References

1. Avi-Itzhak, B., and Naor, P. Some queueing problems with the service station subject to breakdowns. *Oper. Res. 11*, 3 (1963), 303–320.

2. Basket, F., Chandy, K.M., Muntz, R.R., and Palacois, F.G. Open, closed and mixed networks with different classes of customers. *J. ACM 22*, 2 (April 1975), 248–260.

3. Chamberlin, D.D., Fuller, S.H., and Lin, L.Y. A page allocation strategy for multiprogramming systems with virtual memory. Proc. 4th Symp. Operating Systems Principles, 1973, pp. 66–72.

4. Coffman, E.G., and Mitrani, I. Selecting a scheduling rule that meets pre-specified response time demands. Proc. 5th Symp. Operating Systems Principles, 1975, pp. 187–191.

5. Courtois, P. Decomposability, instability and saturation in multiprogramming systems. *Comm. ACM 18*, 7 (July 1975), 371–377.

6. Denning, P.J. A note on paging drum efficiency. *Computing Surveys 4*, 1 (March 1972), 1–4.

7. Ghanem, M.Z. Study of memory partitioning for multiprogramming systems with virtual memory. *IBM J. Res. Devel. 19*, 5 (1975), 451–457.

8. Hine, J.H., Mitrani, I., and Tsur, S. The use of memory allocation to control response times in paged computer systems with different job classes. 2nd Int. Workshop on Modelling and Performance Evaluation of Computer Systems, Stressa, Oct. 1976, pp. 201–216.

9. Lynch, W.L. Do disk arms move? Tech. Rep. 1118, Jennings Comptr. Centre, Case Western Reserve U., Cleveland, Ohio, 1972.

10. Mitrani, I., and Avi-Itzhak, B. A many server queue with service interruptions. *Oper. Res. 16*, 3 (1968), 628–638.

11. Mitrani, I., and Hine, J.H. Complete parameterised families of job scheduling strategies. Tech. Rep. 81, Comptg. Lab., U. of Newcastle upon Tyne, England, 1975.

12. Reiser, M., and Kobayashi, H. Queueing networks with multiple closed chains: Theory and computational algorithms. *IBM J. Res. Devel. 19*, 3 (May 1975), 283–294.

13. Wilhelm, N.C. An anomaly in disk scheduling: A comparison of FCFS and SSTF seek scheduling using an empirical model for disk accesses. *Comm. ACM 19*, 1 (Jan. 1976), 13–17.

Programming Languages

J.J. Horning Editor

Algorithm = Logic + Control

Robert Kowalski
Imperial College, London

An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency. The efficiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the program text.

Key Words and Phrases: control language, logic programming, nonprocedural language, programming methodology, program specification, relational data structures

CR Categories: 3.64, 4.20, 4.30, 5.21, 5.24

Introduction

Predicate logic is a high level, human-oriented language for describing problems and problem-solving methods to computers. In this paper, we are concerned not with the use of predicate logic as a programming language in its own right, but with its use as a tool for the analysis of algorithms. Our main aim will be to study ways in which logical analysis can contribute to improving the structure and efficiency of algorithms.

This research was supported by a grant from the Science Research Council.

Author's address: R.A. Kowalski, Dept. of Computing and Control, Imperial College of Science and Technology, 180 Queens Gate, London SW7 2BZ, England.

The notion that computation = controlled deduction was first proposed by Pay Hayes [19] and more recently by Bibel [2] and Vaughn–Pratt [31]. A similar thesis that database systems should be regarded as consisting of a relational component, which defines the logic of the data, and a control component, which stores and retrieves it, has been successfully argued by Codd [10]. Hewitt's argument [20] for the programming language PLANNER, though generally regarded as an argument against logic, can also be regarded as an argument for the thesis that algorithms be regarded as consisting of both logic and control components. In this paper we shall explore some of the useful consequences of that thesis.

We represent the analysis of an algorithm A into a *logic component* L, which defines the logic of the algorithm, and a *control component* C, which specifies the manner in which the definitions are used, symbolically by the equation

$$A = L + C.$$

Algorithms for computing factorials are a simple example. The definition of factorial constitutes the logic component of the algorithms:

1 is the factorial of 0;
u is the factorial of $x + 1$ *if* v is the factorial of x *and* u is v times $x + 1$.

The definition can be used *bottom-up* to derive a sequence of assertions about factorial or it can be used *top-down* to reduce the problem of computing the factorial of $x + 1$ to the subproblems of computing the factorial of x and multiplying the result by $x + 1$. Different ways of using the same definition give rise to different algorithms. Bottom-up use of the definition behaves like iteration. Top-down use behaves like recursive evaluation.

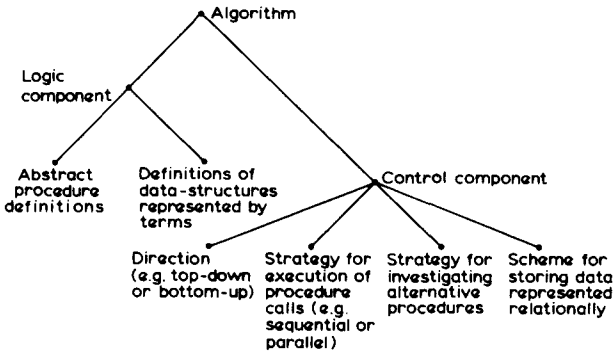
The manner in which the logic component is used to solve problems constitutes the control component. As a first approximation, we restrict the control component C to general-purpose problem-solving strategies which do not affect the meaning of the algorithm as it is determined by the logic component L. Thus different algorithms A_1 and A_2, obtained by applying different methods of control C_1 and C_2 to the same logic definitions L, are equivalent in the sense that they solve the same problems with the same results. Symbolically, if $A_1 = L + C_1$ and $A_2 = L + C_2$, then A_1 and A_2 are equivalent. The relationship of equivalence between algorithms, because they have the same logic, is the basis for using logical analysis to improve the efficiency of an algorithm by retaining its logic but improving the way it is used. In particular, replacing bottom-up by top-down control often (but not always) improves efficiency, whereas replacing top-down sequential solution of subproblems by top-down parallel solution seems never to decrease efficiency.

Both the logic and the control components of an algorithm affect efficiency. The logic component expresses the knowledge which can be used in solving problems and the control component determines the way in which that knowledge can be used. The distinction between logic and control is not wholly unambiguous. The same algorithm A can often be analyzed in different ways.

$A = L_1 + C_1.$
$A = L_2 + C_2.$

One analysis might include in the logic component what another analysis includes in the control component. In general, we prefer an analysis which places the greatest burden for achieving efficiency on the control component. Such an analysis has two advantages: (1) the logic component can afford to be a clearer and more obviously correct statement of the problem and the knowledge which can be used in its solution and (2) the control component assumes greater responsibility for the efficiency of the algorithm, which consequently can be more readily improved by upgrading the efficiency of the control.

It is the intention that this paper should be self-contained. The first part, accordingly, introduces the clausal form of predicate logic and defines the top-down and bottom-up interpretations of Horn clauses. The body of the paper investigates the following decomposition of algorithms into their various components.



We study the affect of altering each of the above components of an algorithm. The final section of the paper introduces a graphical notation for expressing, more formally than in the rest of the paper, certain kinds of control information. Much of the material in this paper has been abstracted from lecture notes [23] prepared for the advanced summer school on foundations of computing held at the Mathematical Centre in Amsterdam in May 1974.

Notation

We use the clausal form of predicate logic. Simple assertions are expressed by clauses:

Father (Zeus, Ares) ←
Mother (Hera, Ares) ←
Father (Ares, Harmonia) ←
Mother (Semele, Dionisius) ←
Father (Zeus, Dionisius) ←
etc.

Here Father (x, y) states that x is the father of y and Mother (x, y) states that x is the mother of y.

Clauses can also express general conditional propositions:

Female $(x) \leftarrow$ Mother (x, y)
Male $(x) \leftarrow$ Father (x, y)
Parent $(x, y) \leftarrow$ Mother (x, y)
Parent $(x, y) \leftarrow$ Father (x, y).

These state that

x is female *if* x is mother of y,
x is male *if* x is father of y,
x is parent of y *if* x is mother of y, and
x is parent of y *if* x is father of y.

