BULLETIN OF THE
AMERICAN MATHEMATICAL SOCIETY
Volume 80, Number 2, March 1974

SEMANTIC AND SYNTACTIC ISSUES IN PROGRAMMING[1]

BY J. T. SCHWARTZ

1. Introduction. Idealized computer models and abstract algorithms. The other speakers in this tutorial symposium will discuss recent theoretical developments in computer science. The studies they will describe have a combinatorial flavor which all mathematicians will find very familiar. My own task is the less tractable one of presenting the immediately pragmatic side of computer science; specifically, programming, and the content of investigations into programming technique. This is work, very directly rooted in the fertile muck of everyday industrial practice, out of which grow the more theoretical endeavors to be explained in the other lectures.

The goal at which programming technique aims is the rigorous, correct, and maximally clear and simple description of complex processes. This description must also be such as to permit efficient implementation on a computer. Rigor and correctness are necessary because programs are acted on by a device (the computer) lacking all but the most rudimentary ability to make inferences or to distinguish between the reasonable and the unreasonable. Clarity and simplicity of programs are essential because of the inherent limitations of the human mind, limitations which the activity of programming always makes painfully evident. It deserves to be noted that large programs, the most complex artificial objects known to mankind, generally threaten at every moment to submerge their creators in a flood of complications. Efficiency is important, not only because computer time is still an expensive commodity, but also because "effective" mathematical procedures can sometimes imply calculations so explosively large as to require astronomically long running times on any conceivable physical device.

In this expository talk I shall explain some of the notions which guide (or ought to guide) programmers in their attempts to develop

A lecture delivered as part of a special tutorial introduction to computer science at the Annual Meeting in San Francisco on January 13, 1974; received by the editors October 19, 1973.

AMS (*MOS*) *subject classifications* (1970). Primary 68A05, 68--02.

[1] Work supported by National Science Foundation, Office of Computing Activities, Contract NSF-GJ-1202X.

clear, efficient, and rigorously correct representations of important processes. I begin by distinguishing the *semantic* and the *syntactic* aspects of programming. By semantics I mean the description of processes, notational issues being ignored. By syntax I mean the study of the specifically notational issues which programming must face. Though semantic issues are more fundamental than syntactic, it will readily be understood that in describing extremely complex processes by texts tens or hundreds of thousands of lines long the choice of helpful notations is of great importance.

In a semantic study one wishes to describe processes with minimum attention to notation. One of the best ways of doing this is by modeling computation using some type of abstract automaton. Various models of this kind have been proposed in the literature. Each model highlights some aspect of the process of computation significant to the proposer of the model. For example, the important Turing machine model of computation brings into focus the fact that any deterministic computational process can be realized by the iterated performance of a very small number of simple and highly stereotyped operations. The Turing model also allows one to establish basic quantitative measures of computation size, measures on which rest rapidly developing theories of computational complexity. Other abstract computer models mimic more directly the action of presently available or of physically conceivable computers.

We shall begin our semantic discussion by defining an abstract computer model, which, though it hides important efficiency-related questions, allows us to approach certain central semantic issues very directly. Within this computer model infinitely many abstract cells

$$\cdots, C_{-j}, \cdots, C_{-1}, C_0, C_1, \cdots, C_k \cdots$$

are available. At any step of a computation, each cell contains either an integer, a vector, or a (finite) set. We allow sets to have elements which are themselves sets, and allow vectors to have components which are themselves vectors.

During every computation all but a finite number of cells will always contain the integer zero. The cell C_0, which will play a special role in our model (it is the so-called *instruction location counter*) is always required to contain an integer (rather than a set or vector).

To compute in this model, we first prescribe the initial contents of finitely many cells (all the others contain 0). After this 'initialization', *execution* starts, and the contents of the cells C_j change, cycle by regular cycle, in accordance with specific *execution rules* (these are given in more detail immediately below). At some later time, the computer will *stop* (we may think of it as ringing a bell to announce that it has done so). At this time, the object (set, vector, or integer) left in some designated cell (as, e.g. C_{-1}) is the *result* of the computation.

The execution rules of our model determine the manner in which the contents of each cell C_j changes during each cycle of execution. These rules are as follows. Suppose that at the start of a cycle the cell C_0 contains the integer m. Then the content of the cell C_m is retrieved. This must be a vector

$$\langle n_1, \cdots, n_k \rangle \tag{1}$$

whose components must be integers (if not, or if (1) is of inappropriate length, execution will halt). The component n_1 of the *instruction vector* (1) designates an *operation* to be performed; components $n_2, \cdots, n_k$ designate *parameters* of the operation. Our abstract machine is furnished with a certain number π of *primitive* operations; these are designated by integers n_1 lying in the range $1 \leqq n_1 \leqq \pi$. Depending on whether or not n_1 lies in this range, the operation vector (1) is interpreted in one or another way, as follows:

Case 1. *n_1 designates a primitive operation, i.e.,* $1 \leqq n_1 \leqq \pi$.

In this case, the primitive operation designated by n_1 is performed. The repertoire of primitive instructions with which our model is furnished can be chosen rather arbitrarily; we shall soon see that this choice is not a very critical one. Suppose for the sake of illustration that $n_1 = 10$ designates the 'power set' operation, that $n_1 = 11$ designates set intersection, and that $n_1 = 12$ designates addition of integers. Then

(a) The instruction vector $\langle 10\ n_2\ n_3 \rangle$ is interpreted as follows: fetch the value u in the cell C_{n_2}. This must be a set (otherwise execution halts). The power set 2^u of u is formed and placed in the cell C_{n_3} (erasing the former contents of C_{n_3}).

