PROLOG

Kenneth A. Bowen
Syracuse University

I. Theorem Provers as Logic Machines

Exactly 100 years ago, the first-order predicate calculus was created and defined by Gottlob Frege. In the ensuing century his system was studied and refined by such logicians as Bertrand Russell, David Hilbert, Kurt Godel, Jacques Herbrand, Alonzo Church, and Alan Turing. In the 1950's and 60's attempts were made to use the results of these studies (especially those of Herbrand) in order to program computers to prove theorems automatically. These attempts introduced a new demon to the study of logic: ferocious computational complexity. The investigations of methods to avoid this demon led to the development of new systems of logic which are equivalent to the traditional systems, but are more suited to the efficient mechanical construction of proofs. The most notable among these is the resolution system of J. Alan Robinson [1965],[1979]. Cordell Green [1969] proposed the use of resolution systems in the construction of deductive question-answering systems, and this proposal eventually led Robert Kowalski [1974] to propose the so-called procedural interpretation of logic which forms the basis for the use of logic as a programming language.

Consider a formula

$$\forall x1,\ldots,xn\ \exists y1,\ldots,ym\ A(x1,\ldots,ym), \quad (1)$$

where A may be an arbitrary formula. It is built up from various primitive atomic formulas, for example "less_than(x,y)," or "tree(t)," or "zip(x,y,z)." The atomic formulas are abstract symbol strings having no intrinsic meaning apart from what a user imparts to them. Given a choice of meanings for the atomic formulas, the meaning of a compound such as (1) is determined in a standard and fixed way, as given in any of hundreds of logic textbooks. Suppose that with such a choice of meanings, formula (1) turns out to be true. Then it can be regarded as describing the input-output relation for an abstract machine or program, taking x's as input and yielding y's as output. If the y's are not uniquely determined by the x's, the machine is non-deterministic.

Virtually all automatic theorem-proving systems are constructive in the sense that when faced with a formula of the form (1), and given terms s1,...,sm, the system will attempt to find terms t1,...,tm such that it can prove

$$A(s1,\ldots,sn,t1,\ldots,tm) \quad (2)$$

is a true formula. Since the terms t1,...,tm are built up from x1,...,xn using the available symbols of the language at hand, this substitution (y1,...)-->(t1,...) can be regarded as a computation procedure: given particular terms s1,...,sn for the x's, construct the appropriate t1',...,tm' out of the s1,...,sn. One view of this process is that in searching for a proof of (1), the theorem-prover constructs the expression

$$A(x1,\ldots,xn, \mathcal{f}1,\ldots,\mathcal{f}m), \quad (3)$$

where $\mathcal{f}1,\ldots,\mathcal{f}m$ are metavariables ranging over expressions of the language, and attempts to find values (expressions) for these variables so that the resulting formula (2) is provable. Resolution systems, as well as other systems, will carry out this search in a pattern-directed way so that if the x1,...,xn are replaced by s1,...,sn, yielding

$$A(s1,\ldots,sn, \mathcal{f}1,\ldots,\mathcal{f}m), \quad (4)$$

the resulting search will produce appropriate instances t1',...,tm'.

Thus the formula A(x1,...,ym) together with the automatic theorem-prover constitute a program/machine pair which, given particular inputs s1,...,sn, computes outputs t1',...,tm'.

Seen this way, any automatic theorem-prover which behaves in such a pattern-directed constructive manner can be used as an interpreter or logic machine which will run logic formulas as programs. The fly in the ointment (or if you like, the sand in the gears) is the demon of computational complexity: most theorem-provers run much too slowly to be of practical use as logic machines. However, since a few very efficient practical logic machines do exist, the logic programming programme is both viable and exciting. The general programme has these two constituents:

(i) to devise systems of logic suitable for easily expressing real computational problems;

(ii) to construct suitable theorem provers for these logics which can be used as practical logic machines.

II. PROLOG Systems

The most completely engineered logic programming systems to date are the PROLOG systems (cf. P. Roussel [1975], G. Roberts[1977], D. Warren et al.[1977],), which are based on ideas of Kowalski, Colmerauer, Roussel, Hayes, and Boyer and Moore (cf. Kowalski [1974], p.573). With minor variations, the syntax of these systems runs as follows. Both integers and identifiers are atomic terms. Identifiers preceeded by ' _ ' are variables. One can optionally choose to take all identifiers beginning with an uppercase letter as variables; this option will be exercised here. Compound terms are expressions of the form a(b1,...bn), where a is an identifier and b1,...bn are terms. An expression of the form