The arrow $\leftarrow$ is the *logical connective* "if"; "x" and "y" are *variables* representing any individuals; "Zeus," "Ares," etc. are *constant symbols* representing particular individuals; "Father," "Mother," "Female," etc. are *predicate symbols* representing relations among individuals. Variables in different clauses are distinct even if they have the same names.

A clause can have several joint conditions or several alternative conclusions. Thus

Grandparent $(x, y) \leftarrow$ Parent (x, z), Parent (z, y)
Male (x), Female $(x) \leftarrow$ Parent (x, y)
Ancestor $(x, y) \leftarrow$ Parent (x, y)
Ancestor $(x, y) \leftarrow$ Ancestor (x, z), Ancestor (z, y)

where x, y, and z are variables, state that for all x, y, and z

x is grandparent of y *if* x is parent of z *and* z is parent of y;
x is male *or* x is female *if* x is parent of y;
x is ancestor of y *if* x is parent of y; and
x is ancestor of y *if* x is ancestor of z *and* z is ancestor of y.

Problems to be solved are represented by clauses which are denials. The clauses

$\leftarrow$ Grandparent (Zeus, Harmonia)
$\leftarrow$ Ancestor (Zeus, x)
$\leftarrow$ Male (x), Ancestor $(x$, Dionisius)

where x is a variable state that

Zeus is not grandparent of Harmonia,
for no x is Zeus ancestor of x, and
for no x is x male *and* is x an ancestor of Dionisius.

A typical problem-solver (or theorem-prover) reacts to a denial by using other clauses to try to refute the denial. If the denial contains variables, then it is possible to extract from the refutation the values of the variables which account for the refutation and represent a solution of the problem to be solved. In this example, different refutations of the second denial find different x of which Zeus is the ancestor:

x = Ares, x = Harmonia, x = Dionisius.

More generally, we define clauses and their interpretation as follows. A *clause* is an expression of the form

$$B_1, \ldots, B_m \leftarrow A_1, \ldots, A_n \qquad m, n \geq 0,$$

where $B_1, \ldots, B_m, A_1, \ldots, A_n$ are atoms. The atoms $A_1, \ldots, A_n$ are *conditions* of the clause and the atoms $B_1, \ldots, B_m$ are alternative *conclusions* of the clause. If the clause contains the variables $x_1, \ldots, x_k$ then interpret it as stating that

for all $x_1, \ldots, x_k$
B_1 or ... or B_m if A_1 and ... and A_n.

If $n = 0$, then interpret it as stating unconditionally that

for all $x_1, \ldots, x_k$
B_1 or ... or B_m.

If $m = 0$, then interpret it as stating that

for no $x_1, \ldots, x_k$
A_1 and ... and A_n.

If $m = n = 0$, then interpret the clause as a sentence which is always false.

An *atom* (or *atomic formula*) is an expression of the form

$$P(t_1, \ldots, t_n)$$

where P is an n-place predicate symbol and $t_1, \ldots, t_n$ are terms. Interpret the atom as asserting that the relation called P holds among the individuals called $t_1, \ldots, t_n$.

A *term* is a variable, a constant symbol, or an expression of the form

$$f(t_1, \ldots, t_n)$$

where f is an n-place function symbol and $t_1, \ldots, t_n$ are terms.

The sets of *predicate symbols*, *function symbols*, *constant symbols*, and *variables* are any mutually disjoint sets. (By convention, we reserve the lower case letters u, v, w, x, y, z, with or without adornments, for variables. The type of other kinds of symbols is identified by the position they occupy in clauses.)

Clausal form has the same expressive power as the standard formulation of predicate logic. All variables $x_1, \ldots, x_k$ which occur in a clause C are implicitly governed by universal quantifiers $\forall x_1, \ldots, \forall x_k$ (for all x_1 and ... and for all x_k). Thus C is an abbreviation for

$$\forall x_1 \ldots \forall x_k\, C.$$

The existential quantifier $\exists x$ (there exists an x) is avoided by using constant symbols or function symbols to name individuals. For example, the clauses

Father (dad (x), $x) \leftarrow$ Human (x)
Mother (mum (x), $x) \leftarrow$ Human (x)

state that for all humans x, there exists an individual, called dad (x), who is father of x, and there exists an individual, called mum (x), who is mother of x.

Although the clausal form has the same power as the standard form, it is not always as natural or as easy to use. The definition of subset is an example: "x is a subset of y if for all z, z belongs to y if z belongs to x." The definition in the standard form of logic

$$x \subseteq y \leftarrow \forall z\, [z \in y \leftarrow z \in x]$$

is a direct translation of the English. The clausal form of

the definition can be systematically derived from the standard form. It can take considerable effort, however, to recognize the notion of subset in the resulting pair of clauses:

$x \subseteq y$, *arb* $(x, y) \in x \leftarrow$
$x \subseteq y \leftarrow$ *arb* $(x, y) \in y$.

(Here we have used infix notation for predicate symbols, writing xPy instead of $P(x, y)$.)

In this paper, we shall avoid the awkwardness of the clausal definition of subset by concentrating attention on clauses which contain at the most one conclusion. Such clauses, called *Horn clauses*, can be further classified into four kinds:

assertions (of the form)	$B \leftarrow$
procedure declarations (of the form)	$B \leftarrow A_1, \ldots, A_n$
denials	$\leftarrow A_1, \ldots, A_n$
and *contradiction*	$\leftarrow$

Assertions can be regarded as the special case of procedure declarations where $n = 0$.

The Horn clause subset of logic resembles conventional programming languages more closely than either the full clausal or standard forms of logic. For example, the notion of subset can be defined recursively by means of Horn clauses:

$x \subseteq y \leftarrow$ Empty (x)
$x \subseteq y \leftarrow$ Split (x, z, x') $z \in y$, $x' \subseteq y$.

Here it is intended that the relationship Empty (x) holds when x is empty, and Split (x, z, x') holds when x consists of element z and subset x'. Horn clauses used in this way, to define relations recursively, are related to Herbrand-Gödel recursion equations as described by Kleene [22], elaborated by McCarthy [28], employed for program transformation by Burstall and Darlington [13], and augmented with control annotations by Schwarz [34].

Top-Down and Bottom-Up Interpretations of Horn Clauses

A typical Horn clause problem has the form of

(1) a set of clauses which defines a problem domain and
(2) a theorem which consists of (a) hypotheses represented by assertions $A_1 \leftarrow, \ldots, A_n \leftarrow$ and (b) a conclusion which is negated and represented by a denial $\leftarrow B_1, \ldots, B_m$.

In top-down problem-solving, we reason backwards from the conclusion, repeatedly reducing goals to subgoals until eventually all subgoals are solved directly by the original assertions. In bottom-up problem-solving, we reason forwards from the hypotheses, repeatedly deriving new assertions from old ones until eventually the original goal is solved directly by derived assertions.

The problem of showing that Zeus is a grandparent of Harmonia can be solved either top-down or bottom-up. Reasoning bottom-up, we start with the assertions

Father (Zeus, Ares) ←
Father (Ares, Harmonia) ←

and use the clause Parent $(x, y) \leftarrow$ Father (x, y) to derive new assertions

Parent (Zeus, Ares) ←
Parent (Ares, Harmonia) ←

Continuing bottom-up we derive, from the definition of grandparent, the new assertion

Grandparent (Zeus, Harmonia) ←

which matches the original goal.

Reasoning top-down, we start with the original goal of showing that Zeus is a grandparent of Harmonia

← Grandparent (Zeus, Harmonia)

and use the definition of grandparent to derive two new subgoals

← Parent (Zeus, z), Parent (z, Harmonia)

by denying that any z is both a child of Zeus and a parent of Harmonia. Continuing top-down and considering both subgoals (either one at a time or both simultaneously), we use the clause Parent $(x, y) \leftarrow$ Father (x, y) to replace the subproblem Parent (Zeus, z) by Father (Zeus, z) and the subproblem Parent (z, Harmonia) by Father (z, Harmonia). The newly derived subproblems are solved compatibly by assertions which determine "Ares" as the desired value of z.

