Appendix A
LOGIC PROGRAMMING WITH PROLOG

Imperative programming languages reflect the architecture of the underlying von Neumann stored program computer: Programs consist of instructions stored in memory with a program counter determining which instruction to execute next. Programs perform their computations by updating memory locations that correspond to variables. Programs are prescriptive—they dictate precisely how a result is to be computed by means of a sequence of commands to be performed by the computer. Assignment acts as the primary operation, updating memory locations to produce a result obtained by incremental changes to storage using iteration and selection commands.

An alternative approach, logic programming, allows a programmer to describe the logical structure of a problem rather than prescribe how a computer is to go about solving it. Based on their essential properties, languages for logic programming are sometimes called:

1. **Descriptive or Declarative Languages** : Programs are expressed as known facts and logical relationships about a problem that hypothesize the existence of the desired result; a logic interpreter then constructs the desired result by making inferences to prove its existence.

2. **Nonprocedural Languages** : The programmer states only *what* is to be accomplished and leaves it to the interpreter to determine *how* it is to be proved.

3. **Relational Languages** : Desired results are expressed as relations or predicates instead of as functions; rather than define a function for calculating the square of a number, the programmer defines a relation, say sqr(x,y), that is true exactly when $y = x^2$.

Imperative programming languages have a descriptive component, namely expressions: “3*p + 2*q” is a description of a value, not a sequence of computer operations; the compiler and the run-time system handle the details. High-level imperative languages, like Pascal, are easier to use than assembly languages because they are more descriptive and less prescriptive.

The goal of logic programming is for languages to be purely descriptive, specifying only what a program computes and not how. Correct programs will be easier to develop because the program statements will be logical descriptions of the problem itself and not of the execution process—the assumptions made about the problem will be directly apparent from the program text.

Prolog

Prolog, a name derived from "Programming in Logic", is the most popular language of this kind; it is essentially a declarative language that allows a few control features in the interest of acceptable execution performance. Prolog implements a subset of predicate logic using the Resolution Principle, an efficient proof procedure for predicate logic developed by Alan Robinson (see [Robinson65]). The first interpreter was written by Alain Colmerauer and Philippe Roussel at Marseilles, France, in 1972.

The basic features of Prolog include a powerful pattern-matching facility, a backtracking strategy that searches for proofs, uniform data structures from which programs are built, and the general interchangeability of input and output.

Prolog Syntax

Prolog programs are constructed from **terms** that are either constants, variables, or structures.

Constants can be either atoms or numbers:

- **Atoms** are strings of characters starting with a lowercase letter or enclosed in apostrophes.
- **Numbers** are strings of digits with or without a decimal point and a minus sign.

Variables are strings of characters beginning with an uppercase letter or an underscore.

Structures consist of a **functor** or **function symbol**, which looks like an atom, followed by a list of terms inside parentheses, separated by commas. Structures can be interpreted as **predicates** (relations):

```
likes(john,mary).
male(john).
sitsBetween(X,mary,helen).
```

Structures can also be interpreted as **structur ed objects** similar to records in Pascal:

```
person(name('Kilgore','Trout'),date(november,11,1922))
tree(5, tree(3,nil,nil), tree(9,tree(7,nil,nil),nil))
```

Figure A.1 depicts these structured objects as trees.

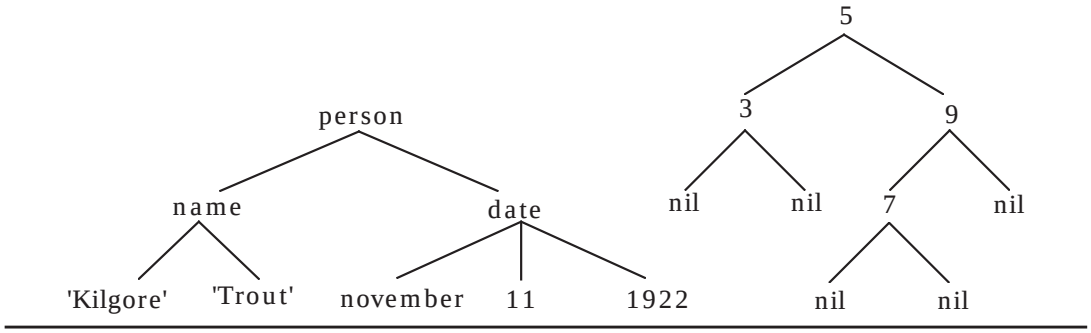

Figure A.1: Structured objects

A Prolog program is a sequence of statements, called **clauses** , of the form

P_0 :- P_1, P_2, ..., P_n.

where each of P_0, P_1, P_2, ..., P_n is an atom or a structure. A **period** terminates every Prolog clause. A clause can be read declaratively as

P_0 is true if P_1 and P_2 and ... and P_n are true

or procedurally as

To satisfy goal P_0, satisfy goal P_1 and then P_2 and then ... and then P_n.

