CHAPTER 5

The Procedural Interpretation of Horn Clauses

A Horn clause

$$B \leftarrow A_1,\ldots,A_m \qquad m \geq 0$$

is interpreted as a procedure whose body $(A_1,\ldots,A_m)$ is a set of procedure calls A_i. Top-down derivations are computations. Generation of a new goal statement from an old one by matching the selected procedure call with the name B of a procedure

$$B \leftarrow A_1,\ldots,A_m$$

is procedure invocation.

A logic program consists of a set of Horn clause procedures and is activated by an initial goal statement.

Conventional programs mix the logic of the information used in solving problems together with the control over the manner in which the information is used. Logic programs are more abstract. They control neither the order in which different procedures are invoked when several match a given procedure call, nor the order in which procedure calls are executed when several belong to the same goal statement.

Logic programs express only the logic of problem-solving methods. They are easier to understand, easier to verify and easier to change. They are especially congenial to inexperienced programmers and database users who do not want to become involved with the details of controlling the program's behaviour.

The first logic programming system, called PROLOG [Colmerauer et al 1973], [Roussel 1975] based on the procedural interpretation of Horn clauses [Kowalski 1974] was designed and implemented in 1972. A PROLOG compiler written in PROLOG for the PDP10 was implemented at the University of Edinburgh by Warren, Pereira and Pereira [1977]. They showed that the PROLOG compiler executes LISP-like logic programs as efficiently as compiled LISP [McCarthy et al 1962].

Terms as data structures

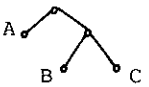
Data in logic programs can be represented by means of terms or relations. The use of terms as data structures gives Horn clause programs many of the characteristics of a list-processing language like LISP. More generally, they function as recursive data structures of the kind

advocated by Hoare [1972]. The use of relations in logic programs, on the other hand, is like the representation of data by relations in database formalisms [Codd 1970]. Relations are also like tables and arrays in conventional programming languages. They will be discussed in more detail later in the chapter.

As in LISP, binary trees can be represented by means of a binary function symbol:

cons(x,y) names the tree 

which has the subtree x immediately to the left of the root node and the subtree y immediately to the right. Thus the term

cons(A,cons(B,C)) names the tree 

and the program

```
Tips(x,1) <- Label(x)
Tips(cons(x,y), w) <- Tips(x,u), Tips(y,v), u+v = w
```

defines the relationship Tips(x,y) which holds when y is the number of tips in the binary tree x. Label(x) holds when x is a label:

```
Label(A) <-
Label(B) <-
Label(C) <-
```

for example. The goal statement

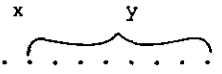
```
<- Tips(cons(A,cons(B,C)), y)
```

expresses the goal of computing the number of tips in the tree pictured above. The term cons(A,cons(B,C)) names the input and the variable y names the output. The top-down solution

```
<- Tips(cons(A,cons(B,C)), y)
<- Tips(A,u), Tips(cons(B,C), v), u+v = y
<- Label(A,u), Tips(cons(B,C), v), u+v = y
u=1
<- Tips(cons(B,C), v), 1+v = y
<- Tips(B,u'), Tips(C,v'), u'+v' = v, 1+v = y
u'=1
v'=1
<- Label(B), Label(C), 1+1 = v, 1+v = y
v =2
<- 1+2 = y
y =3
□
```

is a computation of the output y = 3. The search space contains only the computation.

Lists can be regarded, as in LISP, as a special kind of binary tree. The term cons(x,y) names the list



which has first element x followed by the list y. The constant symbol nil names the empty list. Thus the term cons(A,cons(B,cons(C,nil))) names the list A,B,C and the program

```
Item(cons(x,y), 1, x) <-
Item(cons(x,y), u, z) <- Item(y,v,z), v+1 = u
```

defines the relationship Item(x,y,z) which holds when the y-th element of the list x is z. Notice that the term cons(A,B) does not name the list A,B because B is not a list. The list consisting of B alone is named by cons(B,nil) and therefore the list A,B is named by cons(A,cons(B,nil)).

Programs may be easier to read if infix notation is used for function symbols and conventions are used for suppressing parentheses. It is especially convenient to use an infix function symbol "." for lists

x.y stands for cons(x,y)

and to reduce parentheses by letting

x.y.z stand for cons(x,cons(y,z)).

Thus the list A,B,C can be represented by the term

A.B.C.nil .

Facilities for defining infix function symbols and for reducing parentheses are provided in PROLOG. The programmer can further reduce parentheses by declaring precedence relations among function symbols. Thus by declaring that the infix function symbol & binds more closely than the infix function symbol ⊃ , the term

p & q ⊃ r & s

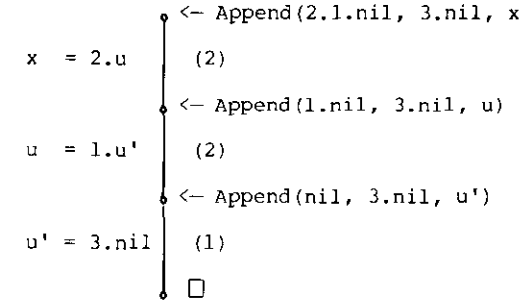
can be written instead of

(p & q) ⊃ (r & s).

Computation by successive approximation to output

Horn clause procedures transmit output throughout computation. Partial outputs accumulate and determine successive approximations to the final output. The approximations are generated whether or not the computation eventually succeeds.

The figure below illustrates the computation by successive approximation of the list which results from appending 3.nil to 2.1.nil .



(1) Append(nil,x,x) <-
(2) Append(x.y, z, x.u) <- Append(y,z,u)

Clause (1) states that appending any list x to the empty list produces the list x. Clause (2) states that appending a list z to a non-empty list x.y produces a list x.u with the same first element and with a remainder u which is the result of appending z to y.

The successive steps of the computation determine successive approximations to the output

x = 2.u
x = 2.1.u'
x = 2.1.3.nil .

