon, and C. Strachey, The Main Features of CPL, *Comp. J.* **6**, 134–143 (1963).
13. S. K. Basu, On Computation in Programming Languages, *ICC Bulletin* **6**, 1–26 (1966).
14. C. Berge, "The Theory of Graphs and Its Applications," John Wiley and Co., New York (1962).
15. D. G. Bobrow (ed.), "Symbol Manipulation Languages and Techniques," Proc. IFIP Working Conference 1966, North-Holland Publishing Co., Amsterdam (1968).
16. C. Böhm, CUCH As a Formal and Description Language, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 179–197, North-Holland Publishing Co., Amsterdam (1966).
17. C. Böhm and W. Gross, Introduction to the CUCH, *in* "Automata Theory" (E. R. Caianiello, ed.), pp. 35–65, Academic Press, New York and London (1966).
18. C. Böhm and G. Jacopini, Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, *Comm. Assoc. Computing Machinery* **9**, 366–372 (1966).
19. W. H. Burge, The Evaluation, Classification, and Interpretation of Expressions, *in* "Proceedings of the Association for Computing Machinery 19th National Conference," pp. A1.4.1–A1.4.22, Assoc. Computing Machinery, New York (1964).

20. R. M. Burstall, Some Aspects of CPL Semantics, Experimental Programming Reports No. 3, University of Edinburgh (April 1965).
21. R. M. Burstall, Semantics of Assignment, *in* "Machine Intelligence" (E. Dale and D. Michie, eds.), Vol. 2, pp. 3–20, Oliver and Boyd, Edinburgh (1967).
22. J. N. Buxton, J. C. Gray, and D. Park, CPL Elementary Programming Manual, Edition II, Technical Report, Cambridge (1966).
23. A. Caracciolo di Forino, *N*-ary Selection Functions and Formal Selective Systems, *Calcolo* **1**, 49–81 (1964).
24. A. Caracciolo di Forino, Linguaggi Programmativi Speciali, *Calcolo* **2**, Supplement No. 2 (1965).
25. A. Caracciolo di Forino, *M*-Valued Logics and *m*-ary Selection Functions, *in* "Automata Theory" (E. R. Caianiello, ed.), pp. 107–114, Academic Press, New York and London (1966).
26. A. Caracciolo di Forino, Generalized Markov Algorithms and Automata, *in* "Automata Theory" (E. R. Caianiello, ed.), pp. 115–130, Academic Press, New York and London (1966).
27. A. Caracciolo di Forino, String Processing Languages and Generalized Markov Algorithms, *in* "Symbol Manipulation Languages and Techniques," Proc. IFIP Working Conference 1966 (D. G. Bobrow, ed.), pp. 191–206, North-Holland Publishing Co., Amsterdam (1968).
28. A. Caracciolo di Forino, A Comparison between Generalized Markov Algorithms and de Bakker Algorithms, unpublished report.
29. A. Caracciolo di Forino and A. Camera, On a Formal Definition of Direct Machine Tool Languages, Instituto di Elaborazione dell'Informazione, Interim Scientific Report No. 1, Pisa (July 1968).
30. A. Caracciolo di Forino and L. Carlucci, On an Algorithmic Interpretation of the Formal Definition of PL/I, unpublished report.
31. A. Caracciolo di Forino and G. Leoni, On the Formal Description of ALGOL 60 Semantics by Means of a Generalized Markov Algorithm, unpublished report.
32. A. Caracciolo di Forino and R. Rebaudo, On a Formal Definition of SIMULA by Means of an Extended Markov Algorithm, unpublished report.
33. A. Caracciolo di Forino, L. Spanedda, and N. Wolkenstein, PANON-1B: A Programming Language for Symbol Manipulation, *Calcolo* **3**, 245–255 (1966).
34. A. Caracciolo di Forino and N. Wolkenstein, On a Class of Programming Languages for Symbol Manipulation Based on Extended Markov Algorithms, Centro Studi Calcolatrici Elettroniche, Report No. 21, Pisa (1963).
35. C. Christensen, Examples of Symbol Manipulation in the AMBIT Programming Language, *in* "Proceedings of the Association for Computing Machinery 20th National Conference," pp. 247–261, Assoc. Computing Machinery, New York (1965).
36. A. Church, "The Calculi of Lambda-Conversion," Annals of Math. Studies, No. 6, Princeton Univ. Press, Princeton, New Jersey (1951).