B. (5)

is a (unit) clause (or unconditional assertion) provided that B is a term. When B and C1,...Cn are all terms,

B <-- C1,...,Cn (6)

is a clause (or conditional assertion). Collectively, conditional and unconditional assertions are referred to as Horn clauses. A program is a list of clauses. The terms appearing at top-level in clauses are regarded as atomic formulas in the usual systems of predicate logic. Clause (5) is treated logically as

$\forall x1,...,xn$ B (7)

where x1,...,xn are all the variables occurring in B, and clause (6) is understood logically as

$\forall x1,...,xn[C1 \& ... \& Cn \Rightarrow B]$, (8)

where => indicates logical implication. Programs are treated logically as the conjunction of the clauses of which they are composed. Consider the following examples:

```
sum(X,0,X).
sum(X,s(Y),s(Z))  <--  sum(X,Y,Z).        (9)

prod(X,0,0).
prod(X,s(Y),Z)    <--                     (10)
            prod(X,Y,W),sum(W,X,Z).

fact(0,s(0)).
fact(s(X),Z)  <--                         (11)
            fact(X, Y),
prod(S(X),Y,Z).
```

Here sum(X,Y,Z) means that X + Y = Z, prod(X,Y,Z) means that X*Y = Z, and fact(X,Y) means that Y is the factorial of X. These are easily recognizable as the familiar inductive definitions of the addition, multiplication, and factorial functions. (While arithmetic can be done this way, PROLOG systems provide direct access to machine arithmetic as will be shown later.) Let SUM be the ordinary logical correspondent of (9):

```
SUM  =  ∀X    sum(X,0,X)    &              (12)
  ∀X,Y,Z[sum(X,Y,Z)
      => sum(X,S(Y),s(Z))].
```

Similarly, let PROD and FACT be the logical correspondents of (10) and (11), respectively. Consider the formula

```
SUM    &    PROD    &    FACT              (13)
     => fact(s(s(s(0))),Y).
```

A suitable prover attacking this formula would find the satisfying substitution:

Y = s(s(s(s(s(0))))).

III. The Procedural Interpretation of Logic

The PROLOG processor does not make use of (13), however. Instead, it maintains the clauses (9) - (11) as an internal database, and accomplishes its searches by:

(a) utilizing the notion of goal clause;

(b) utilizing the procedural interpretation of non-goal clauses;

(c) utilizing the unification process for its pattern-matching.

A goal clause is an expression of the form

<-- C1,...,Cn. (14)

This is treated logically as

$$\forall x1,\ldots,xk \text{ not-}(C1 \;\&\; \ldots \;\&\; Cn), \quad (15)$$

where x1,...,xk are all the variables occurring in any of C1,...,Cn. If DB is the collection of (non-goal) clauses in the processor's database, the logical effect of submitting a goal clause (14) is to cause the processor to attempt to determine whether or not (15) is inconsistent with DB. The processor succeeds if and only if it proves (15) to be inconsistent with DB, and hence that $\exists x1 \ldots xk(C1\&\ldots\&Cn)$ is a logical consequence of DB.

The procedural interpretation of clauses runs as follows. A clause

$$f(a1,\ldots,an) \text{ <-- } C1,\ldots, Cm \quad (16)$$

is treated as part of the definition of a procedure named f; the head of this part of the definition of f is f(a1,...,an), and the body is the set

$$C1,\ldots,Cm, \quad (17)$$

where the C1,...,Cm are themselves procedure calls. Each invocation of a procedure which terminates is regarded as being either successful or unsuccessful. The sense of (16) is then that in order to successfully run f on an input vector whose pattern matches (a1,...,an) it suffices to successfully execute the calls in (17), using as values for variables which occur in a1,...,an any values obtained when the actual input vector to f was matched against (a1,...,an). Variables occurring in any of a1,...,an are the formal parameters of the procedure f; variables occurring in any of C1,...,Cm but not in a1,...,an are the local variables of the procedure. For example, the procedure call sum(s(s(0)),s(0),W) will not match the head of the first clause in (9), but will match the second clause using the substitution

$$\theta_1 = (X/s(s(0)),\ Y/0, W/s(Z)). \quad (18)$$

(In general, if θ is a substitution which replaces the variable X by the term a, the variable Y by the term b, etc., then θ will be written as $\theta = (X/a,\ Y/b,\ \ldots)$.) Consequently the sense of (9) is that in order to execute the given call of sum, it suffices to execute the call sum(s(s(0)),0,Z). A unit clause is regarded as a procedure definition with an empty body which immediately succeeds whenever the given input vector pattern is matched. Since the call sum(s(s(0)),0,Z) matches the first clause of (9) using the substitution

