SEMANTICS - THE MAPPING OF PROGRAMMING LANGUAGES INTO SUBSETS OF THEMSELVES

Professor A. J. Perlis

Department of Computer Science,
Schenley Park,
Carnegie Mellon University,
Pittsburgh, Pa. 15213,
U.S.A.

Abstract:

Questions of semantics are categorised into those appertaining to a programming language and those relating to programs written in the language. Criteria for a semantic description and some methods in current use are discussed. Finally, one such method, the language reduction method, is described in more detail and is shown to be extremely useful in teaching.

Rapporteurs:

Dr. J. Eve
Mr. A. Alderson

1. Introduction

Professor Perlis began with the comment that in Universities we are learning about programming languages at the same time as we teach them, with the result that courses at both graduate and undergraudate level are properly undergoing rapid transition. New ideas and notations are being introduced, underlying strategies are changing; the overall goal of teaching people to program and use computers, however, remains unchanged.

Commencing with Algol 60, syntax was emphasised (perhaps unduly so) in the teaching of programming languages. Professor Perlis expressed the view that syntax analysis has now receded into the background — a background which should be part of the framework of understanding of every computer scientist, but not something with which he needs to concern himself actively. Few people in Universities today are primarily involved with syntax analysis as such, although it is true that there has been little in the way of experimentation into, or tabulation of, the relationship between parsing method and our choice of grammars for languages. There is a tendency to forget that in language design there are three 'variables' available other than the meaning to be attached to the constructs in the language. These are:

1. the base alphabet;
2. the grammar;
3. the parsing method.

In teaching programming, Professor Perlis stressed that a student should understand the importance of these four factors in language design and that all four must be examined, in one way or another, in making decisions relating to the design or use of a language. The difficulty of using APL from Teletype terminals was cited as a simple example of the importance of the base alphabet. In this case, the restricted character set of the terminals is inadequate to cope with the rather rich base alphabet of APL.

Dealing very briefly with the first three 'variables', Professor Perlis commented that very little work has been done on the size of a minimum base alphabet which would be useful and convenient to programmers. On parsers and grammars, he emphasised that the grammar should not only meet the needs of the user as the carrier of semantic information but also the needs of the parser, permitting efficient parsing.

2. Aspects of semantics outside the realm of programming

The growth of interest in semantics was, in the speaker's view, not entirely unrelated to the solution of the more interesting syntactic problems. In consequence, attention was turned to semantics and it has been natural to think of semantics as naturally, logically and mathematically following syntax. The pioneer paper in this respect, that of McCarthy (1965), posed semantic problems dealing with the correctness and termination of programs. Although conveyance of a proof must depend upon the language in which the program is described, these questions are independent of the language in the sense that they remain whatever language the program is written in.

In Professor Perlis' view, questions of correctness and termination can only be of importance in programming at the intuitive level. The question of correctness of a program (other than a program embodying a very elementary algorithm) soon proceeds outside of the bounds of programming per se, and becomes a semantic question which is meant to be answered by the program; the answers to such a question must reside in those other disciplines, mathematics, sociology, artificial intelligence, etc., from which the program originates.

Any theory of semantics of programs would have to answer the question of whether two programs computing a mathematical function, by different means, are computing the same function. To derive the propositions which convey the equivalence of two such programs in a functional sense involves a rapid descent into mathematical analysis, i.e. the algorithmic context in which the problem originated soon becomes irrelevant and only after much work in analysis, which is relatively unrelated to programming, can a return be made to the displays of two pieces of text with propositions establishing equivalence.

As a second example Professor Perlis mentioned a comment of Professor Bauer's, that round-off error will sooner or later cause the Newton method for finding the zero of a function to generate an iterate which is sufficiently near a root for the iterative process to converge. It seemed unlikely that any theory would produce an algorithm for establishing program correctness which would uncover this truth.

These examples illustrate that problems of equivalence, correctness and termination depend on issues lying far outside the displayed text

of a program. They are, in a way, outside the framework of programming except in the intuitive sense, that people learn to write correct programs; in this context the ideas that Professor Dijkstra discussed are extremely important.

Professor Perlis proceeded to argue that semantic questions relating to equivalence and correctness are really questions in mathematics and to supply answers to them would require a mathematical education of a rather formidable kind. They would not, therefore, fall within a course in programming within a University. On the other hand he pointed out that if it was assumed that we should teach people how to write and prove that programs work, suggestions that mathematics is irrelevant to computing science are nonsense. As a middle way, it was suggested that it may be possible to isolate, and teach, a number of equivalence relations within languages which a student could learn and use; safe in the knowledge that writing in one way is equivalent to writing in another way and that embedding such statements in other statements does not destroy the equivalenc. While Professor Perlis doubted that such a set of relations could ever be complete in that correctness would follow from their use alone, he felt that the teaching of programming was largely concerned with such equivalences.

