CHAPTER 1

MAIN APPROACHES TO FORMAL SPECIFICATIONS

The work on the formal definition of programming languages spans some twenty years. This chapter sets the historical context of the VDM work. It does not, however, purport to be a complete history of the subject (more detail of the Vienna work in particular can be found in [Lucas 81a]). The three main approaches to the definition of programming languages are described. The so-called "Vienna Definition Language" (VDL) is probably the first to be used in the definition of large languages: it is based on the "operational approach". Although coming later in time, "mathematical" or "denotational semantics" has become the most widely accepted approach to formal definition. Because of its position as a reference point mathematical semantics is here described before the other two approaches. In order to prove that programs in a language satisfy some specification, it is normal to use rules of deduction about the language. It is possible to regard such rules as axioms and to treat them as a definition of the language: such "axiomatic semantics" is also described. In addition the concept of, and the reasons for the use of, "abstract syntax" are explained. A final section considers some of the open research problems relating to formal specifications.

(This chapter is a revised version of [Lucas 78a])

CONTENTS

1.1 THE NEED FOR FORMAL SPECIFICATIONS

Computer systems can be viewed as machines capable of interpreting languages; they accept and understand declarative sentences, obey imperative sentences and answer questions, all within the framework of those languages for which the systems were built. A computer system accomplishes its tasks on the basis of a prescription of these tasks, that is on the basis of a program expressed in some programming language.

There is no inherent disparity between human languages (including natural language and the artificial languages of science) and languages used to talk to computers. Thus there is no need to apologize for "anthropomorphisms" in the above point of view; in fact, our only way to talk scientifically about the relation of humans to their natural languages is in terms of computer notions (or so it seems to me).

By viewing computers as language interpreting machines it becomes quite apparent that the analysis of programming (and human) languages is bound to be a central theme of computer science.

Part of the fascination of the subject is of course related to its intimate connection to human language, that is the mechanisms we study mirror in some way at least part of our own internal mechanisms.

Although there is no inherent disparity between human language and computer language, there is at present a huge gap between what we can achieve by human conversation and our communication with machines. A little further analysis will indicate the nature of the gap.

First we consider the structural aspect of language, that is how phrases are composed of words and sentences are built from phrases, commonly called "syntax". There are efficient and precise methods to define the syntax of a language and algorithms to compose and decompose sentences according to such definitions. The problem is more or less solved. Yet, computer languages usually have a simpler and more regular syntax than natural languages (and even some scientific notations) and there are technical problems yet to be solved. Moreover, it seems to me, there is not much of a gap.

Second, there is the aspect of meaning, or "semantics" as it is usually called. Now we get into more subtle problems. Let me restrict the dis-

cussion, for the time being, to the objects we can talk about in the various languages (rather than considering what we can say about them). Programming languages in the strict sense talk invariably about rather abstract objects such as numbers, truth-values, character strings and the like. Certainly, the major programming languages in use do not let us talk about tables, chairs or people, nor even about physical dimensions of numbers such as: hours, pounds, or feet. The commercial languages do not know about the distinction of dollars and francs, and scientific languages do not know about time and space. There have been some attempts to include those notions or a device that makes it possible to define these notions within a language (e.g. the class concept in SIMULA and PASCAL and the investigations around abstract data types). If we extend the notion of programming language to include query languages and database languages, we may observe a tendency in the indicated direction. Yet, there is a gap. Artificial Intelligence has experimented for some time with languages that can be used to talk about objects other than numbers and we should probably try to learn from these experiments.

Definition methods concerning semantic, and even more so, mechanical ways to use semantic definitions are much less understood than in the case of the syntactic aspect.

Thirdly, there is the aspect of language understanding; I hesitate to call this "pragmatics" since the latter term has been used for too many things.

Suppose I ride on a train with a friend. The friend observes: "The windows are wet" (*). The statement is presumably structured according to the English grammar and has a certain meaning. However, I would probably not just analyze the sentence and determine its meaning. Most likely I would react by looking at a window, observe that there are drops, conclude that it is raining, prepare my umbrella so that I don't get wet when I get off the train.

(*) It would not make any difference to the following argument if my friend had used *META-IV* and passed a note saying: *"wet(windows)"*. That is to say, I do not discuss the distinction between natural language and standard (formal) notation, but the distinction of the human and computer use of the statement irrespective of the form.