$$\theta_2 = (X/s(s(0)),\ Z/s(s(0))\ ), \quad (19)$$

this call immediately succeeds yielding the substitution $\theta 2$. The PROLOG processor returns from the original call that portion of the composed substitutions $\theta_1\theta_2$ applying to the variables appearing in the original call. So our original call returns the substitution

$$\theta 3 = (W / s(s(s(0)))\ ).$$

As noted above, any clause C (whether unit or not) is treated logically as universally quantified formula with all (apparently) free variables being quantified. Moreover, logic programming systems operate so as to preserve the standard semantics of the predicate calculus. Since one of the standard rules is "change of bound variable" (or α-conversion in the lambda calculus), this is preserved by logic programming systems. Two consequences of this are immediate. First, that the scope of (the implicit declaration of) an identifier is simply the clause in which it occurs. And second, that variable binding is static, rather than dynamic. Finally, it is obvious that the calling mechanism is call-by-name, since the system acts at all times as if it had actually substituted the computed values of variables for any occurrences of those variables.

PROLOG'S pattern-matching is carried out by means of the unification process. First some notation and terminology. Application of a substitution θ to a term t is indicated by $t\theta$. A substitution θ is a unifier for terms a and b if $a\theta$ and $b\theta$ are identical. If $\theta 1$ and $\theta 2$ are both substitutions, $\theta 1$ is more general than $\theta 2$ provided there exists a substitution ψ such that $a\theta 1\psi = a\theta 2$ for any a. There exist almost linear algorithms which given two terms a and b will compute a most general unifier for them if such exists, and indicate the non-existence otherwise. As the computed substitution $\theta 1$ in (18) indicates, the process treats the input terms a and b symmetrically. The pattern matching afforded by uniification is similar to that found in CONNIVER and QLISP using "open segment" variables (cf. Bobrow and Raphel[1974]). Since use of unification and terms allows abstract (or axiomatic) data type manipulation (see the example for trees below), and since logical variables can occur at any point in term structures, it can be said that pattern matching by means of unification is more powerful than that in QLISP or CONNIVER. Unification algorithms (for string representations of terms) can be easily programmed in SNOBOL (cf. griswold[1968]), so pattern matching in SNOBOL is as strong as that of unification. On the other hand, PROLOG systems provide primitive representations of strings, and using these, it is easy to program SNOBOL-type pattern matches. So from the point of view of pattern matching in strings, PROLOG and SNOBOL provide equivalent facilities. However, as is the case vis a vie CONNIVER and QLISP, PROLOG allows pattern matching at all levels in

user defined data types other than strings, and so in this important sense provides more powerful pattern matching facilities than SNOBOL.

The overall action of the PROLOG processor can now be described as follows. The user presents the processor with a database of procedure definitions together with a goal clause (14) to be solved. The processor transforms the goal clause, in a manner to be described, until it is reduced to the empty clause <-- , if such a reduction is at all possible; if and when the empty clause is produced, the computed substitution is returned to the user.

The processing of non-empty goal clauses (14) proceeds as follows. At each cycle, one Ci is selected. Any selection function will do (cf. Kowalski and Kuehner [1971] and Hill [1974]); PROLOG systems choose the left-most. The processor searches through its database seeking to match Ci against some procedure head by means of unification. If

$$D \quad <-- \quad E1,...EK \qquad (20)$$

is such a procedure and θ is the computed unifier of Ci and D, the previous goal (14) is replaced by the new goal

$$<--(C1,..,Ci-1,E1,..,Ek,Ci+1,..,Cn)\theta. \qquad (21)$$

Just as in ordinary procedural languages, infinite computations are possible. Note that it is allowable that the selected Ci be matchable against more than one procedure head in the database, thus permitting non-deterministic computations. PROLOG deals with this non-determinism by use of backtracking search strategies. If after a match of one part of the procedure definition (as above) the resulting computation (21) fails, then the processor attempts to find alternative matches with other parts of the procedure definition. In a word, in one match fails, it tries to find alternative matches. The order in which it attempts the matches is usually the order in which the parts of the procedure definition were entered into the database. Different logic programming systems allow the user varying degrees of control as to whether only one successful computation is sought or all are sought, etc.

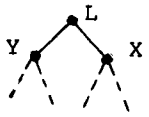
The extreme generality of the pattern-matching process provided by unification makes possible the use of any data type which can be abstractly defined in first-order logic. Thus logic programming permits both the style of programming with abstract data types and also the style of programming with input/output (verification) conditions (cf. van Emden [1976]).