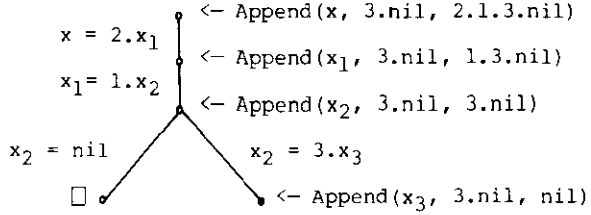
In general, the output of a computation can be regarded as the collection of all output components of matching substitutions performed in the computation. The output can be compactified, as in the example above, by applying output components lower in the refutation to the terms of output components higher in the refutation.

The variation of input-output parameters

The distinction between the input and output parameters of a procedure depends upon the context in which the procedure is invoked. Any subset of the procedure's parameters can be given as input. The remaining parameters are then computed as output.

The following computation illustrates the use of Append to compute the list x which produces 2.1.3.nil when 3.nil is appended to it. The search space contains, in addition to the successfully terminating computation,

only one other step, which fails because no procedure matches its procedure call.



The ability to execute the same procedure with various patterns of input and output is an important feature of logic programs. It implies, for example, that the same procedures which compute derivatives of functions can also be used to compute integrals [Bergman and Kanoui 1973]. Procedures which verify that a given program meets given specifications can also be used to generate programs from specifications [Moss 1977].

Non-determinism$_1$: several procedures match a procedure call

Compared with normal programs, Horn clause programs executed top-down are non-deterministic in two main senses: When several procedures match a given procedure call, the search strategy by means of which the alternative procedures are tried is not determined$_1$. When several procedure calls need to be executed in a single goal statement, the order of execution is not determined$_2$.

In the first case, alternative procedures may compute alternative outputs. If only one output is needed, it is not determined$_1$ which output will be found. If all outputs are required, it is not determined$_1$ in which order they will be generated.

A procedure, which is deterministic$_1$ for one pattern of input and output parameters may be non-deterministic$_1$ for a different pattern. The Append procedure, for example, is non-deterministic$_1$ when it is used to partition a given list into two parts as in the problem

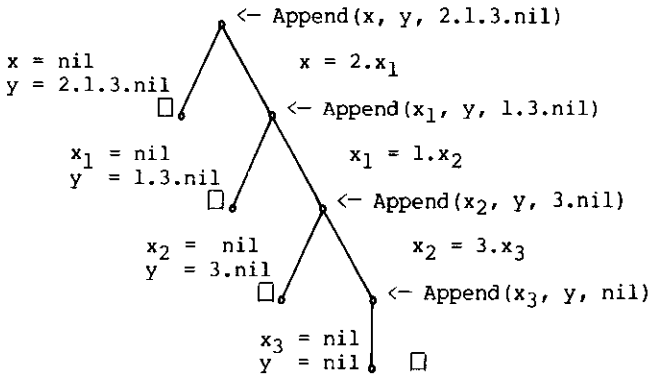
<- Append(x, y, 2.1.3.nil).

The search space of all computations is illustrated below. Notice the economy which is obtained by structuring the search space as a tree. The two different partitions

x = 2.1.nil, y = 3.nil and
x = 2.1.3.nil, y = nil

for example, are both obtained from the single initial approximation

x = 2.1.x_2 .



Sequential search regarded as iteration

The ability to specify repeated execution of the same command is an essential feature of all programming languages. Such repetition, also called iteration* , can be accomplished by executing recursive Horn clause procedures. It can also be achieved by using backtracking to search a space of alternativs. The definition of grandparent is a simple example. Suppose that we are given data about individuals in the parenthood relationship

```
Parent(Zeus,Ares)       <-
Parent(Hera,Ares)       <-
Parent(Ares,Harmonia)   <-
Parent(Semele,Dionysus) <-
Parent(Zeus,Dionysus)   <-
     etc.
```

and the problem is to show that Zeus is a grandparent of Harmonia

```
<- Grandparent(Zeus,Harmonia)
```

using the definition of grandparent

```
Grandparent(x,y) <- Parent(x,z), Parent(z,y).
```

In a conventional programming language, the programmer would have to specify both how the data in the parenthood relationship is stored and how it is retrieved. In a logic program, the same decisions are taken by the program executor instead. In either case, the simplest strategy is to store and retrieve the data sequentially. The parenthood relationship might be stored sequentially, either in a two-dimensional array or in a

---

*Some of the discussion in the next few sections refers to features of conventional programming languages. The reader who is not familiar with such languages can ignore these sections without disadvantage.

linked list. The sequential retrieval strategy is an iteration, consisting of a double loop, one nested inside the other. To show Zeus is a grandparent of Harmonia, the outer loop searches for a child z of Zeus and the inner loop tests whether z is a parent of Harmonia. The iterative algorithm which has to be specified by the programmer in a conventional programming language is identical in this case to the behaviour determined$_1$ by the backtracking strategy for executing non-deterministic$_1$ programs.

In other cases, as when the Append procedure is used to partition lists, backtracking is more general than iteration. In general, whereas iteration searches a tree whose depth is determined by the number of loops which are nested, backtracking searches an arbitrarily deep tree of alternatives,

The suitability of a search strategy depends upon the structure in which the data is stored. Iteration, regarded as sequential search, is suitable for data stored sequentially. Other search strategies are appropriate for such data structures as hash tables, binary trees or semantic networks. Fishman and Minker [1975] for example, store data in a manner which facilitates parallel search, whereas Deliyanni and Kowalski [1979] propose a path-following strategy for retrieving data stored in semantic networks.

"Don't know" versus "don't care" non-determinism$_1$

Non-determinism$_1$ does not always entail the need to search for a solution. The definition of Max(x,y,z) (the maximum of x and y is z) is an example.

```
Max(x,y,x) <- x ≥ y
Max(x,y,y) <- y ≥ x
```

Both procedures apply when x and y are identical, as in the case

```
<- Max(3,3,z).
```

Searching for a solution, which is unavoidable in the general case, creates redundancy when it is unnecessary. Backtracking is redundant, for example, when it is applied to the goal statement

```
<- Max(3,3,z), Even(z)
```

and the procedure calls are executed in the order in which they are written. The second procedure call Even(z), which succeeds when z is even, fails no matter how the first procedure call is executed. Backtracking after the first failure, to try a different way of executing the first procedure call, is both unnecessary and redundant.