(b) $\langle 11\ n_2\ n_3\ n_4 \rangle$ is interpreted by fetching the values u and v held in the cells C_{n_2} and C_{n_3} respectively. Both u and v must be sets. Their intersection $u \cap v$ is formed and placed in the cell C_{n_3} (erasing its former contents).

(c) $\langle 12\ n_2\ n_3\ n_4 \rangle$ is interpreted by fetching values u and v from C_{n_2} and C_{n_3}, respectively; both these values must be integers. Their sum $u+v$ is formed and placed in C_{n_4}.

It is clear that this style of interpretation allows us to regard any set of transformations as the family of primitive operations of an abstract computer model. More precisely, *any recursive function whatsoever can be a primitive operation of our abstract computer*. Of course, some finite selection of primitives must always be made.

Case 2. *The operation code* n_1 *of the instruction vector* (1) *does not designate a primitive operation, i.e.* $n_1 < 1$ *or* $n_1 > \pi$.

The important issue which arises in defining the interpretation rule to be applied in this case is that of *extending the family of primitive operations of the abstract computer*. In the nonprimitive case, i.e. if $n_1 < 1$ or $n_1 > \pi$, the content v of the cell C_{n_1} is fetched. We require that v be a vector $\langle m_1, \cdots, m_l \rangle$ with integer components (otherwise execution halts). The vector $\langle m_1, \cdots, m_l \rangle$ is prefixed to the final part $\langle n_2, \cdots, n_k \rangle$ of (1), to produce

$$\langle m_1, \cdots, m_l, n_2, \cdots, n_k \rangle, \tag{2}$$

and the instruction execution rules that have just been stated are applied to the *new instruction vector* (2) rather than to the original instruction vector (1).

The following codicil completes the definition of the execution rules: If during a given cycle of execution the operation executed does not of itself change the integer contained in the cell C_0, then at the conclusion of the cycle the integer in C_0 is incremented by 1. This ensures that execution will progress in orderly fashion from one instruction to the next, except when C_0 is deliberately changed.

The execution rules just stated can be exploited most easily if we suppose a number of simple but generally useful primitives to be available in our abstract machine. We shall list three such: a 'linking' primitive, a 'conditional assignment' primitive, and an 'indirect addressing' or 'move indirect' primitive.

(a) The linking primitive $\langle 1, n_2, \cdots, n_k \rangle$ is executed as follows: the content m of C_0 is retrieved, and the vector $\langle m+1, n_3, \cdots, n_k \rangle$

formed. This vector is placed in the cell C_{n_2}; n_2+1 is placed in C_0, and execution continues.

(b) The conditional assignment primitive $\langle 2, n_2, n_3, n_4\rangle$ copies the value in C_{n_3} to C_{n_4} if C_{n_2} contains the value 0; otherwise it has no effect.

(c) The 'move indirect' primitive $\langle 3, n_2, n_3\rangle$ is executed as follows. The contents m_2 and m_3 of C_{n_2} and C_{n_3} are retrieved. If m_2 and m_3 are both integers, the content of C_{m_2} is copied to C_{m_3}; otherwise, execution halts.

We now note that a small collection of primitive operations like those which have been described allow any desired (recursive) transformation to be expressed, and to be treated as a new primitive. Rather than proving this far-reaching observation formally, we will illustrate it with an example. Suppose that we desire to add, to the repertoire of primitive operations of the abstract computer, a new operation $\langle j, n_2, n_3, n_4\rangle$ which takes the contents u and v of C_{n_2} and C_{n_3} respectively, forms the intersection $2^u \cap 2^v$ of the power sets 2^u and 2^v, and puts this intersection into C_{n_4}. This can be accomplished as follows. We reserve 18 cells $C_j, \cdots, C_{j+17}$ which are not needed for any other purpose; j must lie outside the range $1 \leqq j \leqq \pi$. Into these cells we put n-tuples representing the following primitive operations of the abstract computer:

	Cell		*Contents of Cell*	
	C_j	contains	the *link* primitive with parameter $j+6$	
	C_{j+1}	contains	the integer $j+4$	
	C_{j+2}	contains	the integer $j+5$	
	C_{j+3}	contains	arbitrary value	
	C_{j+4}	contains	arbitrary value	will be changed by the
	C_{j+5}	contains	arbitrary value	instructions which follow
	C_{j+6}	contains	arbitrary value	
	C_{j+7}	contains	primitive 'put 1st component of vector C_{j+6} into C_{j+3}'	
(3)	C_{j+8}	contains	primitive 'put 2nd component of vector C_{j+6} into C_{j+4}'	
	C_{j+9}	contains	primitive 'put 3rd component of vector C_{j+6} into C_{j+5}'	
	C_{j+10}	contains	primitive 'put 4th component of vector C_{j+6} into C_{j+6}'	
	C_{j+11}	contains	primitive 'move indirect', parameters $j+4$ and $j+1$	
	C_{j+12}	contains	primitive 'move indirect', parameters $j+5$ and $j+2$	
	C_{j+13}	contains	primitive 'put power set of set in C_{j+4} into C_{j+4}'	
	C_{j+14}	contains	primitive 'put power set of set in C_{j+5} into C_{j+5}'	
	C_{j+15}	contains	primitive 'put intersection of C_{j+4} and C_{j+5} into C_{j+4}'	
	C_{j+16}	contains	primitive 'move indirect', parameters $j+1$ and $j+6$	
	C_{j+17}	contains	primitive 'put contents of C_{j+3} into C_0'.	