In both the top-down and bottom-up solutions of the grandparent problem, we have mentioned the derivation of only those clauses which directly contribute to the eventual solution. In addition to the derivation of relevant clauses, it is often unavoidable, during the course of searching for a solution, to derive assertions or subgoals which do not contribute to the solution. For example, in the bottom-up search for a solution to the grandparent problem, it is possible to derive the irrelevant assertions

Parent (Hera, Ares) ←
Male (Zeus) ←

In the top-down search it is possible to replace the subproblem Parent (Zeus, z) by the irrelevant and unsolvable subproblem Mother (Zeus, z).

There are both proof procedures which understand logic top-down (e.g. model elimination [17], SL-resolution [20], and interconnectivity graphs [35]) as well as ones which understand logic bottom-up (notably hyper-resolution [35]). These proof procedures operate with the clausal form of predicate logic and deal with both Horn clauses and non-Horn clauses. Among clausal proof procedures, the connection graph procedure [25] is able to mix top-down and bottom-up reasoning. Among non-clausal proof procedures, Gentzen systems [1] and Bledsoe's related natural deduction systems [5] provide facilities for mixing top-down and bottom-up reasoning.

The terminology "top-down" and "bottom-up" applied to proof procedures derives from our investigation of the parsing problem formulated in predicate logic [23, 25]. Given a grammar formulated in clausal form, top-down reasoning behaves as a top-down parsing algorithm and bottom-up reasoning behaves as a bottom-up algorithm. David Warren (unpublished) has shown how to define a general proof procedure for Horn clauses, which when applied to the parsing problem, behaves like the Earley parsing algorithm [16].

The Procedural Interpretation of Horn Clauses

The procedural interpretation is the top-down interpretation. A clause of the form

$B \leftarrow A_1, \ldots, A_n \qquad n \geq 0$

is interpreted as a *procedure*. The *name* of the procedure is the conclusion B which identifies the form of the problems which the procedure can solve. The *body* of the procedure is the set of *procedure calls* A_i. A clause of the form

$\leftarrow B_1, \ldots, B_m \qquad m \geq 0$

consisting entirely of procedure calls (or problems to be solved) behaves as a *goal statement*. A procedure

$B \leftarrow A_1, \ldots, A_n$

is *invoked* by a procedure call B_i in the goal statement:

(1) By matching the call B_i with the name B of the procedure;
(2) By replacing the call B_i with the body of the procedure obtaining the new goal statement
$\leftarrow B_1, \ldots, B_{i-1}, A_1, \ldots, A_n, B_{i+1}, \ldots, B_m$
and;
(3) By applying the matching substitution θ
$\leftarrow (B_1, \ldots, B_{i-1}, A_1, \ldots, A_n, B_{i+1}, \ldots, B_m)\,\theta$.

(The *matching substitution* θ replaces variables by terms in a manner which makes B and B_i identical: $B\theta = B_i\theta$.) The part of the substitution θ which affects variables in the original procedure calls $B_1, \ldots, B_m$ transmits *output*. The part which affects variables in the new procedure calls $A_1, \ldots, A_n$ transmits *input*.

For example, invoking the grandparent procedure by the procedure call in

← Grandparent (Zeus, Harmonia)

derives the new goal statement

← Parent (Zeus, z), Parent (z, Harmonia).

The matching substitution

x = Zeus
y = Harmonia

transmits input only. Invoking the assertional procedure

Father (Zeus, Ares) ←

by the first procedure call in the goal statement

← Father (Zeus, z), Parent (z, Harmonia)

derives the new goal statement

← Parent (Ares, Harmonia).

The matching substitution

z = Ares

transmits output only. In general, however, a single procedure invocation may transmit both input and output.

The top-down interpretation of Horn clauses differs in several important respects from procedure invocation in conventional programming languages:

(1) The body of a procedure is a *set* rather than a *sequence* of procedure calls. This means that procedure calls can be executed in any sequence or in parallel.
(2) More than one procedure can have a name which matches a given procedure call. Finding the "right" procedure is a search problem which can be solved by trying the different procedures in sequence, in parallel, or in other more sophisticated ways.
(3) The input-output arguments of a procedure are not fixed but depend upon the procedure call. A procedure which *tests* that a relationship holds among given individuals can also be used to *find* individuals for which the relationship holds.

The Relationship Between Logic and Control

In the preceding sections we considered alternative top-down and bottom-up control strategies for a fixed predicate logic representation of a problem-domain. Different control strategies for the same logical representation generate different behaviors. However, information about a problem-domain can be represented in logic in different ways. Alternative representations can have a more significant effect on the efficiency of an algorithm than alternative control strategies for the same representation.

Consider the problem of sorting a list. In one representation, we have the definition

sorting x gives $y \leftarrow y$ is a permutation of x, y is ordered.

(Here we have used distributed infix notation for predicate symbols, writing $P_1x_1P_2x_2 \ldots P_nx_nP_{n+1}$ instead of $P(x_1, \ldots, x_n)$ where the P_i (possibly empty) are *parts* of P.) As described in [24], different control strategies applied to the definition generate different behaviors. None of these behaviors, however, is efficient enough to qualify as a reasonable sorting algorithm. By contrast, even the simplest top-down, sequential control behaves efficiently with the logic of quicksort [17]:

sorting x gives $y \leftarrow x$ is empty, y is empty
sorting x gives $y \leftarrow$ first element of x is x_1, rest of x is x_2,
partitioning x_2 by x_1 gives u and v,
sorting u gives u',
sorting v gives v',
appending w to u' gives y,
first element of w is x_1,
rest of w is v'.

Like the predicates "permutation" and "ordered" before, the predicates "empty," "first," "rest," "partitioning," and "appending" can be defined independently from the definition of "sorting." (Partitioning x_2 by x_1 is intended to give the list u of the elements of x_2 which are smaller than or equal to x_1 and the list v of the elements of x_2 which are greater than x_1.)

Our thesis is that, in the systematic development of well-structured programs by successive refinement, the logic component needs to be specified before the control component. The logic component defines the problem-domain-specific part of an algorithm. It not only determines the meaning of the algorithm but also influences the way the algorithm behaves. The control component specifies the problem-solving strategy. It affects the behavior of the algorithm without affecting its meaning. Thus the efficiency of an algorithm can be improved by two very different approaches, either by improving the logic component or by leaving the logic component unchanged and improving the control over its use.

Bibel [3, 4], Clark, Darlington, Sickel [7, 8, 9], and Hogger [21] have developed strategies for improving algorithms by ignoring the control component and using deduction to derive a new logic component. Their methods are similar to the ones used for transforming formal grammars and programs expressed as recursion equations [13].

In a logic programming system, specification of the control component is subordinate to specification of the logic component. The control component can either be expressed by the programmer in a separate control-specifying language, or it can be determined by the system itself. When logic is used, as in the relational calculus, for example [11], to specify queries for a database, the control component is determined entirely by the system. In general, the higher the level of the programming language and the less advanced the level of the programmer, the more the system needs to assume responsibility for efficiency and to exercise control over the use of the information which it is given.

The provision of a separate control-specifying language is more likely to appeal to the more advanced programmer. Greater efficiency can often be achieved when the programmer is able to communicate control information to the computer. Such information might be, for example, that in the relation $F(x, y)$ the value of y is a function of the argument x. This could be used by a backtracking interpreter to avoid looking for another solution to the first goal in the goal statement

$\leftarrow F(A, y), G(y)$

when the second goal fails. Another example of such information might be that one procedure

$P \leftarrow Q$

is more likely to solve P than another procedure

$P \leftarrow R$.

This kind of information is common in fault diagnosis where, on the basis of past experience, it might be known that symptom P is more likely to have been caused by Q than R.

Notice, in both of these examples, that the control information is problem-specific. However, if the control information is correct and the interpreter is correctly implemented, then the control information should not affect the meaning of the algorithm as determined by its logic component.

Data Structures

In a well-structured program it is desirable to separate data structures from the procedures which interrogate and manipulate them. Such separation means that the representation of data structures can be altered without altering the higher level procedures. Alteration of data structures is a way of improving algorithms by replacing an inefficient data structure by a more effective one. In a large, complex program, the demands for information made on the data structures are often fully determined only in the final stages of the program design. By separating data structures from procedures, the higher levels of the program can be written before the data structures have been finally decided.

The data structures of a program are already included in the logic component. Lists for example can be represented by terms, where

nil	names for the empty list and
cons (x, y)	names the list with first element x and rest which is another list y.