3. Aspects of semantics within the realm of programming

A different class of questions which are independent of the programs written in the language but which deal only with the language were exemplified by the following. Does the language permit dynamically varying data structures? Does it permit decomposition into simpler subsets? What kind of symbol table structure is useful (or required) to build a compiler? Is a stack model adequate for run time storage allocation? What are the binding times of the various semantic components of a language? (That is, when does an expression achieve its final form, beyond which it behaves as a constant?) When can one introduce definitions into a running program?

Unlike the first set of questions which concern an individual program and which, therefore, require analytical tools heavily dependent on the particular program and relatively independent of the language the latter questions are semantic questions concerned solely with the language itself. It is these questions, the speaker suggested which are properly the concern of the computer scientist. Further, he felt that the semantic description

of a language should be independent of particular programs and apply over the class of all programs in that language.

The idea that it is necessary to give an interpreter for a language in order to explain the meaning of it, was not one that Professor Perlis found satisfying. This method, used by the Vienna school, involves building an interpreter in a 'universal notation' which is an expanded form of LISP. Several such descriptions have now been produced including one at Carnegie Mellon University for APL — 'a rather horrendous description, to say the least, although APL is a very simple language'. A method less dependent on the notion of program execution, if possible, would be attractive.

4. Criteria for a semantic description

Professor Perlis referred to and commented upon the criteria of de Bakker (1969) for a method of semantic description.

1. It should be applicable to all programming languages. The difficulty here is to achieve generality and precision without descending into 'the Turing Machine tar-pit'.
2. The vague notions of readability, transparency, preciseness and elegance of description are involved.
3. Is it possible to leave the definition of parts of the language completely or partially open? For example, does the definition provide information about the division of the actions to be performed at compile time, run time, etc.?
4. How much insight is gained into the properties of the language? The description should certainly indicate the defects which a language inevitably has.
5. Is it possible to reflect independent concepts in the language in independent parts of the description? Would language changes which are small in some sense be expressible by small changes in the description or could they increase the complexity of all parts of the description arbitrarily?
6. Can one obtain proofs of properties of the language from the semantic description?

5. Methods of semantic description

Several methods based upon the execution of programs are now taught in courses in programming. Perhaps the earliest, due to Van Wijngaarden (1962) and later developed in De Bakker's Ph.D. thesis (1967),

uses a 'Markov Algorithm machine' which would execute Algol 60 text directly. Landin and Strachey map programs into a representation depending intimately on the lambda notation, with a 'machine' which executes such descriptions. The Vienna method has been used to provide descriptions of PL/1 and Algol 60 and has achieved important successes infinding errors in the PL/1 compiler although it is not of course known how many errors it has not detected.

Professor Perlis felt that Van Wijngaarden's early paper had been rather overlooked although it offered an approach with a great deal of value, both pedagogic and to compiler writers. This approach used a process of reduction, of mapping Algol programs using the full syntax of Algol 60 into programs using only a subset of that syntax. In that the major role of syntax is as a carrier for a large part of our semantic information, this reduction process applies to semantics. Using this approach goto statements can be replaced by procedures, switch and for statements can be eliminated, etc. In the process some of the major trouble spots of Algol were eliminated, i.e. defined, by the process of reduction.

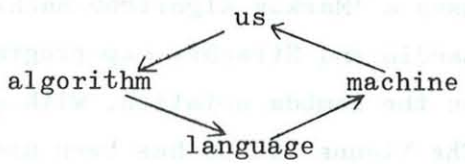
6. The use of language reduction in teaching

Professor Perlis outlined a first course in programming which he has taught using the reduction technique; the three purposes of the course were:

1. To teach students to program; a high level language, Algol, was selected as the carrier language.
2. To teach students how programs are executed and because they know little about computers.
3. To teach them about the devices which execute programs.

The problem is how to make the transition from a language like Algol to the computer. Algol, while very complicated, contains a number of conceptual simplicities in it for writing programs, while the computer contains a very different set. The object is to make the transition in such a way that a student sees the whole process as a natural transformation of his algorithms from his thoughts to Algol, to binary programs which are executed on the machine. As the results are invariably incorrect, the cycle is repeated iteratively.

The diagram representing this process, Professor Perlis pointed out, has the nice property that at any node there is a whole series of identical lozenges behind it. Always, this quartet must be considered, all members of which are equally important.