37. K. Cohen and J. H. Wegstein, AXLE, an Axiomatic Language for String Transformation, *Comm. Assoc. Computing Machinery* **8**, 657–661 (1965).
38. D. C. Cooper, On the Equivalence of Certain Computations, *Comp. J.* **9**, pp. 45–52 (1966).
39. D. C. Cooper, Mathematical Proofs about Computer Programs, *in* "Machine Intelligence" (N. L. Collins and D. Michie, eds.), Vol. 1, pp. 17–28, Oliver and Boyd, Edinburgh (1966).

40. D. C. Cooper, Some Transformations and Standard Forms of Graphs, with Applications to Computer Programs, *in* "Machine Intelligence" (E. Dale and D. Michie, eds.), Vol. 2, pp. 21–32, Oliver and Boyd, Edinburgh (1967).
41. D. C. Cooper, Böhm and Jacopini's Reduction of Flow Charts (A Letter to the Editor), *Comm. Assoc. Computing Machinery* **10**, 463 (1967).
42. H. B. Curry and R. Feys, "Combinatory Logic," North-Holland Publishing Co., Amsterdam (1958).
43. M. Davis, "Computability and Unsolvability," McGraw-Hill Book Co., New York (1958).
44. R. M. Dicker, A Set of Independent Postulates for Boolean Algebra, *Proc. London Math. Soc.* **3** (3), 20–30 (1963).
45. J. J. Donovan and H. F. Ledgard, Canonic Systems and Their Applications to Programming Languages, Mem. Mac-M-347, Project MAC, MIT, Cambridge, Mass. (April 1967).
46. J. Earley, Generating a Recognizer for a BNF Grammar, Carnegie Institute of Technology (June 1965).
47. S. Eilenberg and C. C. Elgot, Iteration and Recursion, IBM Research Report RC 2148 (July 1968).
48. C. C. Elgot, Machine Species and Their Computation Languages, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 160–179, North-Holland, Publishing Co., Amsterdam (1966).
49. C. C. Elgot and A. Robinson, Random Access, Stored Program Machines, an Approach to Programming Languages, *J. Assoc. Computing Machinery* **11**, 365–399 (1964).
50. E. Engeler, Algorithmic Properties of Structures, *Math. Systems Theory* **1**, 183–195 (1967).
51. A. P. Ershov, Operator Algorithms I, *in* "Problems of Cybernetics," Vol. 3, pp. 697–763, Pergamon Press, New York (1962).
52. A. P. Ershov, On Yanov's Operator Schemes, *Problemy Kibernetiki* **20**, Nauka, Moscow (1967).
53. A. P. Ershov and A. A. Lyapunov, On the Formalization of the Notion of Program, *Kibernetika* (*Kiev*) **5** (10), 40–57 (1967).
54. A. Evans, Syntax Analysis by a Production Language, PhD Thesis, Carnegie Institute of Technology (1965).
55. J. Feldman and D. Gries, Translator Writing Systems, *Comm. Assoc. Computing Machinery* **11**, 77–113 (1968).
56. E. M. Fels, Kaluzhnin Graphs and Yanov Writs, *in* "Logik und Logikkalkül" (M. Käsbauer and F. von Kutschera, eds.), pp. 159–178, Verlag Karl Alber, Freiburg/München (1962).
57. M. Fleck and E. Neuhold, Formal Definition of the PL/I Compile Time Facilities, Technical Report TR25.080, IBM Laboratory, Vienna (June 1968).
58. R. W. Floyd, Assigning Meanings to Programs, *in* "Mathematical Aspects of Computer Science," Proc. of Symposia in Applied Mathematics, Vol. 19 (J. T. Schwartz, ed.), pp. 19–32, American Mathematical Society, Providence, Rhode Island (1967).
59. S. Ginsburg, "The Mathematical Theory of Context Free Languages," McGraw-Hill Book Co., New York (1966).

60. H. H. Goldstine and J. von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," Institute for Advanced Study, Princeton, New Jersey (1947).
61. S. Gorn, Mechanical Pragmatics: a Time Motion Study of a Miniature Mechanical Linguistic System, *Comm. Assoc. Computing Machinery* **5**, 576–589 (1962).
62. G. Hain and K. Hain, Automatic Flow Chart Design, *in* "Proceedings of the Association for Computing Machinery 20th National Conference," pp. 513–523, Ass. Computing Machinery, New York (1965).