To draw all these conclusions and act accordingly I need to use a lot of knowledge about the physical world in general and about my specific environment. It is in this area of language understanding, where I see the bigger gap between our interaction with the computer as opposed to humans. What is lacking in the machine are models of the external world and general mechanisms to draw conclusions and trigger actions. Again artificial intelligence and natural languages research have been concerned with the problem. But, this has not as yet had any practical influence on for example commercial applications. With the increase in computer power it might very well be worth seeking such influence.

With the preceding paragraphs I wanted to put the present subject into a much larger context than is usual. Thank God, there is more to programming languages than procedures, assignment and goto's (or no goto's). The rest of this chapter is a lot less ambitious and remains more or less within the traditional concepts of programming languages. It presents my subjective perceptions of the various origins of the methods of semantic definitions.

1.2 HISTORICAL BACKGROUND

The theory of programming languages and the related formal definition techniques, have roots in, and are related to, several other disciplines such as linguistics, formal logic, and certain mathematical disciplines. In fact, the terms "syntax" and "semantics" and the distinction between the respective aspects of language, have been introduced by the American philosopher Charles Morris [Morris 38a, Zemanek 66a]. He developed a science of signs which he called semiotics. Semiotics, according to Morris, is subdivided into three distinct fields: syntax, semantics, and pragmatics. In his book [Morris 55a] Morris defines:

Pragmatics deals with the origin, uses and effects of signs within the behavior in which they occur;

Semantics deals with the signification of signs in all modes of signifying;

Syntax deals with the combination of signs without regard for their specific significations or their relation to the behavior in which they occur.

The clear distinction between syntax and semantics was first applied to a programming language in the ALGOL 60 report [Naur 63a]. The resulting insight has turned out to be tremendously useful. There have been several not so successful attempts to carry the notion of pragmatics into the theory of programming languages (see e.g. San Dimas Conference [ACM 66a]). We may start the history of formal definition methods for programming languages with the year 1959 when J. Backus proposed a scheme for the syntactic definition of ALGOL 60 [Backus 60a]. This scheme (a generative grammar) was then actually used in the ALGOL 60 report; the related notation is known as BNF (for Backus Normal Form or Backus Naur Form). BNF, or variations thereof, have been used in many instances; it has stimulated theoretical research as well as practical schemes for compiler production (both automatic and non-automatic). Roughly speaking, BNF grammars coincide with the class of context free grammars in [Chomsky 59a]; it is worth mentioning that Chomsky defined his grammatical formalisms in an attempt to obtain a basis for the syntax of the English language. Much research has been devoted to the study of subtypes and extended types of BNF grammars. The latter in support of the desire to capture more syntactic properties of the language to be defined; the former, that is the study of subtypes is usually motivated by the wish to find properties which permit fast syntax recognition and analysis algorithms. The subject of formal syntax definition, and the related computational problems and methods, have found their way into textbooks and computer science curricula; in fact, the larger part of compiler writing courses is usually spent on syntax problems.

After the ALGOL 60 report, the lack of rigorous definition methods for the semantics of programming languages has become widely recognized. Furthermore, the success of formal syntax definitions invited similar attempts for the semantic aspects of programming languages. The problem turned out to be of an obstinate nature. To date, there is no solution that enjoys the consensus of the whole computing community.

The instructions of machine languages are defined by the behaviour of the respective machine upon execution of these instructions. The associated manuals usually describe first what constitutes the state of the specific machine (e.g. content of main storage, content of registers, etc.) and then for each instruction and any given state the successor state after execution of the instruction to be defined. Hence, for a programmer, the most direct way to define a programming language is in terms of an interpreting machine; however, for higher level languages,

we must abstract from particularities of hardware machines and implementation details and use a suitable hypothetical machine instead. E.W.Dijkstra formulated the situation in 1962 [Dijkstra 62b] as follows: "A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it" (*).