In a clause, P_0 is called the **head** goal, and the conjunction of goals P_1, P_2, ..., P_n forms the **body** of the clause. A clause without a body is a **unit clause** or a **fact**:

"P." means "P is true" or "goal P is satisfied".

A clause without a head, written

":- P_1,P_2, ..., P_n." or "?- P_1,P_2, ..., P_n."

is a **goal clause** or a **query** and is interpreted as

"Are P_1 and P_2 and ... and P_n true?" or

"Satisfy goal P_1 and then P_2 and then ... and then P_n".

To program in Prolog, one defines a database of facts about the given information and conditional clauses or **rules** about how additional information can be deduced from the facts. A query sets the Prolog interpreter into action to try to infer a solution using the database of clauses.

BNF Syntax for Prolog

Prolog is a relatively small programming language as evidenced by a BNF specification of the core part of Prolog given in Figure A.2. The language contains a large set of predefined predicates and notational variations such as infix symbols that are not defined in this specification. In addition, Prolog allows a special syntax for lists that will be introduced later.

<program> ::= <clause list> <query> | <query>
<clause list> ::= <clause> | <clause list> <clause>
<clause> ::= <predicate> **.** | <predicate> **:-** <predicate list> **.**
<predicate list> ::= <predicate> | <predicate list> **,** <predicate>
<predicate> ::= <atom> | <atom> **(** <term list> **)**
<term list> ::= <term> | <term list> **,** <term>
<term> ::= <numeral> | <atom> | <variable> | <structure>
<structure> ::= <atom> **(** <term list> **)**
<query> ::= **?-** <predicate list> **.**
<atom> ::= <small atom> | ' <string> '
<small atom> ::= <lowercase letter> | <small atom> <character>
<variable> ::= <uppercase letter> | <variable> <character>
<lowercase letter> ::= **a | b | c | d |** ... **| x | y | z**
<uppercase letter> ::= **A | B | C | D |** ... **| X | Y | Z | _**
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
<character> ::= <lowercase letter> | <uppercase letter>
| <digit> | <special>
<special> ::= **+ | - | * | / | \ | ^ | ~ | : | . | ? | @ | # | $ | &**
<string> ::= <character> | <string> <character>

Figure A.2: BNF for Prolog

A Prolog Example

The simple example in this section serves as an introduction to Prolog programming for the beginner. Remember that a Prolog program consists of a collection of facts and rules defined to constrain the logic interpreter in such a way that when we submit a query, the resulting answers solve the problems at hand. Facts, rules, and queries can all be entered interactively, but usually a Prolog programmer creates a file containing the facts and rules, and then after "consulting" this file, enters only the queries interactively. See the documentation for instructions on consulting a file with a particular implementation of Prolog.

We develop the example incrementally, adding facts and rules to the database in several stages. User queries will be shown in boldface followed by the response from the Prolog interpreter. Comments start with the symbol % and continue to the end of the line.

Some facts:
```
parent(chester,irvin).
parent(chester,clarence).
parent(chester,mildred).
parent(irvin,ron).
parent(irvin,ken).
parent(clarence,shirley).
parent(clarence,sharon).
parent(clarence,charlie).
parent(mildred,mary).
```

Some queries:

```
?- parent(chester,mildred).
yes

?- parent(X,ron).
X = irvin
yes

?- parent(irvin,X).
X = ron;
X = ken;            % The user-typed semicolon asks the system for
no                  %   more solutions.

?- parent(X,Y).
X =chester
Y = irvin           % System will list all of the parent pairs, one at a time,
yes                 %   if semicolons are entered.
```

Additional facts:
```
male(chester).      female(mildred).
male(irvin).        female(shirley).
```

```
male(clarence).      female(sharon).
male(ron).           female(mary).
male(ken).
male(charlie).
```

Additional queries:

```
?- parent(clarence,X), male(X).
X = charlie
yes
?- male(X), parent(X,ken).
X = irvin
yes
?- parent(X,ken), female(X).
no
```

Prolog obeys the "closed world assumption" that presumes that any predicate that cannot be proved must be false.

```
?- parent(X,Y), parent(Y,sharon).
X = chester
Y = clarence
yes
```

These queries suggest definitions of several family relationships.