63. C. A. R. Hoare, The Axiomatic Method, The National Computing Centre, Manchester, England (1968).
64. S. Igarashi, On the Logical Schemes of Algorithms, *Information Processing in Japan* **3**, 12–18 (1963).
65. S. Igarashi, A Formalization of the Description of Languages and the Related Problems in a Gentzen-type Formal System, Research Notes of the Research Association of Applied Geometry, Third Series, No. 80 (1964).
66. S. Igarashi, An Axiomatic Approach to the Equivalence Problems of Algorithms with Applications, PhD Thesis, University of Tokyo (1964); reprinted in *Report of the Computer Center University of Tokyo* **1**, 1–101 (1968).
67. S. Igarashi, On the Equivalence of Programs Represented by ALGOL-Like Statements, *Report of the Computer Center University of Tokyo* **1**, 103–118 (1968).
68. R. Itturiaga, Contributions to Mechanical Mathematics, PhD Thesis, Carnegie-Mellon University (May 1967).
69. L. A. Kaluzhnin, Algorithmization of Mathematical Problems, *in* "Problems of Cybernetics," Vol. 2, pp. 371–391, Pergamon Press, New York (1961).
70. D. M. Kaplan, Correctness of a Compiler for ALGOL-Like Programs, Artificial Intelligence Memo No. 48, Stanford University (July 1967).
71. D. M. Kaplan, Some Completeness Results in the Mathematical Theory of Computation, *J. Assoc. Computing Machinery* **15**, 124–134 (1968).
72. R. M. Karp, A Note on the Application of Graph Theory to Digital Computer Programming, *Information and Control* **3**, 179–189 (1960).
73. D. E. Knuth, Algorithm and Program, Information and Data (A Letter to the Editor), *Comm. Assoc. Computing Machinery* **9**, 654 (1966).
74. D. E. Knuth, Semantics of Context Free Languages, *Math. Systems Theory* **2**, 127–145 (1968).
75. L. Krider, A Flow Analysis Algorithm, *J. Assoc. Computing Machinery* **11**, 429–436 (1964).
76. J. Kunze, Selektive Graphschemata, *Z. Math. Logik Grundl. Math.* **13**, 101–122 (1967).
77. P. J. Landin, The Mechanical Evaluation of Expressions, *Comp. J.* **6**, 308–320 (1964).
78. P. J. Landin, A Correspondence between ALGOL 60 and Church's Lambda Notation, *Comm. Assoc. Computing Machinery* **8**, 89–101, 158–165 (1965).
79. P. J. Landin, A λ-Calculus Approach, *in* "Advances in Programming and Non-Numerical Computation" (L. Fox, ed.), pp. 97–141, Pergamon Press, New York (1966).
80. P. J. Landin, A Formal Description of ALGOL 60, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 266–294, North-Holland Publishing Co., Amsterdam (1966).

81. P. J. Landin, The Next 700 Programming Languages, *Comm. Assoc. Computing Machinery* **9**, 157–166 (1966).
82. P. Lauer, Abstract Syntax and Interpretation of ALGOL 60 Programs, Laboratory Report LR 25.6.001, IBM Laboratory, Vienna (April 1968).
83. P. Lauer, Concrete Representation of Abstract ALGOL 60 Programs, Laboratory Report LR 25.6.002, IBM Laboratory, Vienna (May 1968).
84. G. Leoni, On Formal Definition of COBOL Semantics, unpublished report.
85. R. L. London, A Computer Program for Discovering and Proving Sequential Recognition Rules for Well-Formed Formulas Defined by a Backus Normal Form Grammar, PhD Thesis, Carnegie Institute of Technology (May 1964).
86. R. L. London, A Computer Program for Discovering and Proving Recognition Rules for Backus Normal Form Grammars, *in* "Proceedings of the Association for Computing Machinery 19th National Conference," pp. A1.3.1–A1.3.7, Assoc. Computing Machinery, New York (1964).
87. P. Lucas, Two Constructive Realizations of the Block Concept and Their Equivalence, Technical Report 25.085, IBM Laboratory, Vienna (June 1968).
88. P. Lucas, to appear.
89. P. Lucas, K. Alber, K. Bandat, H. Bekić, P. Oliva, K. Walk, and G. Zeisel, Informal Introduction to the Abstract Syntax and Interpretation of PL/I, Technical Report TR 25.083, IBM Laboratory, Vienna (June 1968).
90. P. Lucas, P. Lauer, and H. Stigleitner, Method and Notation for the Formal Definition of Programming Languages, Technical Report TR 25.087, IBM Laboratory, Vienna (June 1968).