The classic paper that has led to much research is by McCarthy [McCarthy 62a]. The paper outlines a basis for a theory of computation; more important for our subject, it establishes the main goals and motivation: methods to achieve correctness of programs in general and of compilers in particular; rigorous language definitions constitute an intermediate problem. The schema for language definitions proposed by McCarthy contains a number of novel subjects. Firstly, a complete separation of notational issues, that is the representation of phrases by linear character strings, from the definition of the essential syntactic structure of a language. The latter definition is called "Abstract Syntax". It is, at least for complicated languages, much more concise than the concrete syntax. Thus, a semantic definition on the basis of an abstract syntax becomes independent of notational details and is also more concise. Secondly, state vectors are introduced as the basis of the semantic definitions proper, that is the meaning of an instruction or statement is defined as a state transition. The paper shows in principle the task of proving compilers correct. The basic scheme of language definitions has been elaborated in many instances during the past decade, e.g. by the earlier work of the Vienna Laboratory on PL/I [Lucas 69a] and the ECMA-ANSI standard [ANSI 76a].

Another successful direction of research was initiated by P. Landin [Landin 64a, 65a], using the lambda-calculus [Church 41a] as the fundamental basis. He revealed that certain concepts of ALGOL 60 (and similar languages) can be viewed as syntactic variations (syntactic "sugar") of the lambda-calculus. The inherently imperative concepts, assignment and transfer of control, were captured by introducing new primitives into the lambda-calculus; the extended base is defined by the so-called

---

(*) It would be unfair to include this quotation and not say that E.W. Dijkstra would probably no longer defend this position, and rather tend to be a proponent of the direction described under "Axiomatic Approach" in this chapter.

SECD machine, a hypothetical machine whose state consists of four components: Storage, Environment, Control and Dump. The machine state has more structure than the state vectors of McCarthy, because the machine had to reflect more complicated concepts (blocks, local names) than McCarthy's original simple example was intended to.

In 1964 C.Strachey [Strachey 66a] argued that, with the introduction of a few basic concepts, it was possible to describe even the imperative parts of a programming language in terms of the lambda-calculus. C. Strachey initiated a development that led to an explication of programming languages known as "mathematical" or "denotational semantics". The fundamental mathematical basis for this development was contributed by D.Scott in 1970 [Scott 70a]. The joint paper, by D.Scott & C.Strachey [Scott 71a] offers a description method and its application to essential language concepts based upon the indicated research.

Research on axiom systems and proof theory suitable as a base for correctness proofs of programs was initiated by R.Floyd [Floyd 67a], with a simple flow-diagram language. C.A.R.Hoare [Hoare 69a,71a], extended and refined the results to apply to constructs of higher level languages. Less formalized, but similar thoughts were expressed in [Naur 66b]. The area has been the most actively pursued, including experiments in automatic program verification.

There are several pioneering research efforts, which do not so evidently fall into the categories introduced above. Among the very early results published on semantics is A. van Wijngaarden's *"Generalized ALGOL"* [van Wijngaarden 62]. J. de Bakker [de Bakker 69a] discovered that the schema proposed by A.van Wijngaarden can be viewed as a generalized Markov Algorithm. A.Carraciolo [Forino 66a] also used Markov Algorithms as the starting point for the formalization of programming language semantics.

For anyone familiar with syntax directed compilers it is tempting to apply similar ideas to the definition of semantics. A definition method on this basis is due to D. Knuth [Knuth 68a]. In some way or another, a formal definition of the semantics of a language invariably specifies a relation between any phrase of the language and some mathematical object called the denotation of the phrase. D. Knuth provides a convenient schema that permits the specification of functions over the phrases of a language (assuming that the phrase structure of the language is given by

a production system). Most research so far has been devoted to the definition and analysis of existing languages (or concepts found in existing languages). Yet, formal semantics could be a most valuable intellectual tool for the design of novel programming concepts (or new programming language constructs). There are rare instances of such applications of formal semantics ~~are found in~~ [Dijkstra 74a, Dijkstra 75a, Dennis 75a, Henderson 75a]). (e.g.

1.3 BASIC METHODOLOGICAL APPROACHES

1.3.1 Abstract Syntax

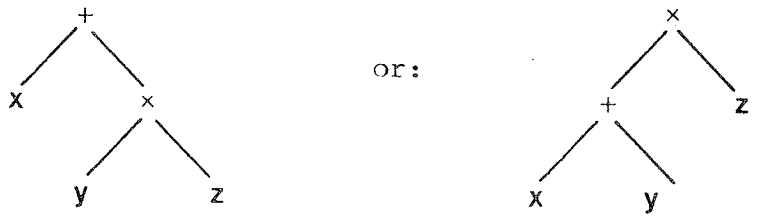
The notion of abstract syntax is of considerable value for practical definitions of notationally complex languages. There exist several methodological variations, which all achieve the same objective: to abstract from semantically irrelevant notational details and reduce the syntax to define the essence of the linguistic forms only.