Some rules:

```
father(X,Y) :- parent(X,Y), male(X).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
paternalgrandfather(X,Y) :- father(X,Z), father(Z,Y).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

The scope of a variable in Prolog is solely the clause in which it occurs.

Additional queries:

```
?- paternalgrandfather(X,ken).
X = chester
yes
?- paternalgrandfather(chester,X).
X = ron;
X = ken;
X = shirley;          % Note the reversal of the roles of input and output.
X = sharon;
X = charlie;
no
```

```
?- sibling(ken,X).
X = ron;
X = ken;
no
```

The inference engine concludes that ken is a sibling of ken since parent(irvin,ken) and parent(irvin,ken) both hold. To avoid this consequence, the description of sibling needs to be more carefully constructed.

Predefined Predicates

1. The equality predicate = permits infix notation as well as prefix.

```
?- ken = ken.
yes
?- =(ken,ron).
no
?- ken = X.          % Can a value be found for X to make it the same as ken?
X = ken
yes                  % The equal operator represents the notion of unification.
```

2. "not" is a unary predicate:
   not(P) is true if P cannot be proved and false if it can.

```
?- not(ken=ron).
yes
?- not(mary=mary).
no
```

The closed world assumption governs the way the predicate "not" works since any goal that cannot be proved using the current set of facts and rules is assumed to be false and its negation is assumed to be true. The closed world assumption presumes that any property not recorded in the database is not true. Some Prolog implementations omit the predefined predicate not because its behavoir diverges from the logical not of predicate calculus in the presence of variables (see [Sterling86]). We have avoided using not in the laboratory exercises in this text.

The following is a new sibling rule (the previous rule must be removed):

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y), not(X=Y).
```

Queries:

```
?- sibling(ken,X).
X = ron;
no
```

```
?- sibling(X,Y).
X = irvin
Y = clarence;          % Predicate sibling defines a symmetric relation.
X = irvin              % Three sets of siblings produce six answers.
Y = mildred;
X = clarence           % The current database allows 14 answers.
Y = irvin;
X = clarence
Y = mildred;
X = mildred
Y = irvin;
Y = mildred
X = clarence           % No semicolon here.
yes
```

A relation may be defined with several clauses:

```
closeRelative(X,Y) :- parent(X,Y).
closeRelative(X,Y) :- parent(Y,X).
closeRelative(X,Y) :- sibling(X,Y).
```

There is an implicit **or** between the three definitions of the relation closeRelative. This disjunction may be abbreviated using semicolons as

```
closeRelative(X,Y) :- parent(X,Y) ; parent(Y,X) ; sibling(X,Y).
```

We say that the three clauses (or single abbreviated clause) define(s) a "procedure" named closeRelative with arity two (closeRelative takes two parameters). The identifier closeRelative may be used as a different predicate with other arities.

Recursion in Prolog

We want to define a predicate for "X is an ancestor of Y". This is true if

```
parent(X,Y) or
parent(X,Z) and parent(Z,Y) or
parent(X,Z), parent(Z,Z1), and parent(Z1,Y) or
   :                        :
```

Since the length of the chain of parents cannot be predicted, a recursive definition is required to allow an arbitrary depth for the definition. The first possibility above serves as the basis for the recursive definition, and the rest of the cases are handled by an inductive step.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Add some more facts:

```
parent(ken,nora).          female(nora).
parent(ken,elizabeth).     female(elizabeth).
```

Since the family tree defined by the Prolog clauses is becoming fairly large, Figure A.3 shows the parent relation between the twelve people defined in the database of facts.

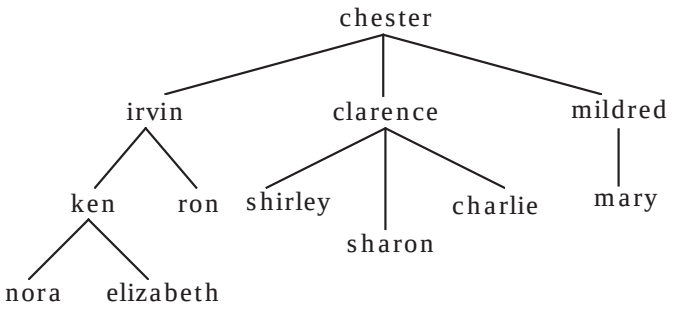

Figure A.3: A Family Tree

Some queries:

```
?- ancestor(mildred,mary).
yes     % because parent(mildred,mary).

?- ancestor(irvin,nora).
yes     % because
        %   parent(irvin,ken)
        %   and ancestor(ken,nora) because parent(ken,nora).

?- ancestor(chester,elizabeth).

yes     % because
        %   parent(chester,irvin)
        %   and ancestor(irvin,elizabeth)
        %       because parent(irvin,ken)
        %       and ancestor(ken,elizabeth) because parent(ken,elizabeth).

?- ancestor(irvin,clarence).
no  % because parent(irvin,clarence) is not provable and
    %           whoever is substituted for Z it is impossible to
    %           prove parent(irvin,Z) and ancestor(Z,clarence).
```

All possibilities for Z are tried that make parent(irvin,Z) true, namely Z=ron and Z=ken, and both ancestor(ron,clarence) and ancestor(ken,clarence) fail.