Thus the term

cons (2, *cons* (1, *cons* (3, *nil*)))

names the three-element list consisting of the individuals 2, 1, 3 in that order.

The data-structure-free definition of quicksort in the preceding section interacts with the data structure for lists via the definitions

nil is empty $\leftarrow$
first element of *cons* (x, y) is $x \leftarrow$
rest of *cons* (x, y) is $y \leftarrow$

If the predicates "empty," "first," and "rest" are eliminated from the definition of quicksort by a preliminary bottom-up deduction, then the original data-structure-free definition can be replaced by a definition which mixes the data structures with the procedures

sorting *nil* gives *nil* ←
sorting *cons* (x_1, x_2) gives y ← partitioning x_2 by x_1 gives u and v,
sorting u gives u',
sorting v gives v',
appending to u' the list *cons* (x_1, v') gives y.

Clark and Tarnlund [6] show how to obtain a more efficient version of quicksort from the same abstract definition with a different data structure for lists.

Comparing the original data-structure-free definition with the new data-structure-dependent one, we notice another advantage of data-structure-independence: the fact that, with well-chosen names for the interfacing procedures, data-structure-independent programs are virtually self-documenting. For the conventional program which mixes procedures and data structures, the programmer has to provide documentation, external to the program, which explains the data structures. For the well-structured, data-independent program, such documentation is provided by the interfacing procedures and is part of the program.

Despite the arguments for separating data structures and procedures, programmers mix them for the sake of run-time efficiency. One way of reconciling efficiency with good program structure is to use the macroexpansion facilities provided in some programming languages. Macroexpansion flattens the hierarchy of procedure calls before run-time and is the computational analog of the bottom-up and middle-out reasoning provided by some theorem-proving systems. Macro-expansion is also a feature of the program improving transformations developed by Burstall and Darlington.

Notice how our terminology conflicts with Wirth's [39]: program = algorithm + data structure. In our terminology the definition of data structures belongs to the logic component of algorithms. Even more confusingly, we would like to call the logic component of algorithms "logic programs." This is because, given a fixed Horn clause interpreter, the programmer need only specify the logic component. The interpreter can exercise its own control over the way in which the information in the logic component is used. Of course, if the programmer knows how the interpreter behaves, then he can express himself in a manner which is designed to elicit the behavior he desires.

Top-Down Execution of Procedure Calls

In the simplest top-down execution strategy, procedure calls are executed one at a time in the sequence in which they are written. Typically an algorithm can be improved by executing the same procedure calls either as coroutines or as communicating parallel processes. The new algorithm

$A_2 = L + C_2$

is obtained from the original algorithm

$A_1 = L + C_1$

by replacing one control strategy by another leaving the logic of the algorithm unchanged.

For example, executing procedure calls in sequence, the procedure

sorting x gives y ← y is a permutation of x, y is ordered

first generates permutations of x and then tests whether they are ordered. Executing procedure calls as coroutines, the procedure generates permutations, one element at a time. Whenever a new element is generated, the generation of other elements is suspended while it is determined whether the new element preserves the orderedness of the existing partial permutation. This example is discussed in more detail elsewhere [24].

Similarly the procedure calls in the body of the quicksort definition can be executed either in sequence or as coroutines or parallel processes. Executed in parallel, partitioning the rest of x can be initiated as soon as the first elements of the rest are generated. Sorting the output, u and v, of the partitioning procedure can begin and proceed in parallel as soon as the first elements of u and v are generated. Appending can begin as soon as the first elements of u', the sorted version of u, are known.

Philippe Roussel [33] has investigated the problem of showing that two trees have the same lists of leaves:

x and y have the same leaves ← the leaves of x are z,
the leaves of y are z',
z and z' are the same
x and x are the same ←

Executing procedure calls in the sequence in which they are written, the procedure first constructs the list z of leaves of x, then constructs the list z' of leaves of y, and finally tests that z and z' are the same. Roussel has argued that a more sophisticated algorithm is obtained, without changing the logic of the algorithm, by executing the same procedure calls as communicating parallel processes. When one process finds a leaf, it suspends activity and waits until the other process finds a leaf. A third process then tests whether the two leaves are identical. If the leaves are identical, then the first two processes resume. If the leaves are different, then the entire procedure fails and terminates.

The parallel algorithm is significantly more efficient than the simple sequential one when the two trees have different lists of leaves. In this case the sequential algorithm recognizes failure only after both lists have been completely generated. The parallel algorithm recognizes failure as soon as the two lists begin to differ.

The sequential algorithm, whether it eventually succeeds or fails, constructs the intermediate lists z and z' which are no longer needed when the procedure terminates. In contrast, the parallel algorithm can be implemented in such a way that it compares the two lists z and z', one element at a time, without constructing and

saving the initial lists of those elements already compared and found to be identical.

In a high level programming language like SIMULA it is possible to write both the usual sequential algorithms and also coroutining ones in which the programmer controls when coroutines are suspended and resumed. But, as in other conventional programming languages, logic and control are inextricably intertwined in the program text. It is not possible to change the control strategy of an algorithm without rewriting the program entirely.

The arguments for separating logic and control are like the ones for separating procedures and data structures. When procedures are separated from data structures, it is possible to distinguish (in the procedures) what functions the data structures fulfill from the manner in which the data structures fulfill them. When logic is separated from control, it is possible to distinguish (in the logic) what the program does from how the program does it (in the control). In both cases it is more obvious what the program does, and therefore it is easier to determine whether it correctly does what it is intended to do.

The work of Clark and Tarnlund [6] (on the correctness of sorting algorithms) and the unpublished work of Warren and Kowalski (on the correctness of plan-formation algorithms) supports the thesis that correctness proofs are simplified when they deal only with the logic component and ignore the control component of algorithms. Similarly, ignoring control simplifies the derivation of programs from specifications [3, 4, 7, 8, 9, 21].

Top-Down vs. Bottom-Up Execution

Recursive definitions are common in mathematics where they are more likely to be understood bottom-up rather than top-down. Consider, for example, the definition of factorial

The factorial of 0 is 1 ←
The factorial of x is u ← y plus 1 is x,
the factorial of y is v,
x times v is u.

The mathematician is likely to understand such a definition bottom-up, generating the sequence of assertions

The factorial of 0 is 1 ←
The factorial of 1 is 1 ←
The factorial of 2 is 2 ←
The factorial of 3 is 6 ←
etc.

Conventional programming language implementations understand recursions top-down. Programmers, accordingly, tend to identify recursive definitions with top-down execution.

An interesting exception to the rule that recursive definitions are more efficiently executed top-down than bottom-up is the definition of a Fibonacci number, which is both more intelligible and efficient when interpreted bottom-up:

the 0–th Fibonacci number is 1 ←
the 1–th Fibonacci number is 1 ←
the u + 2–th Fibonacci number is x ←
the u + 1–th Fibonacci number is y,
the u–th Fibonacci number is z,
y plus z is x.

(Here the terms $u + 2$ and $u + 1$ are expressions to be evaluated rather than terms representing data structures. This notation is an *abbreviation* for the one which has explicit procedure calls in the body to evaluate $u + 2$ and $u + 1$.)

Interpreted top-down, finding the u + 1–th Fibonacci number reintroduces the subproblem of finding the u–th Fibonacci number. The top-down computation is a tree whose nodes are procedure calls, the number of which is an exponential function of u. Interpreting the same definition bottom-up generates the sequence of assertions

the 0–th Fibonacci number is 1 ←
the 1–th Fibonacci number is 1 ←
the 2–th Fibonacci number is 2 ←
the 3–th Fibonacci number is 3 ←
etc.

The number of computation steps is a linear function of u.

In this example, bottom-up execution is also less space-consuming than top-down execution. Top-down execution uses space which is proportional to u, whereas bottom-up execution needs to store only two assertions and can use a small constant amount of storage. That only two assertions need to be stored during bottom-up execution is a consequence of the deletion rules for the connection graph proof procedure [25]. As Bibel observes, the greater efficiency of bottom-up execution disappears if similar procedure calls are executed top-down only once. This strategy is, in fact, an application of Warren's generalization of the Earley parsing algorithm.