Searching can be restricted in general whenever the output variables of a procedure call are a function of the input - for example, when the variable y is a function of x in the relation F(x,y) and x is given as input. Backtracking can be suppressed if the first solution of the goal F(A,y) fails to solve the second goal G(y) in the goal statement

```
<- F(A,y), G(y).
```

When searching for a solution is unnecessary, then the program executor "doesn't care" which solution is generated nor how it is obtained. Otherwise, searching is unavoidable when the executor "doesn't know". Don't care non-determinism$_1$ is a dominant feature of Dijkstra's language of guarded commands [1976]. The use of don't care non-determinism$_1$ to restrict search is a form of intelligent backtracking.

Non-determinism$_1$ can have both don't know and don't care characteristics. The path-finding problem is an example. Given the problem of finding a path from A to N

```
<- Go(A,N)
```

for example, the program executor doesn't care which path is found but normally doesn't know which procedures to apply in order to find it. Searching is necessary to find one path but is unnecessary and redundant thereafter.

The path-finding problem is a special case of the general situation in which a procedure call shares no variables with other calls in the same goal statement. Any non-determinism$_1$ involved in executing the procedure call matters only until the first solution is found. The second procedure call in the body of the procedure

```
Happy(Bob) <- Teaches(Bob,x), Attends(y,x)

Bob is happy if he teaches a course
which someone attends.
```

is an example. If it is executed after the other procedure call, then its only variable y occurs in no other procedure call and it suffices to find only a single solution.

The property that a procedure call contains no variables or that all its variables occur in no other procedure call is a syntactic property which the program executor can easily recognise without the aid of the programmer. The situation, however, in which search can be restricted because a procedure call computes the value of a function is undecidable in principle. It is easier for the programmer to convey such information to the program executor as a comment about the program, than it is for the executor to discover the fact for itself.

Don't care non-determinism$_1$ provides a way of adding extra information to a program without enlarging the search space and even reducing its size. The new information may solve a problem more directly than the original procedures, and if the non-determinism$_1$ doesn't matter then the original procedures can be ignored.

Non-determinism$_2$: The scheduling of procedure calls

In conventional programming languages the program controls the scheduling of procedure calls - usually in some fixed sequence, but sometimes timesharing among them or executing them in parallel. In logic,

however, the body of a procedure specifies only the collection of procedure calls. The manner in which they are executed is determined$_2$ not by the program but by the execution mechanism. Different strategies for scheduling procedure calls affect the efficiency of execution but do not affect the meaning as determined by the relations which are computed.

The definition of sorted lists is a simple example. Assume that the definition of the $\leq$ relation is already given.

```
Sort(x,y)      holds when y is a sorted version of list x,
Perm(x,y)                 y is a permutation of x,
Delete(x,y,z)             z results from deleting any one
                            occurrence of x from y.
```

```
S1   Sort(x,y)       <- Perm(x,y), Ord(y)
S2   Perm(nil,nil)   <-
S3   Perm(z, x.y)    <- Delete(x,z,z'), Perm(z',y)
S4   Delete(x, x.y, y)    <-
S5   Delete(x, y.z, y.u)  <- Delete(x,z,u)
S6   Ord(nil)     <-
S7   Ord(x.nil)   <-
S8   Ord(x.y.z)   <- x ≤ y, Ord(y.z)
```

In principle, the procedure calls in the body of procedure S1 can be executed in any sequence. Given a list l, to generate a sorted version y of l, it is possible firstly to execute the procedure call Ord(y), generating an ordered list y, and then to execute Perm(l,y), testing whether y is a permutation of l. If the test fails, other ordered lists can be generated until the test succeeds. It is more effective, of course, to execute procedure calls in the opposite sequence - first generating permutations of x and then testing whether they are ordered. But no matter in which sequence procedure calls are executed and no matter what the cost in terms of efficiency, the result in terms of the input-output relation computed is the same.

Effective scheduling of procedure calls depends upon the pattern of input and output. Generally it is more efficient to execute a procedure call which contains the input in preference to one which does not. Thus, given the problem

```
<- Sort(l,y)
<- Perm(l,y), Ord(y)
```

of finding a sorted version y of an input list l it is better to select for execution the procedure call Perm(l,y) which contains the input than it is to select Ord(y) which does not. If both l_1 and l_2 are given and the problem

```
<- Sort(l1,l2)
<- Perm(l1,l2), Ord(l2)
```

is to test that l_2 is a sorted version of l_1, then both procedure calls contain the input and it does not affect efficiency which procedure call is executed first. Moreover, since the two procedure calls do not share

variables and since they are equally good candidates for execution, they can be executed together - either timesharing between them if only one processor is available or executing them in parallel if several can be used.

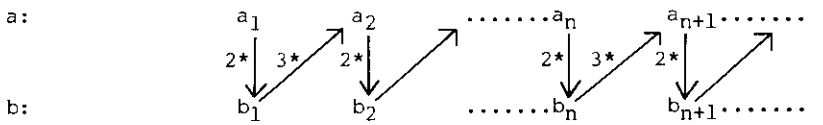
In general, it is advantageous to execute procedure calls as soon as sufficient input is available. Given procedures (S1-8) and the goal of sorting the list 2.1.3.nil, generating permutations before testing them for orderedness, the test for orderedness can be initiated just as effectively when the first two elements of the permutation have been determined as it can when the entire permutation has been generated. Executing procedure calls as soon as possible has the advantage that failure can be detected as soon as possible. The figure below illustrates the effectiveness of eagerly executing the orderedness test to reject in one step all permutations which have first element 2 and second element 1.

```
                    <- Sort(2.1.3.nil, y)

                    <- Perm(2.1.3.nil, y), Ord(y)
y  = x.y'
                    <- Delete(x, 2.1.3.nil, z'), Perm(z', y'), Ord(x.y')
x  = 2
z' = 1.3.nil
                    <- Perm(1.3.nil, y'), Ord(2.y')
y' = x'.y"
                    <- Delete(x', 1.3.nil, z"), Perm(z", y"), Ord(2.x'.y")
x' = 1
z" = 3.nil
                    <- Perm(3.nil, y"), Ord(2.1.y")

                    <- Perm(3.nil, y"), 2<1,  Ord(1.y")
```

The behaviour of the admissible pairs problem is a more dramatic example, which is intolerably non-deterministic, if procedure calls are executed last-in-first-out. A pair (a,b) of lists of numbers is admissible if the two lists have the same length and for every i

if a_i is the i-th element of a and
b_i is the i-th element of b, then
$b_i = 2*a_i$ and
$a_{i+1} = 3*b_i$.