After the cells $C_j, \cdots, C_{j+17}$ are initialized in this way, the intersection of the power sets of the two sets contained in C_{n_2} and C_{n_3} respectively can be calculated and placed in C_{n_4} simply by executing the instruction vector

$$\langle j, n_2, n_3, n_4\rangle. \tag{4}$$

We can verify this by following the interpretation of (4), which we assume to be contained in the cell C_k. Since j lies outside the range $1 \leqq j \leqq \pi$, (4) is interpreted by retrieving the linking operator $\langle 1, j+6\rangle$ from C_j; then, according to the execution rule applicable in Case 2, we use this to form the longer linking primitive

$$\langle 1, j+6, n_2, n_3, n_4\rangle \tag{5}$$

which is then executed. This places $\langle k+1, n_2, n_3, n_4\rangle$ in C_{j+6}, and execution continues with the instruction contained in C_{j+7}. The next four instructions put $k+1, n_2, n_3, n_4$ into C_{j+3}, C_{j+4}, C_{j+5}, C_{j+6} respectively. Then the instructions contained in C_{j+11} and C_{j+12} copy the contents u and v of C_{n_2} and C_{n_3} to C_{j+4} and C_{j+5} respectively. After this, the instructions contained in C_{j+13} and C_{j+14} replace u and v in C_{j+4} and C_{j+5} by 2^u and 2^v, and the next instruction puts the desired intersection $2^u \cap 2^v$ into C_{j+4}. Then, since C_{j+6} contains n_4, the 'move indirect' instruction in C_{j+16} copies $2^u \cap 2^v$ into C_{n_4}. The final instruction of our sequence puts $k+1$ into C_0, so that computation will continue from the cell following the instruction (4) whose execution sequence we have just explained.

It is clear that the technique illustrated above can be used to extend an initially given library of primitives indefinitely. Thus, even if some useful transformation is omitted when an abstract computer model is specified, it can easily be added to the model's repertoire of operations. The semantic side of the programming problem is therefore seen to be that *of discovering interesting and widely useful families of transformations.*

Having carried our discussion of the semantic question thus far, we turn to say a few words about the syntactic side of programming. As has already been stated, the word syntax is used in connection with the notational issues which arise in programming, and may be taken to be descriptive of those processes which transform an external program text into a valid initialization of the cells C_j of an abstract machine. If our abstract computer is to be able to digest external texts, operations which make these texts available in a suitable internal form will obviously be required. This necessity is met as follows. A fixed sufficiently large vocabulary of external characters is chosen. Each character is assigned some conventional numerical code. Thus, for example, we might allow 256 characters, and use the codes

Character:	$a, b, \cdots, z,\ A,\ B,\ \cdots, Z,\ \cdots, \Omega$
Numerical Code:	$0, 1, \cdots, 25, 26, 27, \cdots, 51, \cdots, 255$

With a character set this large, each character is represented by a unique pattern of eight binary digits (bits). For example, we have the correspondences

Character:	a	b	$\cdots$	A	$\cdots$	Ω
Binary Pattern:	00000000	00000001	$\cdots$	00011010		11111111

By concatenating the codes for each of a string of characters and prefixing the whole by the binary digit '1', we obtain an integer representing the character string uniquely. Thus, for example,

$$\text{`Cab'} \sim \overbrace{100011100}^{C}\ \overbrace{00000000}^{a}\ \overbrace{00000001}^{b}$$

This trivial system can be used to furnish our abstract machine with 'input' and 'output' primitives, which we may take to act as follows:

(a) The input primitive $\langle 4\ n\rangle$ reads a character string from an external medium and puts its internal representation (an integer) into C_n.

(b) The output primitive $\langle 5\ n\rangle$ takes the contents of C_n (which must be an integer) and causes the character string which it represents to appear (printed) on a standard external output medium.

Having thus made it possible for external texts to be read into our abstract computer, we can go on to consider the more interesting problems which surround the internal processing of these texts. Programs (in their external representation) are strings of characters which when suitably decomposed describe initial data for the cells C_j; by first initializing the cells C_j in the way which this data indicates, and by then applying the execution rules stated above, we obtain an output which may be called the program's *result*.

An essential step in the conversion of an external program text T into data which may be used directly to initialize the cells C_j is analysis of the text T, which we regard as a set of notations, into its notationally significant subparts. This is a process which has been much and successfully studied by computer scientists, who have developed various useful meta-notations for the description of notational systems. The concept central to many of these meta-notations is that of a *context-free* (or Backus-Naur) *grammar.* Such a grammar consists of a set of *terminal symbols,* a set of *intermediate symbols*, and a family of *productions*. A terminal symbol represents part of a formula literally and directly; an intermediate symbol names a clause type. We will write terminal symbols simply as character strings containing no blanks, e.g. ABC, and write intermediate symbols as character strings contained within pointed braces, e.g. ⟨EXPRESSION⟩. Each production in a context-free grammar begins with an intermediate symbol (its *left-hand side*) which is immediately followed by the symbol '$\rightarrow$' (to be read as 'may be built up as'). The arrow may then be followed by any sequence of terminal and intermediate symbols (the *right-hand side* of the production). A production signifies that a clause of the type indicated by its left-hand side may be built up by combining the elements appearing on its right-hand side.

A particular one of the intermediate symbols of each context free grammar must be designated as its *fundamental* or *root* symbol; any clause of the 'root type' which this symbol designates is a *valid sentence* of the notational system or *language* defined by the grammar.

The following example, which shows a grammar for a restricted subfamily of ordinary algebraic expressions, will illustrate the general concepts which have just been introduced.