Strategies for Investigating Alternative Procedures

When more than one procedure has a conclusion which matches a given procedure call, the logic component does not determine the manner in which the alternative procedures are investigated. Sequential exploration of alternatives gives rise to iterative algorithms.

Although in theory all iterations can be replaced by top-down execution of recursive definitions, in practice some iterations might better be thought of as bottom-up execution of recursive definitions (as in the definition of factorial). Other iterations can better be regarded as controlling a sequential search among alternatives.

Assume, for example, that we are given data about individuals in the parenthood relationship:

Parent (Zeus, Ares) ←
Parent (Hera, Ares) ←
Parent (Ares, Harmonia) ←
Parent (Semele, Dionisius) ←
Parent (Zeus, Dionisius) ←
etc.

Suppose that the problem is to find a grandchild of Zeus

← Grandparent (Zeus, *u*)

using the definition of grandparent. In a conventional programming language, the parenthood relationship might be stored in a two-dimensional array. A general procedure for finding grandchildren (given grandparents) might involve two iterative loops, one nested inside the other, with a jump out when a solution has been found. Similar behavior is obtained by interpreting the grandparent procedure top-down, executing procedure calls one at a time, in the sequence in which they are written, trying alternative procedures (assertions in this case) one at a time in the order in which they are written. The logical analysis of the conventional iterative algorithm does not concern recursion but involves sequential search through a space of alternatives. The sequential search strategy is identical to the backtracking strategy for executing nondeterministic programs [18].

Representation of data by means of clauses, as in the family relationships example, rather than by means of terms, is similar to the relational model of databases [10]. In both cases data is regarded as relationships among individuals. When data is represented by conventional data structures (terms), the programmer needs to specify in the logic component of programs and queries both how data is stored and how it is retrieved. When data is represented relationally (by clauses), the programmer needs only to specify data in the logic component; the control component manages both storage and retrieval.

The desirability of separating logic and control is now generally accepted in the field of databases. An important advantage is that storage and retrieval schemes can be changed and improved in the control component without affecting the user's view of the data as defined by the logic component.

The suitability of a search strategy for retrieving data depends upon the structure in which the data is stored. Iteration, regarded as sequential search, is suitable for data stored sequentially in arrays or linked lists. Other search strategies are more appropriate for other data structures, such as hash tables, binary trees, or semantic networks. McSkimin and Minker [29], for example, store clauses in a manner which facilitates both parallel search and finding all answers to a database query. Deliyanni and Kowalski [15], on the other hand, propose a path-finding strategy for retrieving data stored in semantic networks.

Representation of data by means of terms is a common feature of Horn clause programs written in PROLOG [12, 33, 38]. Tärnlund [36], in particular, has investigated the use of terms as data structures in Horn clause programs. Several PROLOG programs employ a relational representation of data. Notable among these are Warren's [37] for plan-formation and those for drug analysis written in the Ministry of Heavy Industry in Budapest [14].

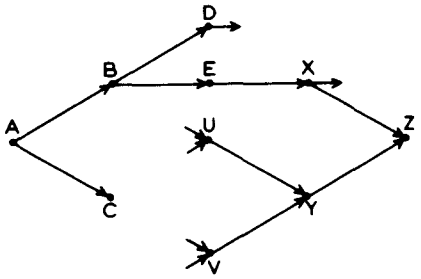
Two Analyses of Path-Finding Algorithms

The same algorithm A can often be analyzed in different ways:

$A = L_1 + C_1 = L_2 + C_2$.

Some of the behavior determined by the control component C_1 in one analysis might be determined by the logic component L_2 in another analysis. This has significance for understanding the relationship between programming style and execution facilities. In the short term sophisticated behavior can be obtained by employing simple execution strategies and by writing complicated programs. In the longer term the same behavior may be obtained from simpler programs by using more sophisticated execution strategies.

The path-finding problem illustrates a situation in which the same algorithm can be analyzed in different ways. Consider the problem of finding a path from A to Z in the following directed graph.



In one analysis, we employ a predicate *Go* (x) which states that it is possible to go to x. The problem of going from A to Z is then represented by two clauses. One asserts that it is possible to go to A. The other denies that it is possible to go to Z. The arc directed from A to B is represented by a clause which states that it is possible to go to B if it is possible to go to A:

Go $(A) \leftarrow$	$\leftarrow$ *Go* (Z)
Go $(B) \leftarrow$ *Go* (A)	*Go* $(Z) \leftarrow$ *Go* (X)
Go $(C) \leftarrow$ *Go* (A)	*Go* $(Z) \leftarrow$ *Go* (Y)
Go $(D) \leftarrow$ *Go* (B)	*Go* $(Y) \leftarrow$ *Go* (U)
Go $(E) \leftarrow$ *Go* (B)	*Go* $(Y) \leftarrow$ *Go* (V)
Go $(X) \leftarrow$ *Go* (E)	etc.

Different control strategies determine different path-finding algorithms. Forward search from the initial node A is bottom-up reasoning from the initial assertion *Go* $(A) \leftarrow$. Backward search from the goal node Z is top-down reasoning from the initial goal statement $\leftarrow$ *Go* (Z). Bidirectional search from both the initial node and the goal node is the combination of top-down and bottom-up reasoning. Whether the path-finding algorithm investigates one path at a time (depth-first) or develops all paths simultaneously (breadth-first) is a

matter of the search strategy used to investigate alternatives.

In the second analysis, we employ a predicate $Go^*(x, y)$ which states that it is possible to go from x to y. In addition to the assertions which describe the graph and the goal statement which describes the problem, there is a single clause which defines the logic of the path-finding algorithms:

$\leftarrow Go^*(A, Z)$
$Go^*(A, B) \leftarrow$ $\quad Go^*(X, Z) \leftarrow$
$Go^*(A, C) \leftarrow$ $\quad Go^*(Y, Z) \leftarrow$
$Go^*(B, D) \leftarrow$ $\quad Go^*(U, Y) \leftarrow$
$Go^*(B, E) \leftarrow$ $\quad Go^*(V, Y) \leftarrow$
$Go^*(E, X) \leftarrow$ $\quad$ etc.
$Go^*(x, y) \leftarrow Go^*(x, z), G0^*(z, y)$

Here both forward search from the initial node A and backward search from the goal node Z are top-down reasoning from the initial goal statement $\leftarrow Go^*(A, Z)$. The difference between forward and backward search is the difference in the choice of subproblems in the body of the path-finding procedure. Solving $Go^*(x, z)$ before $Go^*(z, y)$ is forward search, and solving $Go^*(z, y)$ before $Go^*(x, z)$ is backward search. Coroutining between the two subproblems is bidirectional search. Bottom-up reasoning from the assertions which define the graph generates the transitive closure, effectively adding a new arc to the graph directed from node x to node y, whenever there is a path from x to y.

Many problem domains have in common with path-finding that an initial state is given and the goal is to achieve some final state. The two representations of the path-finding problem exemplify alternative representations which apply more generally in other problem domains. Warren's plan-formation program [37], for example, is of the type which contains both the given and the goal state as different arguments of a single predicate. It runs efficiently with the sequential execution facilities provided by PROLOG. The alternative formulation, which employs a predicate having one state as argument, is easier to understand but more difficult to execute efficiently.

Even the definition of factorial can be represented in two ways. The formulation discussed earlier corresponds to the one-place predicate representation of path-finding. The formulation here corresponds to the two-place predicate representation. Read

$F(x, y, u, v)$

as stating that

the factorial of x is y
given that the factorial of u is v.
$F(x, y, x, y) \leftarrow$
$F(x, y, u, v) \leftarrow u$ plus 1 is u', u' times v is v', $F(x, y, u', v')$.

To find the factorial of an integer represented by a term t, a single goal statement incorporates both the goal and the basis of the recursion

$\leftarrow F(t, y, 0, 1)$.