For illustration consider the following examples. Let v be the category of variables and e be the category of expressions. Several notational variants are in use to denote assignment statements e.g.:

$$v = e \quad \text{or} \quad v := e \quad \text{or} \quad e \rightarrow v$$

The semantically essential structure common to these notations is that there is a syntactic category called assignment statement, and that an assignment statement has two components, a variable and an expression.

An abstract syntax may define an expression to be either an elementary expression (variable, constant, etc.) or a binary operation consisting of an operator, a first operand, and a second operand (the definition of expressions may have several other alternatives); operands are also expressions. As a concrete syntax, meant to define character strings, such a definition would be hopelessly insufficient and ambiguous, for example we would not know whether to parse $x+y\times z$ into:



Thus the concrete syntax has to introduce punctuation marks such as parentheses, and, in the example of expressions, precedence rules of operators to avoid ambiguities. However, the definition of expressions given above is perfectly usable as an abstract syntax definition. It can be regarded as a definition of parsing trees, hence the ambiguity problem is completely avoided. Thus there are advantages gained even in the case where only one language is considered: representational details are suppressed and each phrase is given a kind of normal form.

For practical cases, such as *PL/I*, the number of rules necessary to define an abstract syntax is much smaller than for the corresponding concrete syntax; hence we have obtained a more concise basis for the semantic definition. The price we pay is an additional part for the formalization of a language, which establishes the relation between the concrete and the abstract syntax.

1.3.2 Mathematical Semantics

The semantics of a given language is formalized by associating a suitable mathematical object (set, function, etc.) with each phrase of the language; the phrase is said to denote the associated object; the object is called the denotation of the phrase. Furthermore, to gain a "referentially transparent" view of the language to be defined, denotations of composite phrases are defined solely in terms of the denotations of subphrases. The major problem in establishing the mathematical semantics for a given language is to find suitable mathematical objects, that can serve as the denotations. We will write *M[p]* for the denotation of a phrase *p* (*). To indicate the various phrases to be discussed, we will use an ALGOL-like notation, for example *M[x:=x-1]* is the denotation of the assignment statement *x:=x-1*. Further elaboration of the subject considers a series of programming language concepts in increasing order of complexity. Take first a simple language with a fixed set of variables (*id*), expressions (*e*) without side effects, assignments *(id:=e)* and compound statements *(s1;s2)*. If we were to construct a definitional interpreter, we would certainly introduce a state vector (à la McCarthy).

(*) For small languages it is possible to use concrete syntax and basic function notation. In order to be clear, definitions of larger languages must use abstract syntax and combinators.

Although we do not wish to specify particular ways to compute the effect of executing programs and their parts, we still have to characterize the overall effect of this execution. Therefore we introduce state vectors σ, which are (usually) partial functions from variable names *ID* into the set of values, *VAL*, that is:

$$\sigma: ID \rightharpoonup VAL.$$

Let Σ be the set of all possible states. The kinds of denotations that occur in the example language can now be chosen as follows:

$$M[e] : \Sigma \rightharpoonup VAL$$
$$M[st]: \Sigma \rightharpoonup \Sigma$$

that is the denotations of expressions are functions from states into values and the denotations of statements are functions from states to states.

Assuming that $M[e]$ has been defined elsewhere, the definition of assignment and compound statements according to the philosophy of mathematical semantics read:

$$M[id:=e](\sigma) = assign(\sigma, id, M[e](\sigma))$$
$$\text{where: } assign(\sigma, id, val) = \sigma',\ \sigma'(x) = \begin{cases} val & \text{for } x = id \\ \sigma(x) & \text{for } x \neq id \end{cases}$$

$$M[s1;s2] = M[s2] \circ M[s1] \qquad \text{where } \circ \text{ denotes functional composition}$$

Note that denotations of composite phrases are given in terms of denotations of immediate subphrases and that we have avoided introducing a statement counter. For each additional language feature we may have to revise the definition of states, introduce new ways to compose denotations or even define new kinds of mathematical objects.