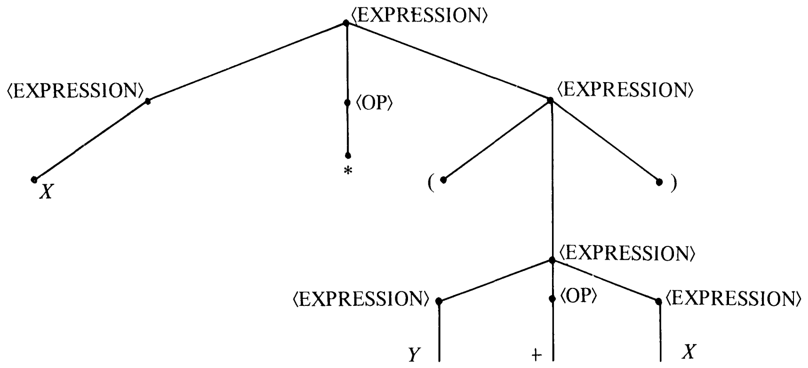
$$\begin{aligned}
&\langle\text{EXPRESSION}\rangle \rightarrow \langle\text{EXPRESSION}\rangle\ \langle\text{OPERATOR}\rangle\ \langle\text{EXPRESSION}\rangle\\
&\langle\text{EXPRESSION}\rangle \rightarrow (\langle\text{EXPRESSION}\rangle)\\
&\langle\text{EXPRESSION}\rangle \rightarrow X\\
(4)\quad &\langle\text{EXPRESSION}\rangle \rightarrow Y\\
&\quad\langle\text{OPERATOR}\rangle \rightarrow +\\
&\quad\langle\text{OPERATOR}\rangle \rightarrow -\\
&\quad\langle\text{OPERATOR}\rangle \rightarrow *\\
&\quad\langle\text{OPERATOR}\rangle \rightarrow /
\end{aligned}$$

We take ⟨EXPRESSION⟩ to be the root type of this grammar.

The grammar (4) tells us that an expression can be formed, either (first production) recursively as a sequence expression-operator-expression, or (second production) by enclosing an expression in parentheses. Moreover (third and fourth productions) the signs X and Y are (elementary) expressions, while (remaining productions) $+$, $-$, $*$, and $/$ are operators.

Each context-free grammar G defines a family of trees called the *G-well-formed syntax trees.* These are ordered trees T (in the ordinary mathematical sense), whose nodes are marked with terminal or intermediate symbols of G. A twig, i.e. a node with no descendants, must be marked with a terminal symbol. The root node of T must be marked with the root symbol of G. For T to be well formed, the following condition must be satisfied at each node ν which is not a twig: if ν is marked with the symbol S_0, and its descendants (in left-to-right order) are marked with the symbols $S_1, \cdots, S_n$, then $S_0 \rightarrow S_1 \cdots S_n$ must be a production of G.

The following figure shows a syntax tree which is well formed according to this definition, the operative grammar being (4).



Many syntactic relationships of central importance can be expressed easily in terms of context free grammars G and the notion of a G-well-formed tree. The *string covered by a syntax tree T* is the concatenation, in left-to-right order, of the symbols attached to the twigs of T. Similarly, the *substring covered by a subtree T′* is the concatenation of the symbols attached to the twigs of T'. A string S is *syntactically well formed* (according to a grammar G) if it is covered by some G-well-formed tree T; it is *unambiguous* if there exists at most one such T. The *subclauses* of a well-formed string S are the substrings covered by the subtrees of T. In general, experience shows that the notions 'grammar' and 'syntax tree' give us much of what is needed for the free treatment and variation of notational systems. Specifically, once the following operation is made available on an abstract computer, it will usually be found that notational issues can be handled without undue difficulty:

Syntax analysis primitive: *Given a grammar G and a string S, form the G-well-formed syntax tree T which covers S* (if T is not unique, form the class of all such trees).

We note that a formal definition of this primitive is easily given in set-theoretic terms. A grammar G can be represented as a set of n-tuples; for example, the grammar (4) can be represented as

$$\{\langle \text{EXPRESSION}, \text{EXPRESSION}, \text{OPERATOR}, \text{EXPRESSION}\rangle,$$
$$\langle \text{EXPRESSION}, \langle(\rangle, \text{EXPRESSION}, \langle)\rangle\rangle, \langle \text{EXPRESSION}, \langle X\rangle\rangle, \cdots\}.$$

In set-theoretic terms, a syntax tree (with operative grammar G) is a triple ⟨root, descendant_function, marking⟩, where *root* is an integer, *descendant_function* is a function f of two variables (i.e., a set of ordered triples), $f(n,j)$ being the jth descendant of the node n, and *marking* is a map which sends each node n into a symbol of G. Many variants of the syntax analysis primitive described above have been studied in depth recently; quite efficient realizations of this primitive are known.

A notational system defining a family of external program texts is commonly called a *source language*; the system defining the internal data structures into which these texts are transformed is called a *target language*. Syntax analysis builds a bridge between source and target language. By subjecting source language input to syntax analysis, and by performing various additional consistency checks and transformations on the resulting trees, we can translate a very great variety of external texts into standard target language structures which directly define useful initializations of the cells of our abstract machine. This allows us to address the computer in any notational system we like, so that the fundamental syntactic issue becomes: what notations are most convenient?