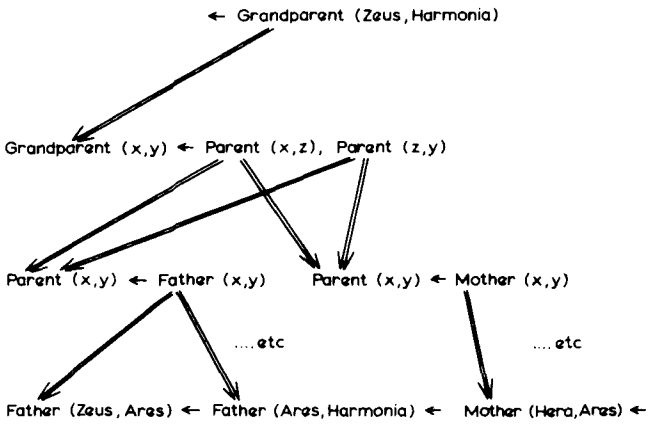
The new formulation of factorial executed in a simple top-down sequential manner behaves in the same way as the original formulation executed in a mixed top-down, bottom-up fashion.

A Notation for Expressing Control Information

The distinction between top-down and bottom-up execution can be expressed in a graphical notation which uses arrows to indicate the flow of control. The same notation can be used to represent different combinations of top-down and bottom-up execution. The notation *does not*, however, aim to provide a complete language for expressing useful control information.

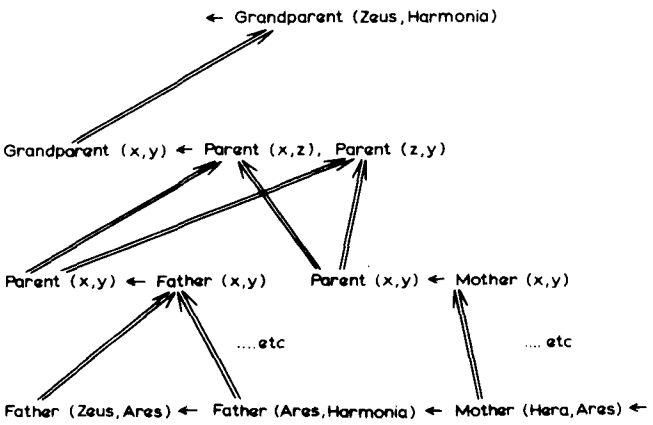
Arrows are attached to atoms in clauses to indicate the direction of transmission of processing activity from clause to clause. For every pair of matching atoms in the initial set of clauses (one atom in the conclusion of a clause and the other among the conditions of a clause), there is an arrow directed from one atom to the other.

For top-down execution, arrows are directed from conditions to conclusions. For the grandparent problem, we have the graph:



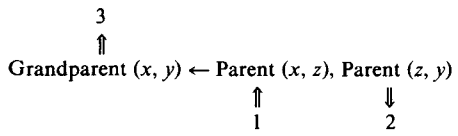
Processing activity starts with the initial goal statement. It transmits activity to the body of the grandparent procedure, whose procedure calls, in turn, activate the parenthood definitions. The database of assertions passively accepts processing activity without transmitting it to other clauses.

For bottom-up execution, arrows are directed from conclusions to conditions:



Processing activity originates with the database of initial assertions. They transmit activity to the parenthood definitions, which, in turn, activate the grandparent definition. Processing terminates when it reaches all the conditions in the passive initial goal statement.

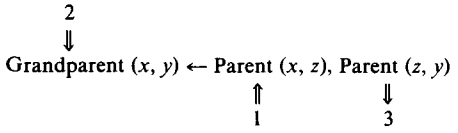
The grandparent definition can be used in a combination of top-down and bottom-up methods. Using numbers to indicate sequencing, we can represent different combinations of top-down and bottom-up execution. For simplicity we only show the control notation associated with the grandparent definition. The combination of logic and control indicated by



represents the algorithm which

(1) waits until x is asserted to be parent of z, then
(2) finds a child y of z, and finally
(3) asserts that x is grandparent of y.

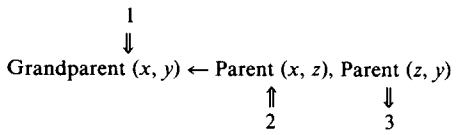
The combination indicated by



represents the algorithm which

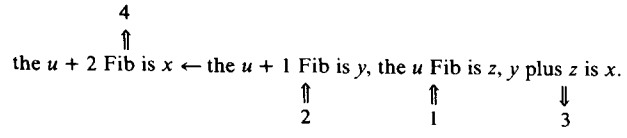
(1) waits until x is asserted to be parent of z, then
(2) waits until it is given the problem of showing that x is grandparent of y,
(3) which it then attempts to solve by showing that z is parent of y.

The algorithm represented by



(1) responds to the problem of showing that x is grandparent of y,
(2) by waiting until x is asserted to be parent of z, and then
(3) attempting to show that z is parent of y.

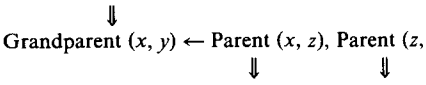
Using the arrow notation, we can be more precise than before about the bottom-up execution of the recursive definition of Fibonacci number. The bottom-up execution referred to previously is, in fact, a mixture of bottom-up and top-down execution:



Arrow notation can also be used to give a procedural interpretation of non-Horn clauses. The definition of subset, for example, "x is a subset of y if, for all z, if z belongs to x, then z belongs to y," gives rise to a procedure which shows that x is a subset of y by showing that every member of x is a member of y. It does this by asserting that some individual belongs to x and by attempting to show that the same individual belongs to y. The name of the individual must be different from the name of any individual mentioned elsewhere, and it must depend upon x and y (being different for different x and y). In clausal notation with arrows to indicate control, the definition of subset becomes

Given the goal of showing that x is a subset of y, the first clause asserts that the individual named *arb* (x, y) belongs to x and the second clause generates the goal of showing that *arb* (x, y) belongs to y.

The grandparent definition illustrates the inadequacy of the arrow notation for expressing certain kinds of control information. Suppose that the grandparent definition is to be used entirely top-down.



The effective sequencing of procedure calls in the body of the procedure depends upon the parameters of the activating procedure call:

(1) If the problem is to find a grandchild y of a given x, then it is more effective (i) first to find a child z of x; (ii) and then to find a child y of z.
(2) If the problem is to find a grandparent x of a given y, then it is better (i) first to find a parent z of y; (ii) and then to find a parent x of z.

Such sequencing of procedure calls depending on the pattern of input and output cannot be expressed in the arrow notation.

In relational database query languages, input-sensitive sequencing of procedure calls needs to be determined by the data retrieval system rather than by the user. Consider, for example, a database which defines the following relations:

Supplier (x, y, z)	supplier number x has name y and status z,
Part (x, y, z)	part number x has name y and unit cost z,
Supply (x, y, z)	supplier number x supplies part number y in quantity z.

Given the query

What is the name of suppliers of pens?
← Answer (y)
Answer (y) ← Supplier (x, y, z), Supply (x, u, v), Part $(u$, pen, $w)$

the system needs to determine that, for the sake of efficiency, the last procedure call should be executed first; whereas given the query

What is the name of parts supplied by Jones?
← Answer (y)
Answer (y) ← Supplier (x, Jones, z), Supply (x, u, v), Part (u, y, w)

the first procedure call should be executed before the others.

The arrow notation can be used to control the behavior of a connection graph theorem-prover [12]. The links of a connection graph are turned into arrows by giving them a direction. A link may be activated (giving rise to a resolvent) only if the link is connected to a clause all of whose links are outgoing. The links of the derived resolvent inherit the direction of the links from which they descend in the parent clauses. Connection graphs controlled in such a manner are similar to Petri nets [16].

Conclusion

We have argued that conventional algorithms can usefully be regarded as consisting of two components:

(1) a logic component which specifies what is to be done and
(2) a control component which determines how it is to be done.

The efficiency of an algorithm can often be improved by improving the efficiency of the control component without changing the logic and therefore without changing the meaning of the algorithm.

The same algorithm can often be formulated in different ways. One formulation might incorporate a clear statement, in the logic component, of the knowledge to be used in solving the problem and achieve efficiency by employing sophisticated problem-solving strategies in the control component. Another formulation might produce the same behavior by complicating the logic component and employing a simple problem-solving strategy.

Although the trend in databases is towards the separation of logic and control, programming languages today do not distinguish between them. The programmer specifies both logic and control in a single language while the execution mechanism exercises only the most rudimentary problem-solving capabilities. Computer programs will be more often correct, more easily improved, and more readily adapted to new problems when programming languages separate logic and control, and when execution mechanisms provide more powerful problem-solving facilities of the kind provided by intelligent theorem-proving systems.