The reader is encouraged to write Prolog definitions for other predicates dealing with family relationships—for example, mother, child, uncle, niece, maternal grandfather, first cousin, and descendant.

Control Aspects of Prolog

In pure logic programming, the predicates in a goal question may be considered in any order or in parallel since logical conjunction (and) is commutative and associative. Furthermore, alternate rules defining a particular predicate (procedure) may be considered in any order or in parallel since logical disjunction (or) is commutative and associative.

Since Prolog has been implemented with a concern for efficiency, its interpreters act with a deterministic strategy for discovering proofs.

1. In defining a predicate, the order in which clauses are presented to the system (the **rule order** or **clause order**) is the order in which the interpreter tests them—namely, from top to bottom. Here the term "rule" includes any clause, including facts (clauses without bodies).

   Rule order determines the order in which answers are found. Observe the difference when the two clauses in ancestor are reversed.

   ```
   ancestor2(X,Y) :- parent(X,Z), ancestor2(Z,Y).
   ancestor2(X,Y) :- parent(X,Y).
   ```

   ```
   ?- ancestor(irvin,Y).
   Y = ron, ken, nora, elizabeth        % Four answers returned separately.
   ```

   ```
   ?- ancestor2(irvin,Y).
   Y = nora, elizabeth, ron, ken        % Four answers returned separately.
   ```

   Depending on the nature of the query, different rule orders may have different execution speeds when only a yes or no, or only one solution is desired.

2. In defining a rule with a clause, the order in which terms (subgoals) are listed on the right-hand side (the **goal order**) is the order in which the interpreter will try to satisfy them—namely, from left to right.

   Goal order determines the shape of the search tree that the interpreter explores in its reasoning. In particular, a poor choice of goal order may permit a search tree with an infinite branch in which the inference engine will become lost. The version below is ancestor2 with the subgoals in the body of the first clause interchanged.

   ```
   ancestor3(X,Y) :- ancestor3(Z,Y), parent(X,Z).
   ancestor3(X,Y) :- parent(X,Y).
   ```

?- **ancestor(irvin,elizabeth).**
yes

?- **ancestor3(irvin,elizabeth).**

This query invokes a new query
ancestor3(Z,elizabeth), parent(irvin,Z).
which invokes
ancestor3(Z1,elizabeth), parent(Z,Z1), parent(irvin,Z).
which invokes
ancestor3(Z2,elizabeth), parent(Z1,Z2), parent(Z,Z1), parent(irvin,Z).
which invokes ...

The eventual result is a message such as

"Out of local stack during execution; execution aborted."

The problem with this last definition of the ancestor relation is the left recursion with uninstantiated variables in the first clause. If possible, the leftmost goal in the body of a clause should be nonrecursive so that a pattern match occurs and some variables are instantiated before a recursive call is made.

Lists in Prolog

As a special notational convention, a list of terms in Prolog can be represented between brackets: [a, b, c, d]. As in Lisp, the head of this list is a, and its tail is [b, c, d]. The tail of [a] is [ ], the empty list. Lists may contain lists: [5, 2, [a, 8, 2], [x], 9] is a list of five items.

Prolog list notation allows a special form to direct pattern matching. The term [H | T] matches any list with at least one element:

H matches the head of the list, and

T matches the tail.

A list of terms is permitted to the left of the vertical bar. For example, the term [X,a,Y | T] matches any list with at least three elements whose second element is the atom a:

X matches the first element,

Y matches the third element, and

T matches the rest of the list, possibly empty, after the third item.

Using these pattern matching facilities, values can be specified as the intersection of constraints on terms instead of by direct assignment.

Although it may appear that lists form a new data type in Prolog, in fact they are ordinary structures with a bit of "syntactic sugar" added to make them easier to use. The list notation is simply an abbreviation for terms constructed with the predefined "." function symbol and with [ ] considered as a special atom representing the empty list. For example,

[a, b, c] is an abbreviation for .(a, .(b, .(c, [ ])))

[H | T] is an abbreviation for .(H, T)

[a, b | X] is an abbreviation for .(a, .(b, X))

Note the analogy with the relationship between lists and S-expressions in Lisp. In particular, the "list" object [a | b] really represents an object corresponding to a dotted pair in Lisp—namely, .(a,b).

List Processing

Many problems can be solved in Prolog by expressing the data as lists and defining constraints on those lists using patterns with Prolog's list representation. We provide a number of examples to illustrate the process of programming in Prolog.

1. Define last(L,X) to mean "X is the last element of the list L".

   The last element of a singleton list is its only element.