In summary: we have seen that the family of primitive operations available in an abstract computer of the type that we have considered is readily extended. The syntactic mechanisms just described make it equally easy to modify and extend the system of notations which one uses to communicate instructions to the computer. Thus, in an initial approach to the programming problem, we can hunt without constraint for widely useful, expressive transformations and notations, and can take these transformations and notations as the fundamental semantic syntactic elements of a programming language.

The semantic and syntactic choices actually made in defining any particular language will reflect its intended area of application, which may be more or less specialized. For a system intended to cover a wide but unspecialized range of applications, it is surely tempting to make use of the conventional operations and notations of set theory, which serve so well and so generally as a basis for mathematics. This has in fact been done in several recently developed programming languages (cf. [1] and [2]). To give the reader some feeling for the flavor of the language which results, we shall present a program which realizes a rather simple combinatorial process, namely Euler's solution of the Bridges of Königsberg problem. The problem, it will be recalled, is that of tracing by a connected path, all the edges of an ordered graph g, no edge to be traced more than once. Euler's construction is as follows:

(a) If the graph is disconnected or contains more than 2 'odd' nodes, i.e., nodes from which an odd number of edges emanate, then no Eulerian path exists. Otherwise start with one of the odd nodes (or, if there are no odd nodes, with any node). Draw a path p in any direction, continuing as long as possible, but never crossing the same edge twice. This process will terminate when one reaches a node all of whose incident edges have already been traversed.

(b) If the path p contains no node n upon which an untraversed edge is incident, then p must include every edge in the graph. In the contrary case, start a path from n using some untraversed edge. Continue this path q as far as possible, proceeding in the same way as in the construction of p. When q can no longer be extended, it will be seen to constitute a loop starting and ending at n. Replace p by the curve p' consisting of the portion of p preceding n, followed by q, followed by the portion of p following n.

(c) Iterate step (b) as often as possible. Eventually p will come to include every edge of the originally given graph.

We will now represent the procedure just described in a set-theoretic programming language, specifically in the SETL language developed by the present author and various collaborators (cf. [1]). In what follows, we give formal program text on the left and informal comments on the right; this should allow the reader to comprehend the partly unfamiliar notations of the formal programming language. It is assumed in the following that an (unordered) graph is given as a symmetric set *graph* of ordered pairs $\langle x,y\rangle$, the presence of $\langle x,y\rangle$ in *graph* signifying that the node x is connected to the node y by an edge. A set of ordered pairs may also be regarded as a multi-valued function; for each multi-valued function f, the notation $f\{x\}$ denotes the *set* of all values which f assumes at the point x.

Program	*Comments*
nodes = $\{e(1), e \in \text{graph}\}$;	Form the set of nodes of the given graph; i.e., the set of all initial points of edges.
odd = $\{x \in \text{nodes} \mid ((\#\,\text{graph}\{x\})$ $\underline{\text{mod}}\ 2) = 1\}$;	Form the set of all nodes which have an odd number of neighboring nodes.
if (# odds) > 2 then	If the set of odd nodes has more
print 'impossible'; *stop*;	than 2 elements, the graph has no
else if odds = *nullset* then	Eulerian path. If there are no odd nodes, choose an arbitrary element
start = $\ni$ nodes;	of the set of nodes. ('$\ni$' denotes the
else	operation of arbitrary choice) and start there. Otherwise start at an
start = $\ni$ odds;	arbitrary odd node.
end if;	
path = $\langle$start$\rangle$;	The *path* to be constructed, which will be represented by an n-tuple, begins as a 1-tuple whose only component is the starting node.
(while $\exists$ point $(k) \in$ path $\mid$ graph $\{$point$\} \neq$ *nullset*)	While there exists a point in the *path* constructed thus far to which untraversed edges are adjacent (let this be the kth point), divide *path*
part_1 = path $(1:k)$;	into a first portion, part_1, extending from its first node up to its kth,
part_2 = path $(k+1:)$;	and a second portion, part_2, extending from its $k+1$st node to the end of *path*. Now, move *point* along

Program	*Comments*
	untraversed edges to build up a loop.
(while $\exists$ next $\in$ graph{point})	Specifically, while there is any neighbor of *point* which can be reached along an untraversed edge *e*,
$\text{part}_1 = \text{part}_1 + \langle\text{next}\rangle$;	append this neighbor to the *n*-tuple part_1;
graph = graph $-$ {⟨point, next⟩, ⟨next, point⟩};	to prevent retraversal of an already traversed edge, remove the edge *e* (and its reverse) from the *graph*, and
point = next;	move *point* to the position of the neighbor which has just been appended to part_1.
end while $\exists$ next;	When this process can no longer be
path = $\text{part}_1 + \text{part}_2$;	continued, redefine *path* to be the concatenation of the extended part_1 and the former part_2. As long as there exists a point in path to which untraversed nodes are adjacent, continue the preceding process.
end while $\exists$ point;	When this process can no longer be continued, a maximal Eulerian path will have been built up.
if (# path) $\neq$ (# nodes) then *print* 'graph is disconnected'; *stop*;	If this path does not include all the nodes of *graph*, then *graph* is disconnected.
else *print* path; stop; end if;	Otherwise we have only to print the *n*-tuple which represents path.