Acknowledgments. The author has benefited from valuable discussions with K. Clark, A. Colmerauer, M. van Emden, P. Hayes, P. Roussel, S. Tärnlund, and D. Warren. Special thanks are due to W. Bibel, K. Clark, M. van Emden, P. Hayes, and D. Warren for their helpful comments on earlier drafts of this paper. This research was supported by a grant from the Science Research Council. The final draft of this paper was completed during a visiting professorship held in the School of Computer and Information Science at the University of Syracuse.

Received December 1976; revised February 1978

References

1. Bibel, W., and Schreiber, J. Proof procedures in a Gentzen-like system of first-order logic. Proc. Int. Comptng. Symp., North-Holland Pub. Co., Amsterdam, 1975, pp. 205–212.
2. Bibel, W. Programmieren in der Sprache der Prädikatenlogik. Eingereicht als Habibitationsarbeit. Fachbereich Mathematik, Techn. München, Jan. 1975. Shorter versions published as: Prädikatives Programmieren. Lecture Notes in Computer Science, *33*, GI-2. Fachtagung über Automatentheorie und formale Sprachen, Springer-Verlag, Berlin, Heidelberg, New York, 1975, pp. 274–283. And as: Predicative Programming. Séminaires IRIA, théorie des algorithms, des languages et de la programmation 1975–1976, IRIA, Roquencourt, France, 1977.
3. Bibel, W. Syntheses of strategic definitions and their control. Bericht Nr. 7610, *Abt. Mathem.*, Techn. München, 1976.
4. Bibel, W. A uniform approach to programming. Bericht Nr. 7633, *Abtl. Mathem.*, Techn. München, 1976.
5. Bledsoe, W.W., and Bruell, P. A man-machine theorem-proving system. *Artif. Intell. 5* (Spring 1974), 51–72.
6. Clark, K.L., and Tärnlund, S.A. A first order theory of data and programs. *Information Processing 77*, North-Holland Pub. Co., Amsterdam, 1977, pp. 939–944.
7. Clark, K., and Sickel, S. Predicate logic: A calculus for the formal derivation of programs. Proc. Int. Joint Conf. Artif. Intell., 1977.
8. Clark, K. The synthesis and verification of logic programs. Res. Rep., Dept. Comptng. and Control, Imperial College, London, 1977.
9. Clark, K., and Darlington, J. Algorithm analysis through synthesis. Res. Rep., Dept. Comptng. and Control, Imperial College, London, Oct. 1977.
10. Codd, E.F. A relational model for large shared databases. *Comm. ACM 13*, 6 (June 1970), 377–387.
11. Codd, E.F. Relational completeness of data base sublanguages. In *Data Base Systems*, R. Rustin, Ed., Prentice-Hall, Englewood Cliffs, N.J., 1972.
12. Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. Un systeme de communication homme-machine en francais. *Rapport preliminaire, Groupe de Res. en Intell. Artif.*, U. d'Aix-Marseille, Luminy, 1972.
13. Darlington, J., and Burstall, R.M. A system which automatically improves programs. Proc. of Third Int. Joint Conf. Artif. Intell., S.R.I., Menlo Park, Calif., 1973, pp. 437–542.
14. Darvas, F., Futo, I., and Szeredi, P. Logic based program for predicting drug interactions. To appear in *Int. J. Biomedical Computing.*
15. Deliyanni, A., and Kowalski, R.A. Logic and semantic networks. *Comm. ACM 22*, 3 (March 1979), 184–192.
16. Earley, J. An efficient context-free parsing algorithm. *Comm. ACM 13*, 2 (Feb. 1970), 94–102.
17. van Emden, M.H. Programming in resolution logic. To appear in *Machine Representations of Knowledge* published as *Machine Intelligence 8*, E.W. Elcock and D. Michie, Eds., Ellis Horwood and John Wylie.
18. Floyd, R.W. Non-deterministic algorithms. *J. ACM 14*, 4 (Oct. 1967), 636–644.
19. Hayes, P.J. Computation and deduction. Proc. 2nd MFCS Symp., Czechoslovak Acad. of Sciences, 1973, pp. 105–118.
20. Hewitt, C. Planner: A language for proving theorems in robots. Proc. of Int. Joint Conf. Artif. Intell., Washington, D.C., 1969, pp. 295–301.
21. Hogger, C. Deductive synthesis of logic programs. Res. Rep., Dept. Comptng. and Control, Imperial College, London, 1977.
22. Kleene, S.C. *Introduction to Metamathematics.* Van Nostrand, New York, 1952.

23. Kowalski, R.A. Logic for problem-solving. Memo No. 75, Dept. Comput. Logic, U. of Edinburgh, 1974.
24. Kowalski, R.A. Predicate logic as programming language. *Information Processing 74*, North-Holland Pub. Co., Amsterdam, 1974, pp. 569–574.
25. Kowalski, R.A. A proof procedure using connection graphs. *J. ACM 22*, 4 (Oct. 1974), 572–95.
26. Kowalski, R.A., and Kuehner, D. Linear resolution with selection function. *Artif. Intell. 2* (1971), 227–260.
27. Loveland, D.W. A simplified format for the model-elimination theorem-proving procedure. *J. ACM 16*, 3 (July 1969), 349–363.
28. Mac Carthy, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Bratfort and D. Hirschberg, Eds., North-Holland Pub. Co., Amsterdam, 1967.
29. McSkimin, J.R., and Minker, J. The use of a semantic network in a deductive question-answering system. Proc. Int. Joint Conf. Artif. Intell., 1977, pp. 50–58.
30. Petri, C.A. Grundsatzliches zur Beschreibung diskreter Prozesse. 3. Colloq. uber Automathentheorie, Birkhauser Verlag, Basel, Switzerland, 1967.
31. Pratt, V.R. The competence/performance dichotomy in programming. Proc. Fourth ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages, Santa Monica, Calif., Jan. 1977, pp. 194–200.
32. Robinson, J.A. Automatic deduction with hyper-resolution. *Int. J. Comput. Math. 1* (1965), 227–34.
33. Roussel, P. Manual de reference et d'Utilisation. Groupe d'Intell. Artif., UER, Marseille-Luminy, France, 1975.
34. Schwarz, J. Using annotations to make recursion equations behave. Res. Memo, Dept. Artif. Intell., U. of Edinburgh, 1977.
35. Sickel, S. A search technique for clause interconnectivity graphs. *IEEE Trans. Comptrs.* (Special Issue on Automatic Theorem Proving), Aug. 1976.
36. Tärnlund, S.A. An interpreter for the programming language predicate logic. Proc. Int. Joint Conf. Artif. Intell., Tiblisi, 1975, pp. 601–608.
37. Warren, D. A system for generating plans. Memo No. 76, Dept. Comput. Logic, U. of Edinburgh, 1974.
38. Warren, D., Pereira, L.M., and Pereira, F. PROLOG—The language and its implementation compared with LISP. Proc. Symp. on Artif. Intell. and Programming Languages; SIGPLAN Notices (ACM) *12*, 8; SIGART Newsletters (ACM) *64* (Aug. 1977), pp. 109–115.
39. Wirth, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Professional Activities: Calendar of Events

ACM's calendar policy is to list open computer science meetings that are held on a not-for-profit basis. Not included in the calendar are educational seminars, institutes, and courses. Submittals should be substantiated with name of the sponsoring organization, fee schedule, and chairman's name and full address.

One telephone number contact for those interested in attending a meeting will be given when a number is specified for this purpose.

All requests for ACM sponsorship or cooperation should be addressed to Chairman, Conferences and Symposia Committee, Seymour J. Wolfson, 643 MacKenzie Hall, Wayne State University, Detroit, MI 48202, with a copy to Louis Fiora, Conference Coordinator, ACM Headquarters, 1133 Avenue of the Americas, New York, NY 10036; 212 265-6300. For European events, a copy of the request should also be sent to the European Representative. Technical Meeting Request Forms for this purpose can be obtained from ACM Headquarters or from the European Regional Representative. Lead time should include 2 months (3 months if for Europe) for processing of the request, plus the necessary months (minimum 3) for any publicity to appear in *Communications*.