91. D. C. Luckham, D. M. R. Park, and M. S. Paterson, On Formalized Computer Programs, to appear.
92. A. A. Markov, "The Theory of Algorithms," Office of Technical Services, US Dept. of Commerce, Washington, D.C. (1962).
93. D. Martin and G. Estrin, Models of Computations and Systems—Evaluation of Vertex Probabilities in Graph Models of Computations, *J. Assoc. Computing Machinery* **14**, 281–299 (1967).
94. D. Martin and G. Estrin, Experiments on Models of Computations and Systems, *IEEE Trans. Electronic Computers* **EC-16**, 59–69 (1967).
95. D. Martin and G. Estrin, Models of Computational Systems—Cyclic to Acyclic Graph Transformations, *IEEE Trans. Electronic Computers* **EC-16**, 70–79 (1967).
96. J. McCarthy, LISP 1.5 Programmer's Manual, Computation Center and Research Laboratory of Electronics, MIT, Cambridge, Mass. (August 1962).
97. J. McCarthy, Computer Programs for Checking Mathematical Proofs, *in* "Recursive Function Theory," Proc. of Symposia in Pure Mathematics, Vol. 5, pp. 219–227, American Mathematical Society, Providence, Rhode Island (1962).
98. J. McCarthy, A Basis for a Mathematical Theory of Computation, *in* "Computer Programming and Formal Systems" (P. Braffort and D. Hirschberg, eds.), pp. 33–69, North-Holland Publishing Co., Amsterdam (1963).
99. J. McCarthy, Towards a Mathematical Science of Computation, *in* "Information Processing 1962," Proc. IFIP Congress 1962 (C. M. Popplewell, ed.), pp. 21–28, North-Holland Publishing Co., Amsterdam (1963).
100. J. McCarthy, Problems in the Theory of Computation, *in* "Information Processing 1965," Proc. IFIP Congress 1965 (W. A. Kalenich, ed.), Vol. 1, pp. 219–222, Spartan Books, Washington, D.C. (1965).

101. J. McCarthy, A Formal Description of a Subset of ALGOL, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 1–12, North-Holland Publishing Co., Amsterdam (1966).
102. J. McCarthy and J. Painter, Correctness of a Compiler for Arithmetic Expressions, *in* "Mathematical Aspects of Computer Science," Proc. of Symposia in Applied Mathematics, Vol. 19 (J. T. Schwartz, ed.), pp. 33–41, American Mathematical Society, Providence, Rhode Island (1967).
103. E. Mendelson, "Introduction to Mathematical Logic," D. van Nostrand Co., Princeton, New Jersey (1964).
104. A. R. Meyer and D. M. Ritchie, Computational Complexity and Program Structure, IBM Research Report, RC-1817.
105. A. R. Meyer and D. M. Ritchie, The Complexity of Loop Programs, *in* "Proc. ACM 22nd National Conference," pp. 465–469 (1967).
106. E. Munteanu, Analyse Logique des Algorithmes, I, *Mathematica* **9**, 111–128 (1967).
107. R. Narasimhan, Programming Languages and Computers: A Unified Metatheory, *in* "Advances in Computers" (F. L. Alt and M. Rubinoff, eds.), Vol. 8, pp. 189–244, Academic Press, New York and London (1967).
108. P. Naur, Proof of Algorithms by General Snapshots, *BIT* **6**, 310–317 (1966).
109. J. Painter, Semantic Correctness of a Compiler for an ALGOL-Like Language, Artificial Intelligence Memo No. 44, Stanford University (March 1967).
110. M. S. Paterson, Equivalence Problems in a Model of Computation, Doctoral Dissertation, Cambridge University (1967).
111. R. Péter, Graphschemata und Rekursive Funktionen, *Dialectica* **12**, 373–393 (1958).
112. R. Péter, Über die Partiell-rekursivität der durch Graphschemata definierten zahlentheoretischen Funktionen, *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös nominatae Sectio mathematica* **2**, 41–48 (1959).
113. C. V. Ramamoorthy, Discrete Markov Analysis of Computer Programs, *in* "Proceedings of the Association for Computing Machinery 20th National Conference," pp. 386–391, Assoc. Computing Machinery, New York, (1965).
114. M. V. Rennie, Theory of Procedures, I, Simple Conditionals (Abstract), *J. Symbolic Logic* **32**, 577 (1967).
115. P. C. Rosenbloom, "The Elements of Mathematical Logic," Dover Publications, New York (1950).
116. J. D. Rutledge, On Ianov's Program Schemata, *J. Assoc. Computing Machinery* **11**, 1–9 (1964).
117. A. Schurmann, The Application of Graphs to the Analysis of Distribution of Loops in a Program, *Information and Control* **7**, 275–282 (1964).