   ```
   last([X], X).
   ```

   The last element of a list with two or more elements is the last item in its tail.

   ```
   last([H|T], X) :- last(T, X).
   ```

   ```
   ?- last([a,b,c], X).
   X = c
   yes
   ?- last([ ], X).
   no
   ```

Observe that the "illegal" operation of requesting the last element of an empty list simply fails. With imperative languages a programmer must test for exceptional conditions to avoid the run-time failure of a program. With logic programming, an exception causes the query to fail, so that a calling program can respond by trying alternate subgoals. The predicate last acts as a generator when run "backward".

```
?- last(L, a).
L = [a];
L = [_5, a];          % The underline indicates system-generated variables.
L = [_5, _9, a];
L = [_5, _9, _13, a] …
```

The variable H in the definition of last plays no role in the condition part (the body) of the rule; it really needs no name. Prolog allows **anonymous variables**, denoted by an underscore:

```
last([ _ |T], X) :- last(T, X).
```

Another example of an anonymous variable can be seen in the definition of a father relation:

```
father(F) :- parent(F, _ ), male(F).
```

The scope of an anonymous variable is its single occurrence. Generally, we prefer using named variables for documentation rather than anonymous variables, although anonymous variables can be slightly more efficient since they do not require that bindings be made.

2. Define member(X,L) to mean "X is a member of the list L".

   For this predicate we need two clauses, one as a basis case and the second to define the recursion that corresponds to an inductive specification.

   The predicate succeeds if X is the first element of L.

   ```
   member(X, [X|T]).
   ```

   If the first clause fails, check if X is a member of the tail of L.

   ```
   member(X, [H|T]) :- member(X,T).
   ```

   If the item is not in the list, the recursion eventually tries a query of the form member(X,[ ]), which fails since the head of no clause for member has an empty list as its second parameter.

3. Define delete(X,List,NewList) to mean
   "The variable NewList is to be bound to a copy of List with all instances of X removed".

   When X is removed from an empty list, we get the same empty list.

   ```
   delete(X,[ ],[ ]).
   ```

   When an item is removed from a list with that item as its head, we get the list that results from removing the item from the tail of the list (ignoring the head).

   ```
   delete(H,[H|T],R) :- delete(H,T,R).
   ```

If the previous clause fails, X is not the head of the list, so we retain the head of L and take the tail that results from removing X from the tail of the original list.

```
delete(X,[H|T],[H|R]) :- delete(X,T,R).
```

4. Define union(L1,L2,U) to mean

   "The variable U is to be bound to the list that contains the union of the elements of L1 and L2".

   If the first list is empty, the result is the second list.

```
union([ ],L2,L2).                                    % clause 1
```

   If the head of L1 is a member of L2, it may be ignored since a union does not retain duplicate elements.

```
union([H|T],L2,U) :- member(H,L2), union(T,L2,U).    % clause 2
```

   If the head of L1 is a not member of L2 (clause 2 fails), it must be included in the result.

```
union([H|T],L2,[H|U]) :- union(T,L2,U).              % clause 3
```

   In the last two clauses, recursion is used to find the union of the tail of L1 and the list L2.

5. Define concat(X,Y,Z) to mean "the concatenation of lists X and Y is Z". In the Prolog literature, this predicate is frequently called append.

```
concat([ ], L, L).                                   % clause α
concat([H|T], L, [H|M]) :- concat(T, L, M).          % clause β
```

```
?- concat([a,b,c], [d,e], R).
R = [a,b,c,d,e]
yes
```

   The inference that produced this answer is illustrated by the search tree in Figure A.4. When the last query succeeds, the answer is constructed by unwinding the bindings:

```
R = [a | M] = [a | [b | M1]] = [a,b | M1] = [a,b | [c | M2]]
  = [a,b,c | M2] = [a,b,c | [d,e]] = [a,b,c,d,e].
```

   Figure A.5 shows the search tree for another application of concat using semicolons to generate all the solutions.

   To concatenate more than two lists, use a predicate that joins the lists in parts.

```
concat(L,M,N,R) :- concat(M,N,Temp), concat(L,Temp,R).
```