Pictorially:



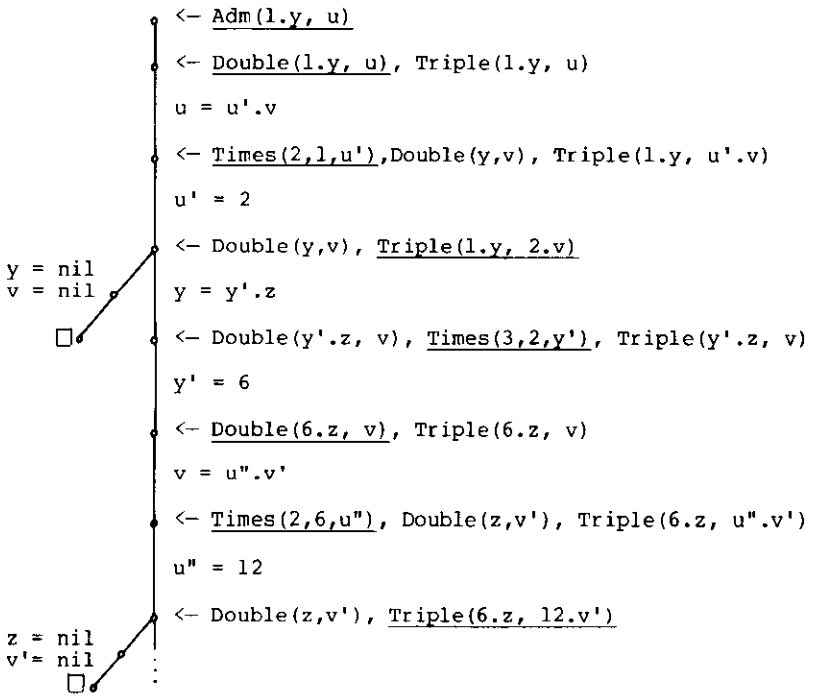
The following clauses, in which lists are represented by means of terms, define the desired relationship:

```
Adm(x,y) <- Double(x,y), Triple(x,y)
Double(nil,nil)  <-
Double(x.y, u.v) <- Times(2,x,u), Double(y,v)
Triple(x.nil, u.nil) <-
Triple(x.y.z, u.v)   <- Times(3,u,y), Triple(y.z, v)
```

Consider the problem of generating an admissible pair of lists whose first list begins with with the number 1:

```
<- Adm(1.y, w)
```

The program is intolerably non-deterministic$_1$ if procedure calls are executed last-in-first-out, completing the execution of one call before initiating another. It becomes virtually deterministic$_1$, however, if procedure calls are executed as soon as sufficient input is available. The two procedure calls behave as co-operating sequential processes. As soon as either one of the two processes, Double or Triple, has enough information about its input it runs until it needs more. By that time it has produced enough output for the other process to resume execution.



Coroutines, which cooperatively produce and consume data, can be written in programming languages such as SIMULA. Such coroutines, however, are syntactically and semantically different from normal procedures. However, more recent schemes, in which procedures are called by need [Henderson and Morris 1976] and the activation of processes is

controlled by the flow of data [Kahn 1974] [Friedman and Wise 1978], resemble the execution of procedures in logic programs. The strategy for executing procedure calls is not determined by the program but by the program executor.

Bottom-up execution of programs

The procedural interpretation of Horn clauses is primarily the top-down interpretation. It is sometimes possible, however, to give a procedural interpretation to bottom-up inference. Although it is generally more efficient for computers to interpret Horn clauses top-down, it is often more natural for people to understand them bottom-up. Moreover, it is sometimes more efficient to execute programs bottom-up rather than top-down.

A student of mathematics, for example, is more likely to understand the recursive definition of factorial

```
The factorial of 0 is 1 <-
The factorial of x is u <- y+1 = x,
                           the factorial of y is v,
                           x*v = u
```

bottom-up, as determining 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.
```

than he is to understand it top-down, as reducing goals to subgoals. In this example, bottom-up derivation of factorials has a computational flavour. It behaves as an iterative computation which accumulates factorials of successively larger numbers until it derives the one which is desired.

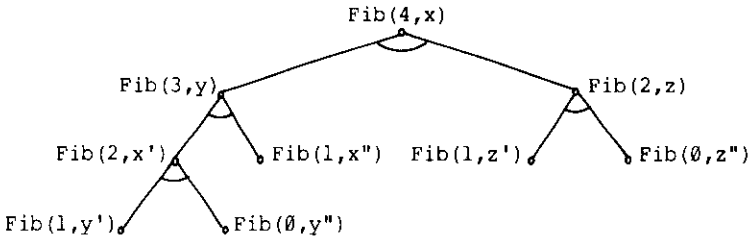
The definition of Fibonacci number can be executed more efficiently bottom-up than top-down.

```
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+z = 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 an and-tree whose nodes are procedure calls, the

number of which is an exponential function of u. The problem of computing the Fibonacci of 4, for example, determines a tree, which ignoring additions contains a total of 9 goals and subgoals.



Here Fib(u,x) means the u-th Fibonacci number is x. Executing 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 for the Fibonacci of u executed bottom-up is a linear function of u.

In this example, bottom-up execution is also potentially 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 therefore 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 (Chapter 8).

Notice that the efficiency of top-down execution approaches that of bottom-up execution if similar procedure calls (i.e. the u-th Fibonacci number is z and the u-th Fibonacci number is z') are executed only once. Such top-down execution is an extension of Earley's parsing algorithm [Earley 1970] as described by Warren [unpublished].