2. **Realistic computer models. Problems of realization and optimization.** When we turn from the idealized world of abstract computer models and take into account certain restrictions which must apply to any physically constructible device, issues which we have till now ignored become visible. Present physical computers consist not of infinitely many cells C_j, but only of several hundred thousand cells. Individual cells are not capable of holding arbitrarily complex data objects, but can only hold single integers lying in some limited range (e.g. -2^{63} to 2^{63}). The primitive operations of a computer will not have indefinitely many arguments, but only 3 or 4; primitive operations are not performed instantaneously; their execution generally requires a time not much less than 10^{-7} seconds. It remains true that substantially any trans-

formation whose inputs and outputs amount only to a few hundred binary digits (bits) can be realized as a built-in primitive of an actual computer. However, not many such transformations seem to be particularly useful. After considerable experimentation, a short list of 'classical' operations, notably integer addition, multiplication, and division, bit-by-bit logical operations such as 'and', 'or', 'invert', and various bit-shift operations remain the most useful.

To make very general, abstractly flavored programming systems like those considered in §1 available in actual fact, one must map abstract machine programs onto efficient physical machine programs having the same output. For this we must represent the compound data objects (particularly sets and vectors) of the abstract machine by arrays of integers each of limited precision, which are the only data objects available on actual physical machines. Representations of compound objects which support passably efficient implementation of most important operations are by now well known. A systematic account of available representation techniques will be found in D. Knuth's excellent treatise (cf. Knuth [**3**]).

Transformation of abstract to concrete programs and data structures can be accomplished by transformations, largely local in character, which map one program into another on a section-by-section basis. However so straightforward an approach can realize only a modest level of efficiency. This observation at once confronts us with the necessity of *optimizing* when translating abstract-machine into physical-machine programs, i.e. with the necessity of developing translation techniques produce efficient physical machine programs. Three main approaches to this problem have developed. The first approach uses relatively routine local transformations during the translation process, but restricts the class of abstract programs which one attempts to translate. Then, during translation one can exploit the special restrictions imposed upon the programs which the translator will have to handle, and can thus attain high efficiency translations. For example, one may require that a maximum size be pre-stated for each data object appearing in an abstract program; a rule of this kind allows one to pre-calculate the layout of the group of physical machine words used to represent each abstract object, generally with very substantial benefit to the efficiency of the resulting translation. The difficulty with this approach is that since it restricts the dictions available to the programmer, possibly in very significant ways, it forces complicated circumlocutions upon him, thus deviating from the criteria of clarity and simplicity which define ideal programming.

A second approach involves the use of much more sophisticated translation techniques. Application of these techniques begins with an overall or 'global' analysis of the program to be translated. This analysis enables facts about the program to be deduced; these facts allow more efficient translations to be produced than would otherwise be possible. Suppose, for example, that analysis of an abstract program shows that the value stored in a certain abstract cell is always a set, and that the elements of this set are always integers which lie between 1 and 64. Then (if we are working with a physical machine whose cells can store integers at least 64 binary digits long) the set can be represented by a single integer in a physical machine cell. The bits of this integer will correspond to possible set elements; a bit will be 1 if the element is actually present in the set, otherwise it will be 0. This example should make it clear that by choosing data structures which reflect important facts about the objects appearing in an abstract program *p* one can achieve both drastic compression of the data to be stored and drastic simplification of the concrete steps needed to realize *p*. The line of endeavor which these reflections support will undoubtedly be central to much of the future development of programming languages; see Schaefer [4] for a survey of progress to date in the important field of program optimization.

It should be understood that the 'automatic' approach to translation of abstract into physical machine programs which is suggested in the preceding paragraph is more something which is developing than something fully feasible at the present time. For this reason, a third and far less formal approach, namely manual translation, actually typifies current programming technique. More often than not in present technique, the abstract program to be translated exists only as an imprecisely formulated overall plan in the programmer's head. Informally combining a more or less accurate understanding of the special properties of the algorithm which he means to program with a knowledge of salient special properties of a particular physical machine, the programmer writes a detailed program in a restricted language close enough to the machine to permit high efficiency to be attained. The great objection to this most commonly used approach is that it is tedious, lengthy, imperspicuous, and highly error-prone.

3. **An overall view of current programming research.** We have carried our technical discussion far enough to be able to pause and comment more broadly on current programming research. This comprises various related activities: including, as already noted, both the search for useful and powerfully expressive families of transformations, and the search for particularly expressive systems of notation. A related endeavor is that of devising semantic frameworks within which individual transformations can be combined readily and flexibly. This involves the development not only of abstract machine models, but also of other systems such as the λ-calculus and the Curry combinator calculus, within which transformations appear as manipulable elements. A related goal is to discover efficient procedures which realize important abstract transformations, or to find variants of these transformations which can be realized in particularly efficient ways. How can syntax analysis or matrix multiplication be performed most rapidly? How can large masses of data best be organized into tables which facilitate rapid search for particular items? These last questions, and many others like them, are of great importance in programming research. The attempt to answer such questions brings the pragmatic study of programming technique into active contact with that more theoretical branch of computer science known as *formal analysis of algorithms*. In formal algorithm analysis, one attempts, in the first place, to develop rigorous formulae (often asymptotic) for the average and worst case behavior of particular algorithms. One also attempts to establish *a priori* lower bounds for the space and time required to realize important abstractly defined transformations. Where this succeeds, one can compare the analyzed behavior of particular algorithms with firm theoretical ultimate limits, and thus can assign an objective 'figure of merit' to algorithms.