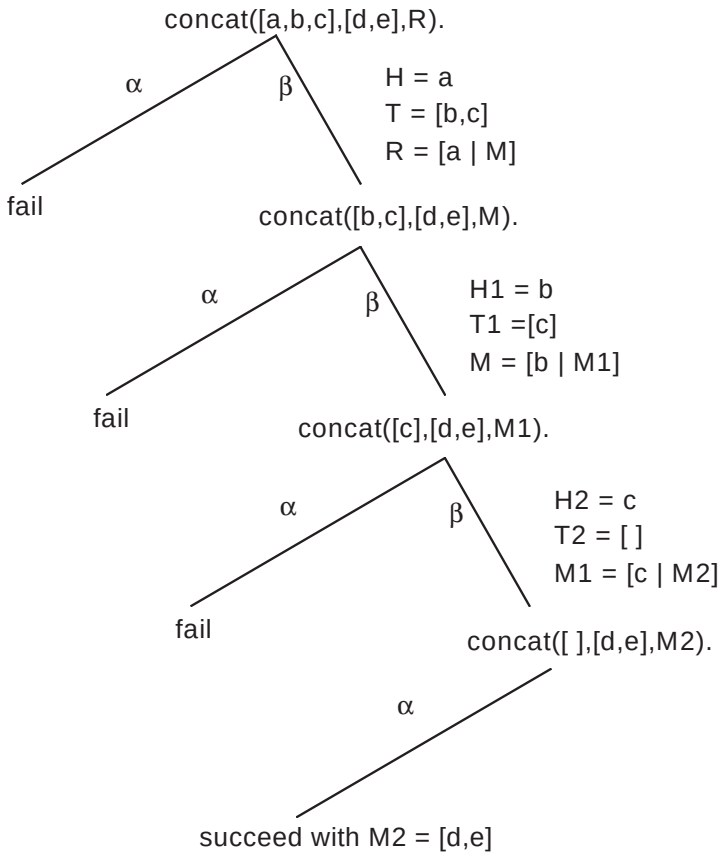

Figure A.4: A Search Tree for concat

No confusion results from using the same name for this predicate, since the two versions are distinguished by the number of parameters they take (the arities of the predicates).

6. Define reverse(L,R) to mean "the reverse of list L is R".

```
reverse([ ], [ ]).
reverse([H|T], L) :- reverse(T, M), concat(M, [H], L).
```

In executing concat, the depth of recursion corresponds to the number of times that items from the first list are attached (cons) to the front of the second list. Taken as a measure of complexity, it suggests that the work done by concat is proportional to the length of the first list. When reverse is applied to a list of length n, the executions of concat have first argument of lengths, n-1, n-2, ..., 2, 1, which means that the complexity of reverse is proportional to n^2.