Iteration in conventional programming languages has three different interpretations in logic programs. The classical interpretation regards iteration as a special case of top-down execution of recursive definitions. The iteration

To do P, repeat Q until R

for example, can be expressed in the form

P(x) <- R(x)
P(x) <- Q(x,x'), P(x')

where x is an input parameter which controls the number of iterations through the loop and R(x) and Q(x,x') hold for distinct x. The recursion is a form of iteration if Q(x,x') is executed before P(x'). Consequently

each new subgoal P(x') can replace the previous subgoal P(x). Execution, therefore, requires only a constant amount of storage for the current subgoal.

The interpretation of iteration as top-down execution of certain forms of recursive definitions is the only interpretation of iteration possible in the conventional model of computation by recursion. In logic programs, however, it is also possible to regard iteration either as sequential search through a space of alternative responses to a procedure call or as bottom-up execution of recursive definitions.

The pragmatic content of logic programs

It is a common mistake to treat logic simply as a specification language whose statements have semantic content without pragmatic value. Such an attitude is self-fulfilling. To use logic while ignoring its pragmatic aspects is to make information potentially unusable.

Two different statements can express the same information and therefore have the same meaning. But one might be useful for solving problems and the other one useless.

The sorting problem, studied by van Emden [1977], is a good example of the pragmatics of logic. The simple program (S1-8) for sorting lists

```
Sort(x,y) <- Perm(x,y), Ord(y)
```

is a good specification, but a useless program. Even the scheduling of procedure calls which uses Ord(y) to monitor the partial output of Perm(x,y) is hopelessly inefficient (taking time 2^n in order to sort a list of length n). In contrast, even simple sequential execution of procedure calls produces an efficient algorithm, Quicksort [Hoare 1961], taking time n*log(n), from the program:

```
Sort*(nil,nil) <-
Sort*(x.y, z)  <- Partition(x,y,u,v), Sort*(u,u'),
                  Sort*(v,v'), Append(u', x.v', z).
```

Here it is intended that Partition(x,y,u,v) holds when u is the list of all members of y which are less than or equal to x and v is the list of all members of y which are greater than x.

Sort and Sort* are equivalent in the sense that Sort(s,t) and Sort*(s,t) hold for the same pairs of terms s, t. Sort is useful as a specification of sortedness but useless for efficiently sorting lists. Sort* is efficient but less obviously correct.

In general, a given problem can be expressed in many different ways. The two representations of the path-finding problem (one using the predicate Go(x), the other using the predicate Go*(x,y)) can be generalised to other problems. Even the definition of factorial can be represented in two ways. The previous definition corresponds to the one-place-predicate formulation of path-finding. The definition below corresponds to the two-place-predicate formulation.

```
Fact*(x,y,u,v)
```

expresses that the factorial of x is y if the factorial of u is v.

```
Fact*(u,v,u,v) <-

Fact*(x,y,u,v) <- u+1 = u', u'*v = v', Fact*(x,y,u',v')
```

To find the factorial of an integer represented by a term t, a single goal statement incorporates not only the goal but also the information that the factorial of ∅ is 1.

```
<- Fact*(t,y,∅,1)
```

The new formulation of factorial executed top-down behaves in the same iterative manner as the original formulation executed in a mixed top-down, bottom-up fashion. The old formulation is more obviously correct, whereas the new formulation is easier to execute efficiently with more limited problem-solving facilities.

Separation of data structures

For a well-structured program, it is desirable that the data structures be separated from the procedures which interrogate and manipulate them. Separation of data structures from procedures means that the representation of the data can be altered without altering the higher-level procedures. It is easier to improve efficiency, therefore, by replacing an inefficient data structure with a more efficient one. In a large complex program the information which needs to be supplied by the data structures is often completely identified only in the final stages of the program design. By separating data structures from procedures, it is possible to write the higher levels of the program before the data structures have been determined.

Data storage and retrieval are automatically separated from procedures when data is represented by relations, as in the family relationships example. When data is represented instead by terms it is the programmer's responsibility to separate them in the program.

The arch recognition problem is a simple example. The previous formulation which mixes procedures and data structures can be replaced by one which separates them. Mention of the data structures in the top-level procedures can be replaced by procedure calls which access, compute or construct the data.

```
Arch(x)  <- Block(v), Tower(u), Tower(w), On(v,u),
            On(v,w), Left(x,u), Right(x,w), Top(x,v)

Tower(x) <- Block(x)

Tower(x) <- Block(u), Tower(v), On(u,v),
            Top(x,u), Bottom(x,v)

On(x,y)  <- Top(y,u), On(x,u)
```

Here the Top, Left, Right and Bottom relations define the interface between the procedures and the data structures. It is intended that

```
Top(x,y)      holds when the top of x is y,
Left(x,y)                the left  subtower of arch x is y,
Right(x,y)               the right subtower of arch x is y,
Bottom(x,y)              the bottom of tower x is y.
```

The data structures can be defined separately by defining their interface with the top-level procedures:

```
Top(a(u,v,w), v)   <-
Top(t(u,v), u)     <-
Left(a(u,v,w), u)  <-
Right(a(u,v,w), w) <-
Bottom(t(u,v), v)  <-
```

In this case the interfacing procedures are defined simply by means of assertions. But in other cases they might be defined by more general kinds of procedures.

Comparing the two formulations of the arches program, we notice another advantage of separating procedures and data structures: with infix notation for predicate symbols and with well chosen names for the interfacing procedures, data-structure-independent programs are virtually self-documenting. For conventional programs which mix data structures and procedures, the programmer needs to provide documentation which explains the data structures and is external to the program. For well-structured programs which separate procedures and data structures, such documentation is provided by the interfacing procedures and is part of the program.

Despite the arguments for separating procedures and data structures, programmers mix them for the sake of run time efficiency. One way of reconciling efficiency with good program structure is to make use of the macro-processing facilities provided in some programming languages. Macro-processing flattens the hierarchy of non-recursive procedure calls by executing them at compile time before a problem is given. It is also a feature of the program improving transformations developed by Burstall and Darlington [1977].

The analogue of macro-processing in logic is bottom-up or middle-out reasoning combined with deletion of clauses. Such macro-processing is a special case of more general facilities provided by the connection graph proof procedure (Chapter 8). In the case of the arches program, the original formulation can be derived from the new one simply by bottom-up execution of the interfacing procedure calls.