Another important activity is the ongoing attempt to program significant procedures which are complex enough to strain existing techniques. Pragmatic attempts of this sort force the growth of new techniques. There exist several continuing problems for which ever-new programming challenges may be expected to arise. One such problem is that of representing (automating) the internal procedures of large social organizations. This is the ordinary stuff of "business data processing"; the procedures to be programmed are complex because they reflect the complex variety of social existence. Another pregnant problem is that of duplicating the abilities of the human mind, at least in part, whether these be "higher" abilities such as theorem proving or "instinctive" functions such as the ability to recognize a two-dimensional line drawing as being the projected representation of a particular three-dimensional object. Attempts to imitate mental function have forced interesting complex program structures on the programmer, and have motivated many sparkling developments in programming technique. At a less ambitious but perhaps more immediately practical level lies the problem of automating certain types of sophisticated but relatively routine mental activity, such as the activity of algebraic calculation and simplification, or the activity of manual translation of programs between different notational systems. Finally, we may mention the problem of program optimization alluded to above, i.e., the problem of developing algorithms capable of analyzing abstract programs deeply enough to discover facts about them which permit their translation into highly efficient programs for a physical machine. Related to this last is the problem of constructing algorithms which can assist in the process, at present barely manageable, of proving programs correct.

These are all problems which we expect to be long-term sources of inspiration for programming research. A part of present research concerns itself directly with these matters; another part is focused upon issues of an origin more plainly internal to programming itself. One such area of current activity is the study of processes proceeding in parallel. We can make parallelism possible in our abstract machine model by allowing the instruction location counter contained in C_0 (see §1, p. 2) to be a *set* S of integers (rather than a single integer as before). Then, on each cycle of execution of the abstract machine (as thus generalized) we allow some subset of S to be chosen, and allow the contents of the cells C_j addressed by the integers in S all to be fetched at once and all executed. This clearly creates a situation in which one will have to investigate more powerful principles of process organization than those which suffice when only one single sequence of operations at a time is being executed.

It may be remarked that in many practical situations some degree of parallelism must be employed if acceptable performance is to be attained. For example, most current computer systems are able to perform internal calculations in parallel with the reading of new input (perhaps from several sources), with the transmission of output information (perhaps to several destinations all at the same time), and with the motion of data to and from storage devices such as magnetic discs and tapes. Indeed, we may note that the rapid progress of computer technology is making it possible to employ parallelism on an ever-larger scale. This fact lends interest even to toally parallel computer models, one of which we may define as follows: A bit-parallel computer consists of infinitely many cells $\cdots B_{-1}, B_0, B_1, B_2, \cdots$, each of which stores a single binary digit. With each cell B_j is associated some Boolean function F_j of some finite collection of all these binary digits:

$$B_j \sim F_j(B_{i_1}, \cdots, B_{i_{k_j}}).$$

Among the functions F_j, only finitely many different Boolean functions appear. In the initial condition of such a model, all but a finite number of the bits B_j are 0. On each cycle of operation, all the B_j are simultaneously transformed, each B_j becoming $F_j(B_{i_1}, \cdots, B_{i_{k_j}})$. When the cell B_0, which serves as a 'stop bit', changes from 0 to 1, execution halts.

This model comes reasonably close to a type of computer which it might be feasible to build within the next few years. In presently visible technologies, a cycle of transformation could be accomplished in a time lying between 10^{-8} and 10^{-7} seconds.

Another important area of current research is the study of techniques whereby programs known to be correct can be allowed to coexist with, and even to control and make use of, other programs which might contain errors or be maliciously designed to act perversely. This is a central issue in the design of operating systems. In studying it, one makes contact with the important question of response to and recovery from error.

4. **Another example of current programming research: Nondeterministic programming.** Still another active area of current work is the search for more general abstract machine models, i.e. of new

semantic frameworks which may be particularly useful for the description of certain classes of problems. An especially felicitous generalization of this kind is due to R. Floyd [5]; functioning systems which realize Floyd's suggestion have now been made available by groups at MIT and Stanford. Floyd's idea, which makes it easier to write programs which explore complex logical 'mazes', is to allow what may be called a 'predictive oracle' to be part of an abstract computer model. Such an oracle can be consulted by the programmer *whenever he must make a decision which will lead his program either to failure or to a successful outcome some* (possibly very large) *number of steps after the point at which the decision is taken.* To formalize the notion of a predictive oracle, we have only to introduce two additional primitives into the abstract computer model described in §1. These primitives are as follows:

(A) A primitive *fail*, without parameters. This, if executed, tells the computer that it has failed. (However, since a predictive oracle will be used to avoid precisely this situation, we intend that the *fail* primitive should never be executed!)

(B) A primitive *goodvalue*, with one parameter n. This is the predictive oracle itself. When executed, it sets C_n either to the value 0 or to the value 1. The value chosen reflects the *future* course of the program being executed. By definition, we choose 0 if this choice, considered in the light of all the subsequent tests and calculations which the program will make, prevents the primitive *fail* from ever being executed. If 0 is inappropriate, but 1 is appropriate, *goodvalue* sets C_n to 1. If neither 0 nor 1 is appropriate, i.e., if the oracle itself is baffled, then execution will halt and the message "the problem is logically impossible" will appear. Of course, since we take it that our oracle's prevision is perfect, this means that there is no way that the given program can avoid failure. It should be obvious that the *goodvalue* oracle will announce this impossibility the first time it is consulted.