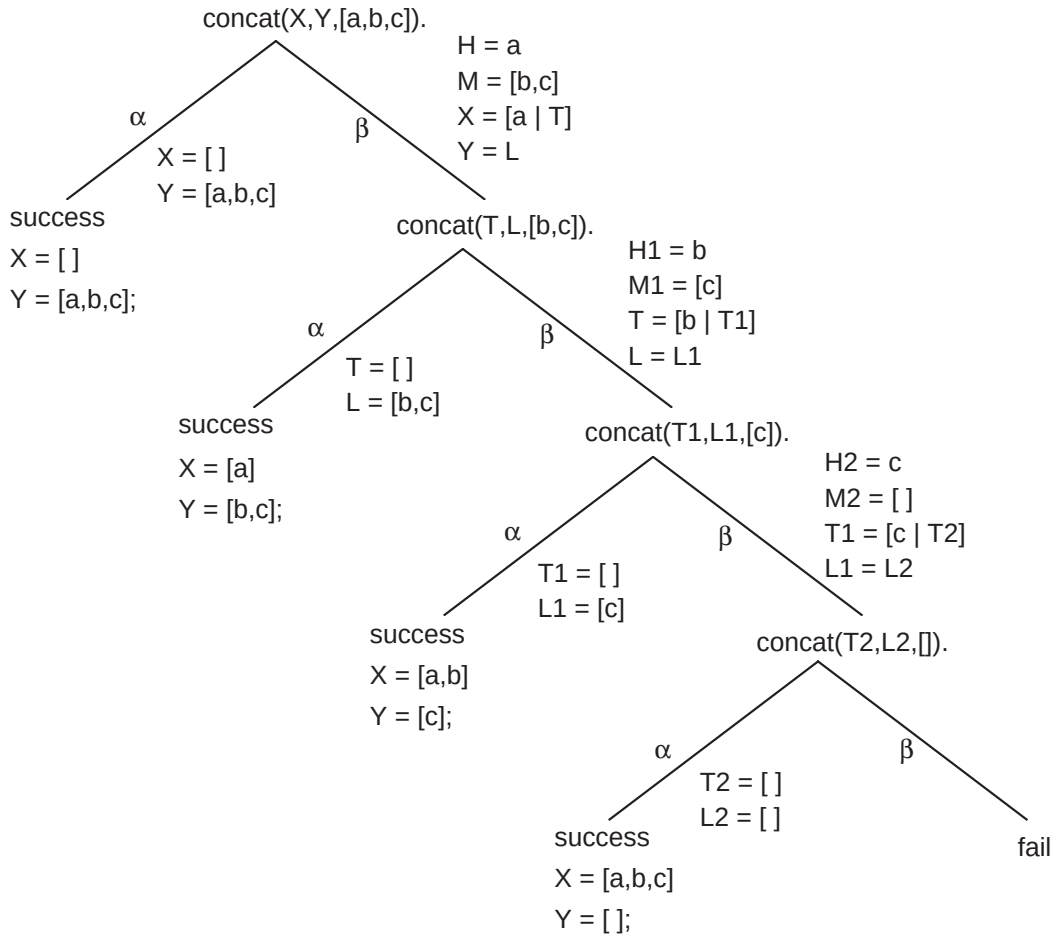

Figure A.5: Another Search Tree for concat

7. An improved reverse using an accumulator:

```
rev(L, R) :- help(L, [ ], R).

help([ ], R, R).
help([H|T], A, R) :- help(T, [H|A], R).
```

The predicate help is called n times if the original list is of length n, so the complexity of rev is proportional to n. Observe that the predicat help is tail recursive.

Sorting in Prolog

A few relations are needed for comparing numbers when sorting a list of numbers (equal and not equal are described later):

```
M < N, M =< N, M > N, M >= N.
```

These relations demand that both operands be numeric atoms or arithmetic expressions whose variables are bound to numbers.

Insertion Sort

If a list consists of head H and tail T, the idea with the insertion sort is to sort the tail T (recursively) and then insert the item H into its proper place in the tail.

```
insertSort([ ], [ ]).
insertSort([X|T], M) :- insertSort(T, L), insert(X, L, M).

insert(X, [H|L], [H|M]) :- H<X, insert(X, L, M).
insert(X, L, [X|L]).
```

Observe that the clauses for insert are order dependent. The second clause is executed when the first goal of the first clause fails—namely, when H>=X. If these clauses are switched, the definition of insert is no longer correct.

Although this dependence on the rule order of Prolog is common in Prolog programming and may be slightly more efficient, a more logical program is constructed by making the clauses that define insert independent of each other:

```
insert(X, [ ], [X]).
insert(X, [H|L], [X,H|L]) :- X=<H.
insert(X, [H|L], [H|M]) :- X>H, insert(X,L,M).
```

Now only one clause applies to a given list. The original clause insert(X, L, [X|L]). must be split into two cases depending on whether L is empty or not.

Quick Sort

The quick sort works by splitting the list into those items less than or equal to a particular element, called the **pivot**, and the list of those items greater than the pivot. The first number in the list can be chosen as the pivot. After the two sublists are sorted (recursively), they are concatenated with the pivot in the middle to form a sorted list.

The splitting operation is performed by the predicate partition(P, List, Left, Right), which means P is a pivot value for the list List, Left = { X∈List | X≤P }, and Right = { X∈List | X>P }.

```
partition(P, [ ], [ ], [ ]).
partition(P, [A|X], [A|Y], Z) :- A=<P, partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A>P, partition(P, X, Y, Z).

quickSort([ ], [ ]).
quickSort([H|T], S) :-  partition(H, T, Left, Right),
                        quickSort(Left, NewLeft),
                        quickSort(Right, NewRight),
                        concat(NewLeft, [H|NewRight], S).
```

The clauses for both partition and quickSort can be entered in any order since they are made mutually exclusive either by the patterns in their head terms or by the "guard" goals at the beginning of their bodies. The goals in the definition of partition may be turned around without affecting correctness but with a severe penalty of diminished efficiency since the recursive call will be made whether it is needed or not. An empirical test showed the sorting of 18 integers took 100 times longer with the goals switched than with the original order.

The Logical Variable

A variable in an imperative language is not the same concept as a variable in mathematics:

1. A program variable refers to a memory location that may have changes in its contents; consider an assignment N := N+1.
2. A variable in mathematics simply stands for a value that once determined will not change. The equations x + 3y = 11 and 2x – 3y = 4 specify values for x and y—namely, x=5 and y=2—which will not be changed in this context. A variable in Prolog is called a **logical variable** and acts in the manner of a mathematical variable.
3. Once a logical variable is bound to a particular value, called an **instantiation** of the variable, that binding cannot be altered unless the pattern matching that caused the binding is undone because of backtracking.
4. The destructive assignment of imperative languages, where a variable with a value binding is changed, cannot be performed in logic programming.
5. Terms in a query change only by having variables filled in for the first time, never by having a new value replace an existing value.
6. An iterative accumulation of a value is obtained by having each instance of a recursive rule take the values passed to it and perform computations of values for new variables that are then passed to another call.

7. Since a logical variable is "write-once", it is more like a constant identifier with a dynamic defining expression as in Ada (or Pelican) than a variable in an imperative language.