IV. An Example

Consider the problem of insertion of labels in a binary search tree. The verification condition is:

$$\text{tree}(T) \;\&\; \text{label}(L) \;\&\; \text{insert}(T,L,T') \Rightarrow \text{tree}(T'), \qquad (22)$$

where tree(T) is to indicate that T is a binary search tree and insert(T,L,T') is to indicate that the tree T' results from the tree T by insertion of the label L. Let < be a previously defined linear relation on labels, let emp denote the empty tree, and let tr be a tree constructor: if X and Y are binary search trees and L is a label, tr(X,L,Y) is to be the binary search tree with X as its left subtree, L its label, and Y as its right subtree:



The natural way to give an inductive definition of tree is to specify that emp is a (binary search) tree, and that if X and Y are (binary search) trees and L is a label which follows all the labels in X with respect to < and preceeds all the labels in Y with respect to <, then tr(X,L,Y) is also a (binary search) tree. The logical definition of tree can now be given by (cf. Clark and Tarnlund [1977]):

$$\begin{aligned}&\text{tree}(\text{emp}) \;\&\\ &\quad X,Y,L\ [\text{tree}(\text{tr}(X,L,Y)) \iff \\ &\qquad \text{label}(L) \;\&\; \text{tree}(X) \;\&\; \text{tree}(Y) \;\&\\ &\qquad\quad M\ [\text{labelof}(X,M) \Rightarrow M < L] \;\&\\ &\qquad\quad M\ [\text{labelof}(Y,M) \Rightarrow L < M\ ]\ ].\end{aligned} \qquad (23)$$

Labelof is given the definition:

$$\begin{aligned}&\text{labelof}(\text{tr}(X,L,Y),M) \iff \\ &\quad (L = M) \;V\; \text{labelof}(X,M) \\ &\qquad\qquad V\ \text{labelof}(Y,M).\end{aligned} \qquad (24)$$

(Here V indicates logical disjunction: inclusive or). Using these as a basis for analysis, the insert predicate can be given the following formal definition:

$$\begin{aligned}&\text{insert}(\text{emp},L,\text{tr}(\text{emp},L,\text{emp})) \;\&\\ &X,Y,....[\ \text{insert}(\text{tr}(X,M,Y),L,\text{tr}(W,N,Z)) \\ &\quad \iff M = N \;\&\\ &\quad [[M = L \;\&\; X = W \;\&\; Y = Z] \\ &\quad\ V[L < M \;\&\; \text{insert}(X,L,W) \;\&\; Y = Z] \\ &\quad\ V[L > M \;\&\; X = W \;\&\; \text{insert}(Y,L,Z)\ ]]].\end{aligned} \qquad (25)$$

For to insert a label L in the empty tree is simply to build the tree tr(emp,L,emp). On the other hand, to insert L in the tree tr(X,M,Y) we proceed recursively as follows: if L < M, insert L in the subtree X, and if M < L, insert L in the subtree Y; if L = M, do nothing since L is already in the tree. From (25) we extract the following logic program, which is easily seen to be an analysis by cases:

```
insert(emp,L,tr(emp,L,emp)).
insert(tr(X,L,Y),L,tr(X,L,Y)).
insert(tr(X,M,Y),L,tr(W,M,Y)      (26)
   <-- L < M, insert(X,L,W).
insert(tr(X,M,Y),L,tr(X,M,Z))
   <-- M < L, insert(Y,L,Z).
```

Program (26) essentially consists of the "if" directions of the "if and only if" clauses in (25). (Cf. Clark and Tarnlund [1977] for proofs of termination and correctness for this program.) To further suggest the flavor of PROLOG programming, the addition of the following clauses to (26) results in a program to read a file LABELS of labels, construct the binary search tree for these labels, and print out a representation of the tree on the terminal:

```
process(FILE)<-- see(FILE),
                 doinserts(emp,TREE),
                 close(FILE),
                 display(TREE,1).

doinserts(T1,T2)<-- read(LABEL),
                    integer(LABEL),
                    insert(T1,LABEL,T3),
                    doinserts(T3,T2).
doinserts(T1,T1).

                                          (27)
display(emp,TB).
display(tr(U,L,V),TB)<--nl,
                 tab(TB),write(L),
                 TB1 is TB+3,
                 display(U,TB1),
                 display(V,TB1).
```

Here the procedures see and close handle the opening and closing of files on the current channel, integer is the obvious built-in predicate, nl starts a new line on the output channel, read and write do as their names imply, and tab outputs the indicated number of spaces. The "is" construction, which permits access to machine arithmetic, is explained below. The program is invoked by submitting the goal clause

```
<--process(LABELS).      (28)
```

to the processor. Given the input FILE consisting of the integers 3, 45, -44, 13, 134, -99, 33, and 435 (in that order), the program will produce the following output on the terminal:

```
3
   -44
      -99
   45
      13
         33
      134
         435
```

The term generated looks like:

```
tr(tr(tr(emp,-99,emp),-44,emp),3,tr(tr(e..
```

Both of these are representations of the tree which is most naturally displayed as:

```
                3
       -44                45
 -99                13            134
                       33              435
```

This last output was also generated by a (slightly more complicated) PROLOG program.