118. A. Schurmann, On the Application of Graph Theory to Determine the Number of Multisection Loops in a Program, *Algorythmy* **II** (3), 73–81 (1964).
119. A. Schurmann, The Distribution of Cycles in a Finite Graph and the Application of Graphs to Computer Programming, *Algorythmy* **II** (4), 85–100 (1965); (in Polish).
120. M. Sintzoff, Existence of a van Wijngaarden Syntax for Every Recursively Enumerable Set, *Annales de la Société Scientifique de Bruxelles*, **81** (II), 115–118 (1967).
121. T. B. Steel, Jr. (ed.), "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964, North-Holland Publishing Co., Amsterdam (1966).
122. T. B. Steel, Jr., Standards for Computers and Information Processing, *in* "Advances

in Computers" (F. L. Alt and M. Rubinoff, eds.), Vol. 8, pp. 103–152, Academic Press, New York and London (1967).

123. C. Strachey, Towards a Formal Semantics, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 198–220, North-Holland Publishing Co., Amsterdam (1966).
124. C. Strachey (ed.), CPL Working Papers, University of London Institute of Computer Science (1966).
125. C. Strachey, Fundamental Concepts in Programming Languages, to appear in the Proceedings of the 1967 NATO Summer School, Copenhagen.
126. H. Thiele, "Wissenschaftstheoretische Untersuchungen in Algorithmische Sprachen I," VEB Deutscher Verlag der Wissenschaften, Berlin (1966).
127. K. Walk, K. Alber, K. Bandat, H. Bekić, G. Chroust, V. Kudielka, P. Oliva, and G. Zeisel, Abstract Syntax and Interpretation of PL/I, Technical Report TR 25.082, IBM Laboratory, Vienna (June 1968).
128. N. Wirth, A Generalization of ALGOL, *Comm. ACM* **6**, 547–554 (1963).
129. N. Wirth and H. Weber, EULER, A Generalization of ALGOL, and Its Formal Definition, *Comm. Assoc. Computing Machinery* **9**, pp. 13–23, 89–99 (1966).
130. B. A. Wittman and P. Z. Ingerman, A Threshold Selection Language, *in* "Proceedings of the Association for Computing Machinery 22nd National Conference," pp. 311–316, Assoc. Computing Machinery, New York (1967).
131. A. van Wijngaarden, Generalized ALGOL, *in* "Symbolic Languages in Data Processing," Proc. ICC Symp. Rome 1962, pp. 409–419, Gordon and Breach, New York (1962); also *in* "Annual Review in Automatic Programming" (R. Goodman, ed.), Vol. 3, pp. 17–26, Pergamon Press, New York (1963).
132. A. van Wijngaarden, Recursive Definition of Syntax and Semantics, *in* "Formal Language Description Languages for Computer Programming," Proc. IFIP Working Conference 1964 (T. B. Steel, Jr., ed.), pp. 13–24, North-Holland Publishing Co., Amsterdam (1966).
133. A. van Wijngaarden, Numerical Analysis as an Independent Science, *BIT* **6**, 66–81 (1966).
134. A. van Wijngaarden (ed.), B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, Report on the Algorithmic Language ALGOL 68, Report MR 101, Mathematisch Centrum, Amsterdam (1968).
135. Y. I. Yanov, On the Equivalence and Transformation of Program Schemes, *Comm. Assoc. Computing Machinery* **1** (10), 8–12 (1958).
136. Y. I. Yanov, On Matrix Program Schemes, *Comm. Assoc. Computing Machinery* **1** (12), 3–6 (1958).
137. Y. I. Yanov, The Logical Schemes of Algorithms, *in* "Problems of Cybernetics," Vol. 1, pp. 82–140, Pergamon Press, New York (1960).
138. Y. I. Yanov, On Local Transformation of Algorithm Schemes, *Problemy Kibernetiki* **20**, Nauka, Moscow (1967); (in Russian).
139. V. H. Yngve, An Introduction to COMIT Programming, The Research Laboratory of Electronics and the Computation Center, MIT, Cambridge, Mass. (November 1961).
140. H. Zemanek, Semiotics and Programming Languages, *Comm. Assoc. Computing Machinery* **9**, 139–143 (1966).

9169