As a first complication we introduce a loop statement of the form: *while e do s*. We assume that *e* returns a truth value and intuitively expect that the denotation of the loop statement can be defined as:

$$M[\underline{while}\ e\ \underline{do}\ s](\sigma) = \begin{cases} M[\underline{while}\ e\ \underline{do}\ s](M[s](\sigma)) & \text{if } M[e](\sigma) \\ \sigma & \text{if } \neg M[e](\sigma) \end{cases}$$

The definition is of the form $f=F(f)$, with $f = M[\underline{while}\ e\ \underline{do}\ s]$ that is: f is defined as a fixed point of F. Before this definition can be accepted as meaningful, one has to ask whether such a fixed point always exists and whether it is unique. The existence can be asserted under appropriate mathematical restrictions (introducing concepts of monotonicity and continuity); there will in general be more than one fixed point satisfying the equation. Thus an additional rule has to be introduced which makes the defined object unique (the "smallest" fixed point under a suitably defined ordering relation). This is not the place to elaborate the issue at length. Chapter 3 discusses this issue in depth.

However, it should by now be evident that we are led into deep mathematical issues, and this at a stage where, from a programming language point of view, we have only introduced the most primitive language constructs. At this point, it seems that we have to consider the potential uses of a semantic definition. One should distinguish between the foundation of the subject matter and more practical problems like the description of real-life programming languages for compiler writers. Like the foundations of mathematics on the one hand and applied mathematics on the other, these two fields are not unrelated but are distinct. If we accept the program of mathematical semantics, the steps we have tried to indicate follow, and the difficulties observed above are inevitable. However, it seems unrealistic and in fact unnecessary to require that each compiler writer be fluent in modern algebra. Rather, one would expect that the foundations are used to justify, once and for all, useful practical methods which in turn can be applied directly by the practitioner.

A further important concept in most programming languages is that of local names, that is names which are declared for a specific textual scope of a program. The syntactic category is called block and takes the form:

$\underline{begin}\ \underline{dcl}\ id;\ st\ \underline{end}.$

Since the same name may now be used in different blocks for different purposes we have to introduce some device in the definition which enables us to distinguish the different uses of a name. One usually introduces an auxiliary object called environment, *env*, which is a function from names (variable names in the present example) to so called locations; the state then maps location into values. Thus a state σ is now a function of type $LOC \xrightarrow{m} VAL$, where LOC is a set of primitive objects called locations; the auxiliary object *env* is of type $ID \xrightarrow{m} LOC$.

In order to interpret a given phrase we always have to have an environment which associates the names occurring in the phrase with locations. The mathematical types of the denotations have to be revised so that *M[st]* when applied to an environment, yields a function which transforms a state. The types of denotations of the other constructs are designed similarly.

The last features to be discussed in this section are procedure declarations and parameter passing. What should the denotation of a procedure (in the sense of *ALGOL* or *PL/I*) be? According to the philosophy of mathematical semantics this must be an object which yields a state transformation ($\Sigma \to \Sigma$) when applied to the denotations of the arguments to the procedure. It is important that the procedure denotation is built in the environment where the procedure is declared. In some higher level programming languages, procedures can be passed as arguments. In particular, a procedure might be passed as an argument to itself. This concept presents certain mathematical problems and the establishment of a suitable domain of denotations is a major achievement [Scott 70a] - cf. chapter 3.

There are some language constructs whose treatment is not yet so widely accepted in the framework of mathematical semantics. In particular, parallel and quasi-parallel execution give rise to the use of rather complicated mathematical objects (see [Plotkin 76a, Smyth 76a] on "power domains").

Condition handling in *PL/I*; labels and goto's have been formulated, but, the models do not closely correspond to the intuitive concept of the construct, cf. chapter 5.

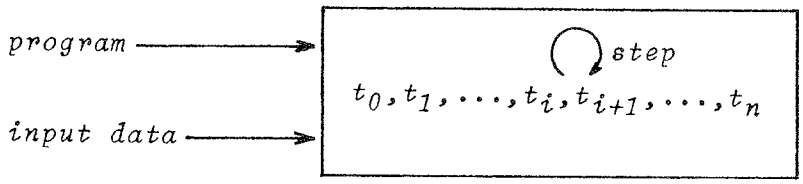
There are sizable language definitions in the denotational style (e.g. a definition of *ALGOL 60* [Mosses 74a], and a definition of a subset of *PL/1* [Bekić 74a]). There is an excellent introductory book on the subject by J.Stoy [Stoy 77a].