Tab could have been defined by the clauses:

```
tab(0).                                   (29)
tab(s(X)) <-- write(' '), tab(X).
```

Then the clauses (9) defining sum would have been added, and the expression

```
TB1   is   TB   +   3                     (30)
```

would have been replaced by

```
sum(TB, s(s(s(0))), TB1).                 (31)
```

The "is" construction in ((27) and (30) allows access to arithmetic on the underlying machine. On the left it expects a variable. On the right it expects an arithmetic expression which it evaluates and binds to the variable on its left.

On the other hand, (29) illustrates the use of unification and sequencing as control structures. On any given call to tab, unification is used to test whether the actual parameter is identical with 0. If it is, the call immediately succeeds. If it isn't, unification is used to test whether the actual argument is of the form s(X). If so, the unification process will have indicated the necessary value for X in order to achieve the match; this value is used in executing the body of the second clause of (29). If by chance the actual argument is not of the form s(X), the call fails.

Another (controversial) control mechanism present in the PROLOG systems is the cut determiner, symbolized by !. As a procedure, any call on this expression immediately succeeds. However, if later calls to other procedures fail, causing an attempt to backtrack, the backtracking cannot be propagated back across the cut. Thus crossing a cut definitely commits the program/processor to the computation produced thus far. To illustrate its power, here is a way of defining an operator which functions like negation:

```
not(P) <-- P, !, fail.
not(P).                                   (32)
```

The procedure fail simply always fails (by simply always being undefined) and does nothing else. Now suppose a given call of not has as a value for P a term (in this

context, an atomic formula!) which is in fact verifiable by the processor (i.e., running the procedure call to which P is bound will succeed). In attempting to run the first clause of (32), the processor first runs the procedure to which P is bound, which succeeds, next runs the cut procedure (!) which succeeds, and then runs the fail procedure, which fails. At this point the processor would normally unwind the previous computation in the clause, looking for decision points at which alternate computation paths could be followed. But this is prevented by the cut. Consequently its attempt to run the first clause of (32) fails; moreover, because it is prevented from backtracking, it cannot try any other clause defining not for the given procedure call of not. Consequently, this call of not(P) fails, as it ought to, since P succeeds. On the other hand, if P fails, the processor is allowed to backtrack out of the first clause since the failure of P occurs before the cut can be executed. But now the processor can run the second clause of (32), and this immediately succeeds. Hence the call of not(P) succeeds whenever P fails. It should be stressed that this negation is not true logical negation, but rather negation by failure: not(P) succeeds precisely when the processor fails to prove P. (None of the early PROLOG implementations incorporate a primitve negation, though all incorporate cut. IC-PROLOG (Clark and McCabe [1979]) does provide negation as a built-in primitive.) The difficulty with negation by failure (and a fortiori, the cut determiner) is that at first glance it appears to destroy the natural denotational semantics of logic, since its apparent definition is stricly process-oriented. However, Clark [1978] has shown that such use of negation by failure can be logically interpreted as follows. A negative goal :-not(P) is failure-derivable over a database D of procedures if and only if not(P) is a logical theorem in the theory D + D', where D' consists of all the converses of the clauses (implications) in D.

The cut determiner is analogous to the fence operator of SNOBOL (cf. Griswold[1968]). Both have the same control over their corresponding backtracking mechanisms, and both are quite primitve mechanisms for such control, as compared to the mechanisms of CONNIVER or truth-maintenance systems (Doyle[1978]). Yet careful use of these primitive mechanisms can have remarkable effects on the efficiency or even logical character of programs. The fundamental difference between these mechanisms and the more sophisticated backtracking algorithms of CONNIVER and truth maintenance systems is simply the extent of necessary explicit programmer concern for the backtracking.