The reader will readily agree that if an oracle of the type we have just described is available, the writing of many algorithms will be simplified. We shall now show how the two primitives *fail* and *goodvalue* which describe the oracle can be implemented on the straightforwardly deterministic abstract machine described in §1. This implies that any program in which the above 'oracular' primitives are used can be converted in a routine way to a program in which they are not used. We proceed as follows. Reserve four cells C_j of the abstract machine; for emphasis, and to stress the fact that these cells are used for no other purpose, call them D_1, D_2, D_3, D_4. Initially, D_1 is to contain the smallest index j for which C_j is not zero (we call this index α), D_2 the largest index j for which C_j is not zero (we call this index β), and D_3 is to contain the null tuple, i.e. a tuple of length 0. To execute the *goodvalue* operation (with parameter n), we place the value 0 in C_n, and append

$$\langle \alpha, \beta, n \ \text{content}(C_\alpha), \text{content}(C_{\alpha+1}), \cdots, \text{content}(C_\beta) \rangle$$

as a final component to the tuple present in D_3. Following this, the integer in C_0 must of course be incremented by 1. To execute the *fail* primitive, we fetch α from D_1, β from D_2, and enter 0 into each of the cells $C_\alpha, \cdots, C_\beta$. Then we detach the last component v from the tuple t stored in D_3; v will be a vector of the form

$$\langle \alpha', \beta', n', c_{\alpha'}, c_{\alpha'+1}, \cdots, c_{\beta'} \rangle.$$

Finally, we place α' in D_1, β' in D_2, $c_{\alpha'}$ in $C_{\alpha'}, \cdots, c_{\beta'}$ in $C_{\beta'}$, and put the value 1 into $C_{n'}$.

The heuristic significance of this quite straightforward formal procedure may be explained as follows. When our deterministic computer is called upon to play oracle by supplying a 0 or a 1, it *guesses* that a 0 is the value required. The cell D_3 is used to contain a history of the computation states preceding each of these guesses. The jth component of D_3 records the situation at the time the jth guess was made. If a series of guesses leads the program to failure, the last preceding guess made is simply reversed, i.e. the situation is restored to exactly what it was immediately before this guess, and the opposite guess is made. Our quite deterministic computation process imitates an infallible oracle simply by forgetting completely about all its mistakes! Of course, this does not do away with the computational effort which these wrong guesses make necessary. However, it does allow the programmer to pretend in his dictions that his program can infallibly guess the right step to take in exploring a logical maze.

As an example of the type of program made far simpler by the 'oracular' dictions which the preceding considerations legitimize, we consider a familiar puzzle, the so-called *problem of eight queens*. This is the problem of placing eight queens on a chessboard in a pattern in which no two queens attack each other. For a person (or a program) who never makes mistakes, the problem is trivial. Namely, it is clear that one queen must be placed in each column of the chessboard. Therefore merely place the first queen in the correct row of the first column, the second queen in the correct row of the second column, and so forth until eight queens have been placed on the board. To tell which row is correct, simply consult the oracle. The following code, which is written in an extension of the SETL language already used once, formalizes exactly this procedure. Note that in the code the *goodvalue* primitive is represented by a special nondeterministic boolean quantity *ok*, which when oracularly evaluated gives *true* or *false*, whichever will prevent the statement *fail* from having to be executed subsequently. As before, we give a formal program on the left and a copious set of informal comments on the right.

```
columns = $\{n, 1 \leqq n \leqq 8\}$;                 There are eight columns on the board,
rows = columns;                                 and equally many rows.
positions = nulltuple;                          We will use an $n$-tuple to represent
                                                the sequence of positions in which
                                                successive queens are placed.
                                                Initially, no queens have been placed,
                                                so this is a tuple of length 0.
$(1 \leqq \forall k \leqq 8)$                               Then, for each column in left-to-
                                                right sequence,
possibilities =                                 form the set of rows in which
                                                it is still possible to place a queen,
                                                namely
rows                                            the set of all rows, minus those
                                                rows which are attacked by earlier
  $- \{\text{positions}(j), 1 \leqq j < k\}$              queens along a horizontal, minus
                                                those rows which are attacked by
  $- \{\text{positions}(j) + k - j, 1 \leqq j < k\}$      earlier queens along a rising diagonal,
                                                minus those rows which are attacked
  $- \{\text{positions}(j) + j - k, 1 \leqq j < k\}$;     along a falling diagonal.
  if $\exists\, r \in$ possibilities | ok then            If the oracle finds that one of these
                                                possibilities is correct for placing
    positions(k) = r;                           an additional queen, place it there;
  else                                          otherwise
     fail;                                        encounter failure.
  end if;
end $\forall k$;                                      When this has been done eight times,
print positions;          print the sequence of positions in
stop;                     which the eight queens have been
                          placed.
```

The preceding program shows how great an economy of expression can be attained by adapting one's semantic framework to the class of problems one intends to treat.

BIBLIOGRAPHY

1. J. T. Schwartz, *On programming. An interim report on the SETL project.*
Installment 1. *Generalities* (1973).
Installment 2. *The SETL language and examples of its use* (1973) 502 pp.
Installment 3. *Extension and optimization* (in prep.)
Computer Science Department, Courant Inst. Math. Sci., New York Univ., New York, N. Y.

2. J. B. Morris, *A comparison of MADCAP and SETL,* Los Alamos Scientific Laboratory, Univ. of California, Los Alamos, New Mexico, (1973).

3. Donald Knuth, *The art of computer programming,* Vol. I. *Fundamental algorithms,* (1968); Vol. II. *Seminumerical algorithms,* (1969); Vol. III. *Sorting and searching,* (1973), Addison-Wesley Publishing Co., Reading, Mass.

4. Marvin Schaefer, *A mathematical theory of global program optimization,* Prentice-Hall Publishers, Englewood Cliffs, N. J., (1973).

5. Robert Floyd, *Nondeterministic algorithms,* J. Assoc. Comput. Mach. 14 (1967), 636-644.

DEPARTMENT OF COMPUTER SCIENCE, COURANT INSTITUTE OF MATHEMATICAL SCIENCES, 251 MERCER ST., NEW YORK, NEW YORK 10012