'The cybernetic dialogue'

Starting at the programming phase one teaches top-down; complex tasks are reduced to simpler tasks so that one always works with conceptually simple structures. At this level it can be said that one programmer can tell another what he is doing. Unfortunately, this cannot persist — algorithms which are simple at one level become enormously complex at another. One of the things to be taught is the replacement of a program by a program which is better in one of the two senses, more general or more special and more efficient. In neither case is there any guarantee that they be able to create an algorithm which is simple and can be explained to someone else. It is important that they should thereafter be able to create a better one and as a student proceeds along this chain he will pass the point where no one on earth can understand it except himself and after a while he will not either. In this process he learns to reject constructs which are more inefficient than are necessary for this problem, e.g. removing procedures.

In the other direction they may introduce procedures to aid generality. From the beginning they are thus encouraged to consider the mapping of Algol programs into other Algol programs which are better in some way. One of the ways in which it must be better is that it runs better on a computer. In view of its importance, students should be expected to undertake such transformations many times.

In providing an explanation of the meaning of a program, Professor Perlis' aim was to provide a meaning in programming rather than mathematical terms, i.e. the meaning of the text in terms of what happens. To have used an Algol interpreter in the Vienna notation would have involved '18 out of the 16 weeks available explaining how the notation works'. Instead the process of reducing Algol programs was adopted. By a succession of steps, an Algol program is reduced to a transparent structure which, while not strictly Algol, is easily understood and is very similar to the structure of the machine on which the program is executed. In the final step the

description is at a very primitive level. There is only one array called MEMORY; there are no scalars only array elements; every expression has at most one operator; all procedures have two parameters; there is no block structure, no For loops, no switches. In fact, it is machine code in another notation. When better machines are built, the reduction process can stop at a higher level. A further step would be the reduction of this description to the PMS (Processor, Memory Storage) notation developed by Gordon Bell and Allen Newell. While not descending to the circuit level, this notation contains all the logical design information so that the resulting machine could in principle be built.

This was the content of a course presented in 32 one-hour lectures. Of a class of 150 students about 20 liked it, felt it appropriate and worthwhile.

7. Comments on the language reduction process

Professor Perlis summarised some additional reduction processes to those introduced by van Wijngaarden.

1. The first step is to ensure that all identifiers are distinct. (The mapping of identifiers onto stacks is a quite separate semantic process which is not specified in the definition of Algol. It is the interpretation of block structures which indicates that two variables may occupy the same location and this must be deduced.) In the lexical process comments are stripped out.
2. Maps of statements, e.g. closing procedures to blocks; closing programs to blocks by adding a standard prologue of support routines.
3. Producing mappings of expressions, e.g. in array bounds all expressions which could be real in value are rounded properly; all conversions of types are inserted explicitly. All actual parameters are replaced by procedures and a stack of integers is introduced to receive control information, etc.

Professor Perlis indicated that the process had been worked out in detail for Algol and it was his intention to apply it to other languages. As an example for further illustration he took APL which, although superficially complex, is simple in both its logical and operator structure. The semantics of APL language can be reduced to two subsets. For example, the operators

$$\varphi \quad \in \quad \uparrow \quad \downarrow \quad \emptyset \quad / \quad \backslash \quad \iota \quad \rho$$

are all merely ways of extracting arrays from arrays. These operations can all be described in terms of subscripting and sequencing through arrays which require only the concepts of subscript, scalar operation and goto. The outer product, inner product and reduction operators are more complex, they require some sequencing control; outer product is the only way of generating for loops in APL, i.e. for generating iterative computation inside a statement. However, the sequencing control required is very simple, nothing more than two nested for loops. Thus, the richness and apparent complexity map very quickly to very simple things — three array accessing routines.

1. A row by column routine, which peels off elements in standard order.

2. A three level routine which is used in reduction, inner product, rotation and transposition, where one indexes most rapidly on the k^{th} subscript and the rest of the subscripts are indexed in standard order.

3. An n-level routine (assuming the array has rank n). In the case where array elements are subscripted a more general routine can now be written which handles them. An interesting feature of this routine is that, with the aid of a stack, the routine may be written independently of the rank of the array.

8. APL compilation

As APL can be reduced to such a simple base, Professor Perlis raised the question of a compiler for APL (assuming that the interactive features of the APL system are ignored and that only the language is considered). Other than the nature of the system, which allows a user to change a program interactively, there are two reasons for executing the language interpretively.

1. The rank and shape of arrays may change dynamically during execution.

2. So much time is spent in the efficiently coded array processing routines and so little in parsing that probably there is little to gain.

There exist APL programs which remain static and compilation, if possible, would be appropriate in those cases, so the problems of constructing a compiler were considered. Studying the semantic structure of APL indicated that:

1. Its variable structure is very simple; block structure is absent; parameters are passed, in effect, by address, so no stack mechanism is needed to resolve the scope of identifiers.
2. Assuming an APL function can be isolated, its logical structure is much simpler than that of an Algol function.
3. The statement structure is very simple, assignment and a primitive conditional are all that exist.