V. The Database Interpretation of Logic

The action of a theorem prover can also be seen as the action of a database machine attempting to respond to queries over a database of assertions. This database is defined both by definite assertions and general laws. Consider a collection of unit clauses containing no variables:

```
cost(bolt23,34).     cost(bolt27,55).
                                          (33)
cost(bolt31,233).    cost(bolt41,09).
```

These assertions can alternatively be viewed as part of an array "cost":

cost	
bolt23	34
bolt27	55
bolt31	233
bolt41	9

(34)

Such arrays are of course just relations in extension, and so the collection of such definite unit clauses can be viewed as defining the extensional portion of a relational data base(cf. Codd[1970]). Clauses containing variables can be taken as intensionally specifying relations. For example, if comp is to define the "component of" relation, the following provide both extensional and intensional aspects of comp:

```
comp(bolt23,widget9).
comp(bolt23,widget11).
comp(bolt31,widget11).
comp(bolt41,widget9).
                                          (35)
comp(X,X).
comp(X,Z) <--
            comp(X,Y),
            subassembly(Y,Z).
```

A query over such a database is then simply a goal clause (14) with the logical interpretation (15). The response of the machine to the query is to attempt to show the query inconsistent with the database, using the same processing technique as indicated in the procedural interpretation. If the machine succeeds, the query is given an answer. If the machine provides simply one substitution as indication of success, the tuple of values corresponding to the variables in the query is the answer. If the machine provides a collection of such substitutions, the set of all the corresponding tuples is a relation which constitutes the answer. The retrieval procedure is the action of the theorem prover; in the case of PROLOG systems, it is SL-resolution. (cf. van Emden[1978]).

A number of interesting relations and retrieval operations can now be defined. The first will retrieve the list of components of a given entity. These definitions will make use of the built-in

list facilities of PROLOG. The list with elements A,B, and C is denoted [A,B,C]; the empty list is []. The list with head (first element) A and tail T is denoted [A,..T]. The relation append(L,M,N) which holds when N is the result of appending M to L is defined:

```
append([],M,M).                              (36)
append([H,..L],M,[H,..N])<--
append(L,M,N).
```

As expected, the query <--append([a,b],[c,d],N) will produce the result N = [a,b,c,d]. It is, however, interesting and useful to note that the symmetry and generality of unification make a great many other uses of this definition possible. For example, the query <--append([a,b],M,[a,b,c,d]) will produce the response M = [c,d], while the query <--append(L,[c,d],[a,b,c,d]) will produce the response L = [a,b]. Any successful computation of the query <--append(L,M,[a,b,c,d]) produces a pair (L,M) which is a partition of [a,b,c,d]; the set of all such successful computations yields the set of all partitions of this list. It is also interesting that the use of unification for matching allows the number of arguments of procedures to vary. The following clause may be added to the definition of append:

```
append(L1,L2,L3,N)<--append(L1,L2,M),
                     append(M,L3,N).         (37)
```

Then a query <--append([1],[2],[3],R) will produce the response R = [1,2,3].

The membership relation is defined in the natural recursive way:

```
on(X,[X,..T]).                               (38)
on(X,[Y,..T]) <-- on(X,T).
```

Using these, we can now retrieve the list of components of a given item:

```
getComponents(A,L) <--
                compList(A,[],L).
                                             (39)
compList(A,L,N) <-- comp(X,A),
                not(on(X,L)),
                append(L,[X],M),
                compList(A,M,N).
compList(A,L,L).
```

The relation compList(A,L,M) holds whenever L is a partial list of components of A, and M is the extension of L to the complete list of components of A. The definition given constructs these lists in the order the components are retrieved from the database. If the order of the lists is immaterial, the first clause of compList can be replaced by the more efficient:

```
compList(A,L,N) <-- comp(X,A),
                not(on(X,L)),                (40)
                compList(A,[X,..L],N).
```

Now the definition of cost (which here is simply material cost) can be extended to all items:

```
cost(A,C) <-- getComponents(A,L),
                  sumCosts(L,C).
                                             (41)
sumCosts([],0).
sumCosts([H,..T],S) <-- sumCosts(T,ST),
                        cost(H,CH),
                    S is ST+CH.
```

Various schemes have been studied for integrating the action of the logic processor with a database management system for retrieval of tuples from the extensional part of the database. Some approaches process a query against the intensional part of the database to construct a set of atomic queries which are then handed to the database management system. This style of approach is used by Kellog et al. [1978], Minker [1978], and Reiter [1978]. Another approach integrates the extensional database retrieval operations with the indexing methods of the logic processor so that intensional and extensional processing are integrated.