1.3.3 Operational Semantics

The semantics of a programming language can be defined via a hypothetical machine which interprets the programs of that language; such methods have been called "operational", or "constructive". The latter term is, however, misleading, because the specification of hypothetical machines

may contain non-constructive elements, such as quantifiers, implicit definitions and infinite objects. The term "definitional interpreter" is sometimes used instead of "hypothetical machine".

By machine we understand a structure consisting of a set of states, two subsets thereof: the initial states and the end states, a state transition function and a function which maps programs and their input data into initial states; also usually not given explicitly, there should be a function which takes end states as arguments and yields



t_0 ... initial state
step ... state transition function
t_n ... end state
$t_0, \ldots, t_n$ ... computation

an end state which is the result of the program. Since most higher level languages are such that the program remains constant, that is is not modified, during its interpretation, one could also keep the program separate and only include a statement counter to the currently executed statement within the state itself. The definition of the step function, if properly done, will reflect the syntactic structure of the language, such that we cannot only relate an entire program to a computation, but also sub-phrases of the program to sections of the computation, that is we may ask what a specific subphrase in a given context means. In languages, like *PL/I*, where the order of operations is not entirely fixed, the defining hypothetical machine is non-deterministic, that is the *step* function will, in general, yield a set of possible successor states and a program will thus be related to a set of computations.

For simple languages, mathematical and operational definitions of languages are very similar. In fact, if the latter are carefully constructed, there is a direct correspondence between such definitions. The strength and danger of the operational approach comes from its machine-like behaviour. For example, the VDL definitions showed that a language could be non-deterministic by building a control tree of possible next actions. The early operational definitions tended, however, to put rather too much in the state. For example, it is tempting to put the environment which was discussed above in the state: since environments change and have to be restored, one ends up with a stack of environments. This, the

"grand state" approach causes considerable difficulties in deducing properties about language definitions. It is possible to construct "small state" operational definitions (e.g. [Allen 72a, Plotkin 81a]) which offer many of the advantages of mathematical semantics but which avoid deep mathematical issues.

In the literature there exist various examples relating language definition to implementations (e.g. [McCarthy 67a]). A comprehensive elaboration of this subject would complement the existing material on the subject of syntax definition and parsing. The step from the syntax definition to the respective parser can be automatic. There are current efforts to master the semantics part of the problem, e.g. [Mosses 76a]. The subject is further elaborated in chapters 8 and 9.

The method has been applied to several large languages; in fact the proposed *ECMA-ANSI PL/I* standard [ANSI 76a] has been formulated using an operational definition. The method is the only one presently known which is capable of covering the currently existing language constructs. There is an introduction to the subject by A.Ollongren [Ollongren 75a], and several summaries, e.g. [Lucas 69a] on VDL and an in depth evaluation by J.Reynolds [Reynolds 72a].

1.3.4 Axiomatic Approach

Each of the two approaches so far described provide models for the languages to be defined. In contrast, the axiomatic approach implicitly defines the semantics of a programming language by a collection of axioms and rules of inference, which permit the proof of properties of programs, in particular that a given program is correct, that is realizes a specified input/output relation. Of course, one can prove assertions about programs using either a mathematical or operational definition and ordinary mathematical reasoning. In fact, the axioms and rules of inference can be regarded as theorems within the framework of mathematical semantics.

However, the objective of the axiomatic method is a formal system which permits the establishment of proofs using only the uninterpreted program text (that is without referring to denotations of the program or program parts). Whenever we talk about denotations in this section, this is for explanatory purposes and is not part of the axiomatic system.

The problem of correctness proofs of programs is usually split into two subproblems; the first is conditional correctness (that is correctness under the assumption that the execution of the program terminates); the second is the proof that the program terminates. Until further notice this section deals with conditional correctness.