Terms versus relations as data structures

Data in logic programs can be represented either by means of terms, as in the Append and Arches examples, or by means of relations, as in the Parsing and Family Relationships examples.

When data is represented by terms, the input to a program is normally represented by a term in the initial goal statement. Top-down execution is problem-dependent and behaves like recursive evaluation in conventional programming languages. Bottom-up execution, although it sometimes behaves like iteration, as in the Factorial and Fibonacci examples, is more often problem-independent and computationally explosive, unless it can somehow be guided by a global consideration of the problem to be solved. Global strategies for problem-solving are investigated in Chapter 9.

When data is represented by means of relations (defined by assertions and procedures) the input is normally expressed by assertions. Both top-down and bottom-up execution are problem-dependent. Top-down execution interrogates the input and bottom-up execution manipulates it, deriving new data from that which is initially given.

It is always possible to represent data by means of terms. LISP for example, represents all data by means of constant symbols and a single binary function symbol "cons". Recursion theory represents all data by means of natural numbers using a single constant symbol Ø and a unary function symbol "s". It is instructive to compare the previous formulation of the parsing problem with a formulation which represents data by means of terms.

```
Sent(x) <- Np(y), Vp(z), Append(y,z,x)
Np(x)   <- Det(y), Adj(z), Noun(v),
           Append(y,z,u), Append(u,v,x)
Vp(x)   <- Aux(y), Verb(z), Append(y,z,x)
Det(the.nil)     <-
Adj(slithy.nil)  <-
Noun(toves.nil)  <-
Aux(did.nil)     <-
Verb(gyre.nil)   <-
```

Both the input string of words and the problem of showing that it is a sentence are incorporated in the initial goal statement:

```
<- Sent(the.slithy.toves.did.gyre.nil)
```

Notice the procedure calls Append, which have no analogue in the earlier formulation of the parsing problem. When the data is represented by means of assertions, the program has direct access to the data, similar to that given by arrays in conventional programming languages. When the data is represented by terms, then special procedures like Append are needed to provide access to the contents of the data structures.

It is possible to represent data entirely by means of relations as in relational databases [Codd 1970]. Instead of representing the list

```
a, c, b, a
```

by the term

```
cons(a, cons(c, cons(b, cons(a, nil))))
```

or a.c.b.a.nil

we can give it a name, say A, and represent it by the assertions

```
Item(A,1,a) <-
Item(A,2,c) <-
Item(A,3,b) <-
Item(A,4,a) <-
Length(A,4) <-
```

where Item(x,y,z) means that

z is the y-th item of x

and Length(x,y) means that

y is the length of x.

Instead of writing an explicitly recursive program for reversing lists, either

```
Reverse(nil,nil) <-
Reverse(x.y, z)  <- Reverse(y,u), Append(u, x.nil, z)
```

or more efficiently

```
Reverse(x,y) <- Rev(x,nil,y)
Rev(nil,y,y) <-
Rev(x.y, z, u) <- Rev(y, x.z, u)
```

we can write a non-recursive program:

```
Item(rev(x), u, y) <- Item(x,v,y), Length(x,w),
                      u+v = w', w+1 =w'
Length(rev(x), y)  <- Length(x,y)
```

Here the term rev(x) names the list which is the reverse of x.

When data is represented by means of terms, the program needs to specify how data is stored and retrieved and it needs to take responsibility for the separation of the data from the higher levels of the program. Data located closer to the surface of a term can be accessed more directly than data located deeper inside. When data is represented by relations, the program defines the data at an abstract level which is independent of the storage and retrieval scheme adopted by the programming system. When a relation is defined by means of assertions, the program has direct access to the information.

Database formalisms and programming languages

Conventional database formalisms are different from the formalisms used for programming languages. Logic, in contrast, is the same whether it is used for databases, database queries and programs or for database integrity constraints and program specifications. Indeed, especially

when relations are used as data structures, the use of logic blurs the normal distinction between databases and programs. General laws for data description are indistinguishable from procedures in programs, and database integrity constraints are the same as program properties.

The conventional distinction between databases and programs is not reflected by the nature of computational problems. A representation in logic of the symbolic integration problem, for example, like the one written in PROLOG by Bergman and Kanoui [1973] can be regarded as both a database and a program. The relationship of a function to its integral is defined by means of assertions such as

sin(x) is the integral of cos(x) with respect to x

and by general rules, such as

u + v is the integral of u' + v' with respect to x
if u is the integral of u' with respect to x
and v is the integral of v' with respect to x.

The definition of the relation can be viewed both as the definition of a recursive procedure and as the description of a database by a combination of explicit assertions and implicit rules.

The desirability of combining databases and programs more intimately than is possible with conventional formalisms is beginning to be appreciated by the database community. The design of a programming language [Zloof and deLong 1977] based on query-by-example is a significant development of this kind.

Algorithm = Logic + Control

Conventional algorithms and programs expressed in conventional programming languages combine the logic of the information to be used in solving problems with the control over the manner in which the information is put to use. This relationship can be expressed symbolically by the equation

Algorithm = Logic + Control (A = L + C).

Logic programs express only the logic component L of algorithms. The control component C is exercised by the program executor, either following its own autonomously determined control decisions or else following control instructions provided by the programmer.

The conceptual separation of logic from control has several advantages:

(1) Algorithms can be constructed by successive refinement, designing the logic component before the control component.

(2) Algorithms can be improved by improving their control component without changing the logic component at all.

(3) Algorithms can be generated from specifications, can be verified and can be transformed into more efficient ones, without considering the control component, by applying deductive inference rules to the logic component alone.

(4) Inexperienced programmers and database users can restrict their interaction with the computing system to the definition of the logic component, leaving the determination of the control component to the computer.

In the systematic development of well-structured algorithms it is appropriate for the logic component to be specified before the control component. The logic component expresses the domain-specific part of an algorithm. It both determines the meaning of the algorithm and influences the way it behaves. The control component, on the other hand, determines the general-purpose problem-solving strategy. It affects only the efficiency of the algorithm without affecting its meaning.