VI. Natural Language Processing

Computational approaches to natural language processing are of great current interest, especially in the database community. Indeed, Alain Colmerauer's principal motivation in working on the development of PROLOG was to devise a suitable vehicle for his investigations in natural language processing (cf. Colmerauer et al [1973] and Colmerauer [1978]). The present implementations of PROLOG incorporate grammar rules in their syntax; these make possible a quite direct expression of Colmerauer's metamorphosis grammars. Here we will sketch an alternate approach inspired by the work of Richard Montague [1970]. As with Colmerauer's work, PROLOG provides an especially suitable vehicle for expressing the requirements of Montague grammars.

A system which allows natural language queries to a database would be expected to process the natural language into an internal logical form, respond to this form with another logical form, and then process its response back into natural language, as the following top-level relations suggest:

```
cycle <-- input(Expression),                 (42)
      internalForm(Expression,Formula1),
      respond(Formula1,Formula2),
      internalForm(Response,Formula2),
      output(Response),
      cycle.
```

For simplicity, assume that input produces

a list consisting of the words, in order, in the natural language expression. Then the following clauses begin the definition of internalForm (which we will abbreviate as iF):

```
iF(E, [NP,VP]) <-- append(NP,VP,E),
     nounPhrase(NP),
     intransitiveVerbPhrase(VP).
                                        (43)
iF(E,[NP,VP,OBJ]) <--
append(NP,VP1,OBJ,ADV,E),
     nounPhrase(NP), nounPhrase(OBJ),
     append(VP1,ADV,VP),
     transitiveVerbPhrase(VP1),
     adverb(ADV).

iF(E, [conj, F1,F2]) <--
     append(E1, [and], E2, E),
     iF(E1, F1), iF(E2, F2).

iF(E, [negation F]) <--
     append(E1, [not], E2, E),
     append(E1,E2,E3), iF(E3,F).
              .
              .
              .
```

Notice that the backtracking facilities of PROLOG provide mechanisms for interaction between the syntactic and semantic components in (42). For should response fail (presumably because it cannot interpret Formula1), backtracking will lead internalForm to search for an alternate construal of the input expression. The same consideration can extend to response itself. After producing a response Formula2 in (42), the final action of the program is to recursively call itself. Should the user input some sentence such as "I don't understand(or accept) that," the program for response could react to the logical form of this by failing, which would eventually backtrack into an attempt to find a different response to the original input (and perhaps to reconstrue the original input).

VII. Final Remarks

The point of view of logic programming is to decompose algorithms into a logic component and a control component. For a wide range of normal programming problems, the logic component can be specified in Horn clause logic and will run correctly under any choice of control component. A common procedure is to specify the problem in full first-order logic, and then derive the clausal program by standard logical transformations (cf. Clark and Tarnlund [1977] and Nilsson [1971], Chapter 6). On the other hand, in his forthcoming book [1979], Kowalski amply demonstrates that the clausal form is often an extremely natural medium for expression of problem solutions. A note about efficiency is in order here. The naturual methodology when working in PROLOG is to first express the problem (often its specifications) in as clear and direct form as possible in Horn clauses. Often these will run quite efficiently (cf. Warren et al. where the PROLOG's efficiency is favorably contrasted with that of LISP). When this is not the case, one then begins transforming the program into an equivilant, but more efficient one, by additions or alterations of the control structure (sequencing and use of the cut determiner). Programs of quite acceptable efficiency can be obtained in this way. Moreover, working in PROLOG has the advantange that in the transition from initial specification to final (efficient) program, one remains in the same language, avoiding the problems of transition from logical specifications to procedurally-oriented languages.

A wide variety of semantics are available for logic programs. Any logic program automatically carries with it the usual Tarskian semantics. It is this semantics which is involved in both intuitive and formal specifications of programs (both logic programs and those of other languages). This standard semantics was connected with the fixed-point semantics in van Emden and Kowalski [1976]. Besides the procedural and database interpretations discussed above, it is possible to provide a flowchart semantics (cf. Clark and van Emden[1979]), and a parallel-process semantics (cf. van Emden and Lucena [1979]) in which a goal statement is interpreted as a network of stacks interconnected by channels.

Some natural expressions of problems require extensions of Horn clause logic, as illustrated above. Negation is valuable (and problematic). Current research is studying both extensions to the logic components (to include more of full first-order logic and to study the merger of Horn clause logic and its metalanguage) and to the control component (especially to study the use of co-routining and parallel execution). Related work is also underway investigating the incorporation of logic programming constructs into existing high-level languages, notably LISP.

Bibliography

Bobrow, Daniel G., and Raphel, Bertram
[1974] New programming languages for artificial intelligence research, Computing Surveys, 6, pp 153-174.

Clark, Keith
[1978] Negation as failure, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), New York: Plenum Press, 293-322.