To illustrate the approach we will refer to the simplest language level of section 1.3.2 above, that is a fixed set of variables, assignment and compound statements. The notation and particular axioms of the example are due to C.A.R. Hoare [Hoare 69a]. The basic new piece of notation are propositions of the form:

$$p1\{st\}p2$$

where $p1$ and $p2$ are propositions referring to variables of the program, and st is a statement. The intuitive meaning of the form is: if $p1$ is true before the execution of st and the execution of st terminates, then $p2$ is true after the execution of st. $p1$ is called pre-condition, $p2$ is the so-called post-condition or consequence.

The axiom (more precisely the axiom schema) for the assignment statement reads:

$p_e^x\{x:=e\}p$ $\quad$ p_e^x means: replace all free occurrences of x in p by e

In fact, p_e^x is the weakest possible precondition given p. Conversely, given p_e^x as the precondition, p is the strongest possible consequence. That is the schema captures all there is to know about the assignment statement. A specific instance of the schema would be:

$$0<x+1 \ \{x:=x+1\} \ 0<x$$

Note that in order to use the schema it is not necessary to refer to the denotation of $x:=x+1$.

The definition of the compound statement takes the form of a rule of inference and reads:

$$\frac{\underline{IF}\ p1\{st_1\}p2\ \underline{AND}\ p2\{st_2\}p3}{\underline{THEN}\ p1\{st_1;st_2\}p3}$$

For a full language definition there will usually be an axiom per primitive statement and a rule of inference per composite statement; in addition, there are some general rules which have not been exemplified in this section.

The structure of the proofs reflects the syntactic structure of the program text, as one would hope.

There is a simple relation between the discussed axiomatic approach and a corresponding definition using mathematical semantics. As already mentioned the axioms and rules of inference can be interpreted as theorems within mathematical semantics. In particular we interpret the new propositional form $p1\{st\}p2$ as follows. Assume for the moment that $p1$ and $p2$ are expressions that are also valid expressions in the programming language, denoting truth values.

$$p1\{st\}p2 = M[p1](\sigma) \supset M[p2](M[st](\sigma))$$

for all σ for which $M[st]$ is defined, that is st terminates.

The various axioms and rules of inference may now be rewritten according to the above interpretation and proven with respect to the definitions of mathematical semantics (see [Manna 72a]).

Neither the generation of the proof nor solving the termination problem can be completely mechanical, since both are in general undecidable. However, there is hope that, for frequently occurring program structures, the problems can be solved effectively by algorithms. Proposals to solve the termination problem frequently rely on an indirect proof (in particular on finding a quantity which decreases as the computation proceeds, but cannot decrease indefinitely).

The subject of axiomatic definitions and program verification has stimulated widespread research activities due to the intellectually pleasing content and its potential economic value. The belief in the value is based on the vision that program testing can ultimately be replaced by systematic program design and verification and possibly to some extent automated.

There are many examples of correctness proofs of specific programs (see [London 70a]) and several automated verification aids (e.g. [King 75a,

Boyer 79a, Good 78a, SVG 79a, Lee 81a]). The existing examples are mostly small programs for complicated mathematical problems. Some of the algorithms published in the respective section of the *CACM* are certified by proofs. An attempt to axiomatize a full language, *Pascal*, has been undertaken by Hoare and Wirth [Hoare 73d] resulting in the definition of a large subset (see also [London 78a] on *Euclid*).

Intimately connected to axiom systems for programming languages is the issue of programming style and development methodology. The essence of structured programming is the recommendation to use only language constructs which have simple axioms (this excludes, for example, the general form of goto statements, although restricted forms may well lead to simple correctness arguments). As it turns out, the process of developing a program is intimately connected to the generation of the corresponding correctness proof. Thus we obtain guidance on how to develop programs rather than merely learn how to prove ready made programs correct.

There is, at present, an enormous gap between, on the one hand, current programming practice and the complexity of the software being produced and, on the other hand, the vision and capabilities of the systematic techniques described. The proper discussion of the dilemma needs a larger context than has been given in this section and will therefore be deferred to the next section.

1.4 CHALLENGES

The scope of this section excludes topics considered to belong to the theory of computation. With this restriction in mind we may certainly say that the definition of programming language semantics is not an end in itself; consequently, the discussion of research challenges cannot be isolated from the intended applications of semantic definitions (that is precise definition of real life languages, compiler development, program development and language design).

There are two topics that should be clearly separated to avoid confusion: firstly, the semantic analysis and formal definition of existing programming languages; secondly, the design of novel, useful language constructs.