Thus different algorithms A_1 and A_2, obtained by applying different control C_1 and C_2 to the same logic L, are equivalent in the sense that they solve the same problems with the same results. Symbolically

$$A_1 \text{ and } A_2 \text{ are equivalent if } A_1 = L + C_1 \text{ and } A_2 = L + C_2.$$

The equivalence of different algorithms having the same logic can be used to improve the efficiency of an algorithm by improving its control without changing its logic. In particular, replacing bottom-up by top-down control often, though not always, improves efficiency, whereas replacing top-down sequential execution of procedure calls by top-down consumer-producer and parallel execution almost always improves efficiency, and never harms it.

The arguments for separating logic from control are like the arguments for separating procedures from data structures. When procedures are separated from data structures, it is possible to distinguish what functions the data structures perform from the manner in which they perform them. An algorithm can be improved by replacing an inefficient data structure by a more efficient one, provided that the new data structure performs the same functions as the old one. Similarly, when logic is separated from control, it is possible to distinguish what the algorithm does, as determined by the logic component, from the manner in which it is done, as determined by the control component. An algorithm can be improved by replacing an inefficient control strategy by a more efficient one, provided that the logic component is unaltered. In both cases, it is easier to determine the meaning of the algorithm and to improve efficiency without affecting meaning.

The separation of logic from control simplifies the problem of relating programs to specifications. By ignoring the control component entirely, it is possible to use rules of deduction to show, for example, that the logic component of an algorithm is correct, because it is implied by its specification. The same techniques of deduction can also be used to generate a logic program from its specification or to transform an inefficient program into a more efficient one. These techniques have been developed by Bibel [1976a, 1976b, 1978], Clark and Tarnlund [1977] Clark and Sickel [1978], Clark and Darlington [1978] and

Hogger [1979] for logic programs and are similar to ones developed for recursion equations by Burstall and Darlington [1977] and for LISP by Manna and Waldinger [1978]. A brief introduction to these methods is presented in Chapter 10, which deals with the standard form of logic and its relationship to clausal form.

The analysis of algorithms into logic and control components provides two distinct methods for improving the efficiency of an algorithm. Given a fixed control component, incorporated in a program executor with limited problem-solving capabilities, efficiency can be improved by changing the representation of the problem in the logic component; or, given a fixed logic component, it can be improved by improving the problem-solving capabilities of the program executor. Changing the logic component is a useful short-term strategy, since the representation of the problem is generally easier to change than the problem-solver. Changing the control component, on the other hand, is a better long-term solution, since improving the problem-solver improves its performance for many different problems.

Specification of the control component

The control component can be expressed by the programmer in a separate control language; or it can be determined by the program executor itself. The provision of a separate control language allows the programmer to advise the problem-solver about program execution and is suitable for the more experienced programmer. The determination of control by the program executor, on the other hand, relieves the programmer of the need to specify control altogether and is more useful for the inexperienced programmer, the casual database user, and even the expert programmer during the early stages of program development.

A completely satisfactory, autonomous control strategy, however, has not yet been designed. The problem of designing an efficient algorithm for scheduling procedure calls, in particular, has still to be solved. The principle of procrastination, which delays execution when a procedure call can be executed in many ways, and the complementary principle, which initiates execution as soon as a procedure call can be executed in no more than one way, work efficiently in a large number of cases. But they are inadequate when all procedure calls are non-deterministic$_1$. Annotations for controlling the execution of procedure calls as coroutines have been provided in the PROLOG system [Clark and McCabe 1979] at Imperial College. They are similar to the annotations for recursion equations proposed by Schwarz [1977].

Autonomous search strategies have been designed for both top-down and bottom-up search spaces in theorem-proving. These strategies use merit orderings or evaluation functions to guide the generation of clauses in the search space. Arguments against such search strategies have been advanced by Hayes [1973]. He argues that the kind of information they provide is not adequate for effective problem-solving and proposes that more suitable information can be supplied by the programmer in an auxiliary control language. That a given relation is a function of certain arguments is an example of such information.

Control primitives for guiding search strategies have been provided in programming languages like PLANNER [Hewitt 1969], MICROPLANNER [Sussman, Winograd and Charniak 1971], CONNIVER [Sussman and McDermott 1972], POPLER [Davies 1973], SAIL [Feldman et al 1972], QA4 [Rulifson et al 1973] and QLISP [Reboh and Sacerdoti 1973]. The recommendation lists of PLANNER and MICROPLANNER in particular enable the programmer to specify the order in which procedures should be tried in order to execute a given procedure call. Such information might be useful in fault diagnosis programs, for example, when the programmer knows that a symptom P is more likely to be caused by Q than by R. This might be indicated to the problem-solver by the recommendation that the procedure

P <- Q

be tried before P <- R .

Both autonomous and user-specified control over the direction of execution have been provided in theorem-proving and in artificial intelligence programming languages. In programming languages of the PLANNER family, the direction in which procedures are executed is specified in advance by the types associated with procedure declarations (consequent theorem type if the direction is top-down, antecedent theorem type if it is bottom-up). Moreover each procedure call is assigned the type of the procedures which it is allowed to invoke. Autonomous, system-determined strategies for controlling direction of execution are more common in operational research and theorem-proving. Few strategies have been investigated, however, other than the one which chooses the direction having the current least branching rate. Both system-determined and user-specified control over direction are investigated in Chapter 8, which describes the connection graph proof procedure.

Despite the difficulties involved, the desirability of separating logic from control and of allocating responsibility for exercising control to the problem-solver is generally accepted in the field of databases. Given, for example, a data base which defines the 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.

the query Who supplies books?

```
<- Answer(y)
Answer(y) <- Supplies(x,y,z), Supply(x,u,v), Part(u,book,w)
```

specifies only the logic component of the problem. The data retrieval system needs to determine that, for the sake of efficiency, the procedure call Part(u,book,w) (containing the input) should be executed first. Given the structurally similar query

What parts are supplied by John?

```
<- Answer(y)
Answer(y) <- Supplier(x,John,z), Supply(x,u,v), Part(u,y,w)
```

however, it needs to recognise that Supplier(x,John,z) should be executed first.

For inexperienced database users it is desirable that queries be expressed in a formalism as close to natural language as possible. Since logic originates from the analysis of natural language, it is not surprising that database query languages express only the logic component of algorithms. Restricting query languages to the logic component has other advantages. It has the consequence 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. 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 it is given.