The power of logic programming and Prolog comes from using the logical variable in structures to direct the pattern matching. Results are constructed by binding values to variables according to the constraints imposed by the structures of the arguments in the goal term and the head of the clause being matched. The order that variables are constrained is generally not critical, and the construction of complex values can be postponed as long as logical variables hold their places in the structure being constructed.

Equality and Comparison in Prolog

Prolog provides a number of different ways to compare terms and construct structures. Since beginning Prolog programmers often confuse the various notions of equality and related predicates, we provide a brief overview of these predicates.

Unification

"T1 = T2" Succeed if term T1 can be unified with term T2.

```
| ?- f(X,b) = f(g(a),Y).
X = g(a)
Y = b
yes
```

Numerical Comparisons

"=:=", "=\=", "<", ">", "=<", ">="

Evaluate both expressions and compare the results.

```
| ?- 5<8.
yes
| ?- 5 =< 2.
no
| ?- N =:= 5.
! Error in arithmetic expression: not a number (N not instantiated to a number)
no
| ?- N = 5, N+1 =< 12.
N = 5                % The unification N = 5 causes a binding of N to 5.
yes
```

Forcing Arithmetic Evaluation (is)

"N is Exp" Evaluate the arithmetic expression Exp and try to unify the resulting number with N, a variable or a number.

```
| ?- M is 5+8.
M = 13
yes
| ?- 13 is 5+8.
yes
| ?- M is 9, N is M+1.
M = 9
N = 10
yes
| ?- N is 9, N is N+1.
no                          % N is N+1 can never succeed.
| ?- 6 is 2*K.
! Error in arithmetic expression: not a number (K not instantiated to a number)
no
```

The infix predicate is provides the computational mechanism to carry out arithmetic in Prolog. Consider the following predicate that computes the factorial function:

The factorial of 0 is 1.

```
fac(0,1).
```

The factorial of N>0 is N times the factorial of N-1.

```
fac(N,F) :- N>0, N1 is N-1, fac(N1,R), F is N*R.
```

```
| ?- fac(5,F).
F = 120
yes
```

Identity

"X == Y" Succeed if the terms currently instantiated to X and Y are literally identical, including variable names.

```
| ?- X=g(X,U), X==g(X,U).
yes
| ?- X=g(a,U), X==g(V,b).
no
| ?- X\==X.                  % "X \== X" is the negation of "X == X"
no
```

Term Comparison (Lexicographic)

"T1 @< T2", "T1 @> T2", "T1 @=< T2", "T1 @>= T2"

```
| ?- ant @< bat.
yes
| ?- @<(f(ant),f(bat)).          % infix predicates may also be entered
yes                              %   as prefix
```

Term Construction

"T =.. L" L is a list whose head is the atom corresponding to the principal functor of term T and whose tail is the argument list of that functor in T.

```
| ?- T =.. [@<,ant,bat], call(T).
T = ant@<bat
yes
| ?- T =.. [@<,bat,bat],call(T).
no
| ?- T =.. [is,N,5], call(T).
N = 5,
T = (5 is 5)
yes
| ?- member(X,[1,2,3,4]) =.. L.
L = [member,X,[1,2,3,4]]
yes
```

Input and Output Predicates

Several input and output predicates are used in the laboratory exercises. We describe them below together with a couple of special predicates.

get0(N) N is bound to the ascii code of the next character from the current input stream (normally the terminal keyboard). When the current input stream reaches its end of file, a special value is bound to N and the stream is closed. The special value depends on the Prolog system, but two possibilities are:

26, the code for control-Z or

-1, a special end of file value.

put(N) The character whose ascii code is the value of N is printed on the current output stream (normally the terminal screen).

see(F) The file whose name is the value of F becomes the current input stream.

seen Close the current input stream.

tell(F) The file whose name is the value of F becomes the current output stream.

told Close the current output stream.

read(T) The next Prolog term in the current input stream is bound to T. The term in the input stream must be followed by a period.

write(T) The Prolog term bound to T is displayed on the current output stream.

tab(N) N spaces are printed on the output stream.

nl Newline prints a linefeed character on the current output stream.

abort Immediately terminate the attempt to satisfy the original query and return control to the top level.

name(A,L) A is a literal atom or a number, and L is a list of the ascii codes of the characters comprising the name of A.

```
| ?- name(A,[116,104,101]).
A = the

| ?- name(1994,L).
L = [49, 57, 57, 52]
```

call(T) Assuming T is instantiated to a term that can be interpreted as a goal, call(T) succeeds if and only if T succeeds as a query.

This Appendix has not covered all of Prolog, but we have introduced enough Prolog to support the laboratory exercises in the text. See the further readings at the end of Chapter 2 for references to more material on Prolog.