■ This symbol indicates that the Conferences and Symposia Committee has given its approval for ACM sponsorship or cooperation.
In this issue the calendar is given in its entirety. New Listings are shown first; they will appear next month as Previous Listings.

NEW LISTINGS

6-8 August 1979
Seventh Conference on Electronic Computation, Washington University, St. Louis, Mo. Sponsor: ASCE, Washington University. Contact: C. Wayne Martin, 212 Bancroft Hall, University of Nebraska, Lincoln, NE 68588.

7-8 August 1979
Workshop on the Use of Computers in Teaching Statistics, University of New Hampshire, Durham, NH. Sponsor: University of New Hampshire. Contact: Office of Academic Computing, University of New Hampshire, 304 McConnell Hall, Durham, NH 03824; 603 862-1990.

12-14 September 1979
7th SIMULA Users' Conference, Hotel Regina Olga, Cernobbio, Lake Como, Italy. Sponsor: Association of SIMULA Users. Contact: Eileen Schreiner, Norwegian Computing Center, Postboks 335, Blindern, Oslo 3, Norway.

8 November 1979
Annual Western Systems Conference, Los Angeles, Calif. Sponsor: Association for Systems Management. Gen. chm: Sylvia Twomey, 18700 Yorba Linda Blvd., Apt. 47, Yorba Linda, CA 92686; 714 993-6730.

27-30 November 1979
CAUSE National Conference, Planning Higher Education Information Systems for the 1980s, Orlando, Fla. Sponsor: CAUSE. Contact: CAUSE, 737 Twenty-Ninth St., Boulder, CO 80303; 303 492-7353.

14-15 February 1980
■ **ACM SIGCSE Technical Symposium on Computer Science Education,** Kansas City, Mo. Sponsor: ACM SIGCSE. Conf. chm: William C. Bulgren, Dept. of Computer Science, The University of Kansas, Lawrence, KS 66044; 913 864-4482.

12-14 March 1980
International Symposium on Distributed Databases, Versailles, France. Sponsor: IRIA. Contact: Symposium Secretariat, IRIA, Services des Relations Extérieures, Domaine de Voluceau-BP 105, 78150 Le Chesnay, France.

19-21 March 1980
■ **13th Annual Simulation Symposium,** Tampa, Fla. Sponsors: ACM SIGSIM, IEEE-CS, SCS. Symp. chm: Harvey Fisher, Alcan Products, Box 511, Warren, OH 44482; 216 841-3416.

28 March-3 April 1980
Sixth International ALLC Symposium on Computers in Literary and Linguistic Research, University of Cambridge, England. Sponsor: Association for Literary and Linguistic Research. Contact: J.L. Dawson, Secretary, 1980 Symposium, Literary and Linguistic Computing Centre, Sidgewick Site, Cambridge CB3 9DA, England.

2 May 1980
■ **Role of Documentation in the Project Life Cycle,** New York City. Sponsors: ACM SIGDOC, SIGCOSIM. Conf. chm: Belden Menkus, Box 85, Middleville, NJ 07855; 201 383-3928.

19-22 May 1980
■ **NCC 80,** Anaheim, Calif. Sponsor: AFIPS. Contact: Jerry Chriffriller, AFIPS, 210 Summit Ave., Montvale, NJ 07645; 201 391-9810.

3-6 June 1980
4th International IFAC Conference on Instrumentation and Automation in the Paper, Rubber, Plastics, and Polymerization Industries, Ghent, Belgium. Sponsor: IFAC. Contact: 4th IFAC-P.R.P. Automation Conference, Jan Van Rijswijcklaan, 58, B-2000 Antwerp, Belgium.

16-18 June 1980
IFAC/IFIP Symposium on Automation for Safety in Shipping and Offshore Operations, Trondheim, Norway. Sponsors: IFAC, IFIP SINTEF, Norwegian Petroleum Directorate. Contact: SINTEF, Automatic Control Division, N-7034 Trondheim-NTH, Norway.

PREVIOUS LISTINGS

15-20 July 1979
International Users' Conference, Cambridge, Mass. Sponsor: Harvard University Laboratory for Computer Graphics and Spatial Analysis. Contact: Kathleen Quigley, Center for Management Research (conference coordinators), 850 Boylston St., Chestnut Hill, MA 02167.

16-18 July 1979
1979 Summer Computer Simulation Conference, Toronto, Ont., Canada. Sponsors: SCS, ISA, AMS, SHARE. Gen. chm: A.J. Schiewe, c/o The Aerospace Corp., Box 92957, Los Angeles, CA 90009; 213 648-6120.

16-20 July 1979
Sixth International Colloquium on Automata, Languages, and Programming, Technical University of Graz, Austria. Sponsor: European Association for Theoretical Computer Science. Contact: H. Maurer, Institut für Informationsverarbeitung, Techn. Universität Graz, Steyrergasse 17, A-8010-Graz, Austria.

18-20 July 1979
Fifth South African Symposium on Numerical Mathematics, University of Natal, Durban, South Africa. Sponsor: University of Natal. Contact: H. Roland Weistroffer, Computer Science Dept., University of Natal, King George V Avenue, Durban, 4001, Republic of South Africa.

27-29 July 1979
Seminar on Scientific Go Theory (with European Go Congress 1979) near Bonn, W. Germany. Contact: Klaus Heine, Kleiststr. 67, 294 Wilhelmshaven, W. Germany.

6-8 August 1979
1979 Pattern Recognition and Image Processing Conference, Chicago, Ill. Sponsor: IEEE-CS. Contact: PRIP79, Box 639, Silver Spring, MD.

6-8 August 1979
Seventh Conference on Electronic Computation, St. Louis, Mo. Sponsors: ASCE, Washington University. Contact: C. Wayne Martin, 212 Bancroft Bldg., University of Nebraska, Lincoln, NE 68588.

6-10 August 1979
■ **SIGGRAPH 79, Sixth Annual Conference on Computer Graphics and Interactive Techniques,** Chicago, Ill. Sponsor: ACM SIGGRAPH. Conf. co-chm: Thomas DeFanti, Bruce H. McCormick, Dept. of Information Engineering, University of Illinois at Chicago Circle, Box 4348, Chicago, IL 60680; 312 996-2315.

6-10 August 1979
■ **ACM SIGPLAN Symposium on Compiler Construction,** Brown Palace Hotel, Denver, Colo. Sponsor: ACM SIGPLAN. Conf. chm: Fran Allen, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

11-12 August 1979
Association for Computational Linguistics 17th Annual Meeting, University of California (San Diego), La Jolla, Calif. Sponsor: ACL. Contact: Donald E. Walker, ACL Sec'y-Treas., SRI International, Menlo Park, CA 94025.

13-15 August 1979
■ **Conference on Simulation, Measurement, and Modeling of Computer Systems,** Boulder, Colo. Sponsors: ACM SIGMETRICS, SIGSIM, NBS. Conf. chm: Paul F. Roth, National Bureau of Standards, A-265 Technology Bldg., Washington, DC 20234.

16-17 August 1979
■ **SIGCPR 16th Annual Conference on Computer Personnel Research,** Princeton, N.J. Sponsor: ACM SIGPCR. Conf. chm: T.C. Willoughby, Management Science, College of Business Administration, Ball State University, Muncie, IN 47306; 317 285-1265.

16-18 August 1979
IFAC Symposium on Computer Applications in Large Scale Power Systems, Bangalore, India. Sponsor: International Federation of Automatic Control. Contact: Institution of Engineers, 8 Cookhale Road, Calcutta-700020, India.

19-22 August 1979
3rd Rocky Mountain Symposium on Microcomputers, Pingree Park, Colo. Sponsor: Colorado State University. Contact: Carolyn Frye, Office of Conferences and Institutes, Colorado State University, Fort Collins, CO 80523; 303 491-6222.

19-24 August 1979
Seventeenth Annual URISA Conference, San Diego, Calif. Sponsor: Urban and Regional Information Systems Association. Prog. chm: Lee P. Johnston, URISA Conf. Prog. Chm., 823 Monticello Drive, Escondido, CA 92025.

20-22 August 1979
Fourth International Conference on Computers and the Humanities, Dartmouth College, Hanover, N.H. Sponsors: Dartmouth College and the Association for Computers and the Humani-

(Calendar continued on p. 439)