Clark, Keith L. and van Emden, Maarten H.
[1979] Consequence verification of flowcharts, Report CS-79-23, Department of

Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

Clark, Keith L. and McCabe, Frank
[1979] IC-PROLOG, Proc. AISB.

Clark, Keith L. and Tarnlund, Sten-åke
[1977] A First-order theory of data and programs, in Information Processing 77, B. Gilchrist (ed.), Amsterdam, North Holland, 939-944.

Codd, E. F.
[1970] A relational model of data for large shared data banks, Comm. ACM ,13, pp. 377 - 387.

Colmerauer, Alain
[1978] Metamorphosis grammars, in Natural Language Communication with Computers, Leonard Bloc (ed.), Lecture Notes in Computer Science #63, Berlin: Springer-Verlag, 133-189.

Colmerauer, A., Kanoui, H., Pasero, R., Roussel, P.
[1973] Un Systeme de Comunication Homme-machine en Francaies. Rapport Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy.

Doyle, Jon
[1978] Truth maintenance systems for problem solving, Report AI-TR-419, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Maarten H. van Emden
[1976] Verification conditions as programs, in Automata, Languages, and Programming, S. Michaelson and R. Milner (eds.), Edinburgh: Edinburgh University Press.
[1977] Programming in resolution logic, in Machine Intelligence 8, E.W. Elcock and D. Michie (eds.), Edinburgh: Edinburgh University Press.
[1978] Computation and Deductive Information Retrieval, in Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.), Amsterdam: North Holland.

van Emden, Maarten H and Kowalski, Robert A.
[1976] The semantics of predicate logic as programming language, J. Assoc. Comp. Mach., 23, 733 - 742.

van Emden, Maarten H. and de Lucena, G.J.
[1979] Predicate logic as a language for parallel programming, Faculty of Mathematics Report CS-79-15, University of Waterloo.

Green, Cordell
[1969] Theorem-proving by resolution as a basis for question answering systems, in Machine Intelligence 4, B. Meltzer and D. Michie (eds.), Edinburgh: Edinburgh University Press.

Griswold, R.E., Poage, J.F., and Polonsky, I.P.
[1968] The SNOBOL4 Programming Language, Englewood Cliffs, New Jersey: Prentice-Hall.

Hill, Robert
[1974] LUSH-resolution and its completeness, DCL Memo 78, Department of Computational Logic, University of Edinburgh.

Kowalski, Robert
[1974] Predicate logic as a programming language, Proc. IFIP 74, Amsterdam: North Holland, 556-574.
[1978] Logic for data description, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), New York: Plenum Press, 77 - 103.
[1979] Logic for Problem Solving, New York: Elsevier- North Holland.

Kowalski, Robert, and Kuehner, Donald
[1971] Linear resolution with selection function, Artificial Intelligence 2, 227-260.

Kellog, Charles, Klahr, Philip, and Travis, Larry
[1978] Deductive planning and pathfinding for relational data bases, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), New York: Plenum Press, 179 - 200.

Montague, Richard
[1970] English as a formal language, in Linguaggi nella Societa e nella Tecnica, B. Visentini et al. (eds.), Milan: Edizioni di Comunita, 189 - 224; reprinted in: Formal Philosophy, Selected Papers of Richard Montague, R.H. Thomason (ed.), New Haven: Yale University Press, 1974, pp. 188 - 221.

Minker, Jack
[1978] An experimental relational data base system based on logic, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), New York: Plenum Press, 107 -147.

Nilsson, Nils J.
[1971] Problem-Solving Methods in Artificial Intelligence, New York: McGraw-Hill.

Reiter, Raymond
[1978] Deductive question-answering on relational data bases, in Logic and Data Bases, H. Gallaire and J. Minker (eds.), New York: Plenum Press, 149 - 177.

Roberts, Grant M.
[1977] An implementation of PROLOG, M.Sc. Thesis, Dept. of Computer Science, University of Waterloo.

Roussel, P.
[1975] PROLOG: Manuel e Reference et d'Utilisation. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Universite d'Aix-Marseille.

Robinson, John Alan
[1965] A machine-oriented logic based on the resolution principle, J. Assoc. Comput. Mach. 12, 23-41.
[1979] Logic: Form and Function, Edinburgh: Edinburgh University Press.

Warren, David H. D., Pereira, Fernando, and Pereira, Luis M.
[1977] PROLOG: The language and its implementation compared with LISP, in Proc. Symp. on Artifical Intelligence and Programming Languages, Special Issue: SIGPLAN Notices, vol. 12, no. 8 (SIGART Newsletter, no. 64), 109-115.