A model was constructed which could be realised as either compiler or interpreter of which the following are the salient features. In processing a statement, no more of an array than is needed is produced. For example, in the operation of reduction it is not necessary to produce the whole of an array which is the consequence of some expression before carrying out the reduction. This operation can be expressed as a generator using the three basic array routines. This generator need have no previous knowledge of dimensions. The expression to be reduced, which puts its values on a stack one at a time, is also in the form of a generator which keeps the following information: type, name, where in the generator the cycles begin, where the result (always scalar) is produced and where it finishes. The expression generator runs producing values one at a time, which are overlayed in the same stack location. The expression generator and the reduction operator generator are linked as co-routines, the latter obtaining its result from the stack just after the former has placed it there.

The outcome of this study is a compiler which is structurally quite simple, but also quite inefficient. The conclusions reached are that a compiler can be built and that it should not be, because of inefficiency. The estimate, that compiled APL would run 1.8 times as fast as the interactive APL system, does not represent a major gain. In Professor Perlis view, continued popularity of APL is more likely to result in a machine which will process it directly. As the system is a proprietry secret, without undertaking the semantic reduction of the language, sufficient understanding, to construct a compiler, would not have been possible. Nor would it have been clear that a compiler should not be built.

9. Conclusions

Professor Perlis said that he felt that language reduction offered an appealing way to cross the bridge between the writing of programs

in a language and getting them executed by a machine. Having spoken principally of an approach that educators may take of reduction his final comment was to language designers. The lack of enthusiasm for Algol 68 did not imply stabilisation on existing languages, as BLISS and L* recently designed at Carnegie-Mellon testify. The process of semantic reduction offers a way of seeing what constructs in a language mean and, since language designers must always be concerned with implementation, how the constructs can be implemented. Further it shows what additional constructs are necessary to make the process easy.

10. Discussion

In response to Professor Higman's query concerning additions to Algol 60 concepts which emerged during its reduction, Professor Perlis said these were in effect the lambda notation and actual parameters and, secondly, a means of finding addresses. Replying to a series of questions by Professors Pilloty and Randell relating to the apparent unpopularity of the course with the great majority of its students, Professor Perlis made a number of points. These students felt it to be a great deal of fuss over nothing — they really wanted to learn Fortran to write programs to solve problems in their own fields, whereas the purpose of the course is really to teach the concept of a programming language and the concept of a machine. At the same time they are provided with a language, Algol, which is a useful tool that they might use. The problem with the first course is that there are so many things to be conveyed, it being the only course that many students attend. Professor Perlis agreed that the course did act as a filter in selecting out the potential computer scientists. He felt that as a teacher, it was necessary that he take a long term view and that even though the majority of students did not like the course they still learnt to program; they would certainly learn Fortran subsequently anyway and with little difficulty since programming languages are all similar. One of the positive things they learnt is that only a few of them are interested in computer science and these few have a picture of some of the major things with which they will be concerned. In addition to this first course in programming there are short courses available in a variety of languages which are taught by graduate students.

Dr. Scoins asked if there was a particular reason for teaching algorithms, languages, machines in this sequence as shown in the lozenge diagram. Professor Perlis said that this was the natural sequence in which

we work and that he could not see how to teach machines except by algorithms nor how to get algorithms to machines except by languages. The important thing here was that there was recycling around the four components but he agreed with Professor Higman that there could be inner loops within the diagram, for example, around the machine if a compiler makes supposed corrections.

Professor Bauer suggested that from some languages it is easy to learn others but not necessarily vice versa. Professor Perlis concurred stating that this is a positive reason for teaching Algol 60 as it contains so many more of the important ideas than Fortran. He added that an important issue in education for the future would be the effect of terminals. Teaching by terminal, he felt, only awaited development of a cheap enough terminal. Another important topic would be the design of languages which work well with terminals. Algol 68 failed grievously on this issue.

Professor Pengelly asked the reason for using a formal description of the machine such as PMS. Professor Perlis replied that it acquainted the student with the fact that at the hardware level things became totally specified and rigid and hence they should go through this process of specifying something to the smallest detail and so the PMS notation is used.

11. References

de Bakker, J. W. (1967): 'Formal Definition of Programming Languages: with an application to ALGOL 60'. Mathematics Centre Tracts 16, (Mathematisch Centrum, Amsterdam).

de Bakker, J. W. (1969): 'Semantics of Programming Languages'. Advances in Information Systems Sciences, Vol.II, Plenum Press (New York and London) p.173.

McCarthy, J. (1965): 'Problems in the Theory of Computation', Proceedings of the IFIP Congress 65, Spartan Book (New York), Macmillan & Co. Ltd. (London) p.219.

van Wijngaarden, A. (1962): 'Generalised Algol', Proceedings of the ICC Symposium on Symbolic Languages in Data Processing, Gordon & Breach (New York and London) p.409.