Current programming languages are a compromise between the desire to pro-

provide the most comfortable and elegant language for the human user and the aim to construct efficient implementations on given systems with known compiler technology. Furthermore, the more intensely used languages undergo an evolution over the years to support new system functions. It is now important to design languages with the aim to make formal correctness proofs easy or to fit into the framework of mathematical semantics. However, it would be a mistake to conclude that existing languages are no longer worth the attention of computer science. In view of the heavy investment by users as well as manufacturers it is not likely that the current programming languages will change radically in the near future. Thus the carriers of new programming style will be, at least for some time, current languages. The initial motivation of formal semantics, precise definition to achieve portability, is still valid; there is a need for semantic analysis of *COBOL* (the most widely used programming language). A comparative language study on the semantics level would be quite valuable [Strachey 73a]. Finally, there should be a comprehensive representation of the existing implementation techniques related to formalized semantic concepts.

Whereas *BNF*, or variations thereof, are widely accepted as a means to define a concrete syntax, there is no such widespread consensus for any of the semantic description schemes. Finding such an agreed semantic meta-language should be treated as an urgent problem.

Next I wish to offer a top down argument to justify the major long range goals of the present subject. Firstly, we can observe that over the past two decades the speed and storage capacity of computers have been increased roughly at a rate of about 40 percent a year. This trend has been balanced by a similar decrease of cost per operation and per storage unit. Similarly the size of system-code (operating system, compilers, etc.) has increased exponentially as well. However, in this case no balancing trend of decreasing cost per line-of-code can be observed. Furthermore, we will not only have to master greater quantity but larger complexity as well. We conclude that software production is or soon will be the bottleneck for the use of computers unless some progress is made on three general research directions promising to improve the situation:

1. Advance Automatic Programming
2. Remove Testing in Favor of Correctness Proofs
3. Advance Modular Programming

By the first research area we mean to extrapolate the development of "very high level languages" by introducing more abstract data-types (e.g. sets) and their associated operations; relax restrictions in current languages and introduce more powerful control structures. The intent is, of course, to automate part of the production process: in short, to follow the trends suggested under the term "very high level language". Topics one and two are intimately connected. As J.Schwartz [Schwartz 75a] observes, it is much easier to prove the correctness on an abstract level rather than on the level of detailed representations. If the abstract program can be compiled, the task of the programmer is completed, provided the compiler has been proven as well. Thus the step from the abstract algorithm to its ultimate representation in machine form is proven once and for all by a compiler proof. The author believes that research in correctness proofs must therefore be investigated hand in hand with the development of very high level languages. Even under the assumption that the level of programming languages can be raised, correctness proofs will remain sufficiently complicated to warrant machine assistance in the form of proof generators and checkers. Although study of the latter subject has advanced over the last decade, it has not yet reached the stage of applicability in practical programming.

Various subgoals may be envisaged, e.g. conversational systems like *EFFIGY* [King 75a] which offer a combination of generalized testing by symbolic execution and some assistance for generating proofs. A notorious problem in designing large pieces of software is modularity. It is rarely the case that existing modules can be used to build new systems without major trimming. As J. Dennis [Dennis 75a] observes, the success of modular modular programming not only depends on how modules are written, but also on the characteristics of the linguistic level at which these modules are expressed. Dennis supports this observation by a detailed analysis of some high level languages. Modules are usually expressed by procedures, subroutines or programs (depending on the specific languages used). In short, we have to look for constructs other than procedures and the related traditional ways to compose procedures into larger units, in order to achieve the desired modularity.

In conclusion we ask what is the relevance of formal semantics to these issues? Firstly, axiomatic semantics provides the proof theory for program correctness proofs, and thus is also the basis for the mechanical aids in this area. It is difficult to propose useful axioms and rules of inference without having an interpreted system (such as provided by a

mathematical or operational system).

In search for new language constructs (such as a useful notion of module), formal semantics ought to provide the framework for formulating the problem and for stating and justifying solutions [Strachey 73a]. So far, most research in formal semantics has been concerned with constructs as found in traditional languages. ("Here is a piece of language, what does it mean?") In order to tackle new applications we should start from the other end: construct novel denotations and associate a name after we are satisfied with their properties.