The notion that

computation = controlled deduction

was first proposed by Hayes [1973] and more recently by Bibel [1978], Kowalski [1976], Pratt [1977] and Schwarz [1977]. The similar thesis that database systems be decomposed into a relational component which defines the logic of the data, and a control component which manages data storage and retrieval, has been advocated by Codd [1970]. Hewitt's argument [1969] for the programming language PLANNER, though generally regarded as an argument against logic, can be regarded more positively as an argument for the thesis that algorithms consist of both logic and control components.

Natural Language = Logic + Control

The procedural interpretation of Horn clauses reconciles the classical role of logic in the analysis of language with the interpretation of natural language statements as programs [Winograd 1972]. Like algorithms, natural language combines logic with control. The sentence

If you want Mary to like you then give her presents and
be kind to animals.

combines the declarative information

Mary likes you if you give her presents and
are kind to animals.

with the advice that it be used top-down to solve problems of being liked by Mary to subproblems of giving her presents and being kind to animals.

Exercises

1) Let the Delete relation be defined by the procedures

```
D1          Delete(x, x.y, y) <-
D1          Delete(x, z.y, w) <- Delete(x,y,w)
```

a) Use D1-2 top-down to delete 1 from the list 2.1.nil . Exhibit the entire top-down search space.

b) Use D1-2 top-down to add 1 to the list 2.nil . Exhibit the entire search space.

c) Assume that Diff(x,y) holds when x and y are not identical. Define the relation Delallocc(x,y,w) which holds when w is the list which results from deleting all occurrences of x from the list y.

2) Describe a representation of the path-finding problem which makes it possible to find the list of all nodes in a path from one node to another.

3) Reformulate the water containers problem of Chapter 4 to incorporate loop checking into the program, so that it can be executed efficiently even if the problem-solver does not recognise and delete loops.

4) Let Partition(x,y,u,v) be defined by

```
Partition(x,y,u,v) <- Shuffle(u,v,y), Small(x,u), Big(x,v)
Shuffle(nil, v, v) <-
Shuffle(v, nil, v) <-
Shuffle(x.y, z, x.u) <- Shuffle(y,z,u)
Shuffle(y, x.z, x.u) <- Shuffle(y,z,u)
```

where Small(x,u) holds when $x \leq$ all members of u,
Big (x,u) $x \geq$ all members of u,
Shuffle(u,v,y) the lists u and v can be shuffled together to obtain the list y.

Consider the problem <- Partition(s,t,u,v) where s and t are given as input and u and v are desired as output.

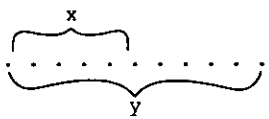
a) Define Small(x,u) and Big(x,u) recursively in terms of the relations $\leq$ and $\geq$.

b) Describe the behaviour of the procedures given above and in part a) when backtracking is used to solve the problem top-down, executing procedure calls sequentially, left-to-right.

c) Describe a more deterministic way of executing procedure calls for the same problem.

d) Redefine Partition(x,y,u,v) so that behaviour similar to that of part c) is achieved by simple left-to-right execution of procedure calls.

5) Let the relation Is(x,y) which holds when x is an initial sublist of y

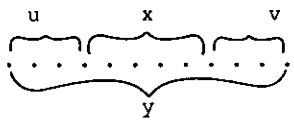


be defined by

Is(x,y) <- Append(x,z,y)

a) Define Is(x,y) recursively without using Append.

b) The relation Sl(x,y) which holds when x is a sublist of y



can be specified by

Sl(x,y) <- Append(u,x,w), Append(w,v,y)

Define Sl(x,y) recursively in terms of Is without using Append.

c) Describe an execution strategy for the two procedure calls in the specification of Sl above which behaves in the same way as top-down sequential execution of the recursive definition of Sl.

7) a) Express the 8-queens problem by means of Horn clauses:

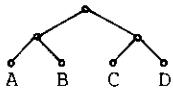
Given an 8 by 8 checker board, find a list of eight queen positions such that no queen can take another. One queen can take another if both are located on the same row, same column or same diagonal of the checker board. Assume that the Plus relation

Plus(x,y,z) (x+y = z)

is already defined by variable free assertions.

b) Modify the 8-queens problem and show that the 2-queens problem (placing 2 queens on a 2 by 2 checker board) is unsolvable by generating the entire top-down search space. Execute procedure calls in a manner which minimises the size of the search space.

8) Any binary tree can be regarded as representing a list. For example, the tree



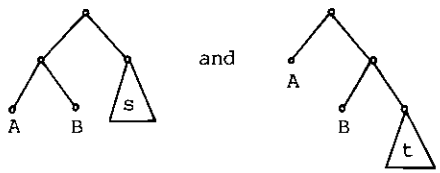
named by the term
cons(cons(tip(A), tip(B)), cons(tip(C), tip(D)))
represents the list A.B.C.D.nil .

In general the relationship Represents(x,y) which holds when the tree x represents list y can be defined by the clauses:

```
R1    Represents(nil,nil) <-
R2    Represents(tip(x), x.nil) <-
R3    Represents(cons(tip(x), y), x.z) <- Represents(y,z)
R4    Represents(cons(cons(x,y), z), w) <-
                          Represents(cons(x, cons(y,z)), w)
```

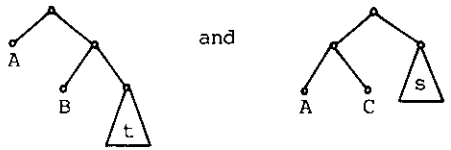
a) Define the relationship Samelists(x,y) which holds when the trees x and y represent the same lists.

b) Use procedures R1-4 and (a) to reduce the problem of showing the two trees



represent the same lists to the problem of showing that the subtrees named by s and t represent the same lists.

c) Use procedures R1-4 and (a) to show that the problem of showing the two trees



represent the same lists, where t and s name any subtrees, is not solvable.

d) Generalise the execution strategies employed in (b) and (c) and describe an efficient general strategy for executing the procedure calls in R1-4 and (a) cooperatively rather than sequentially.