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 **structured 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.

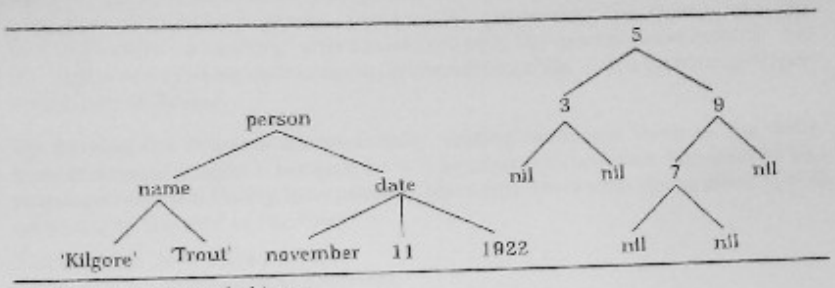

Figure A.1: Structured objects

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.

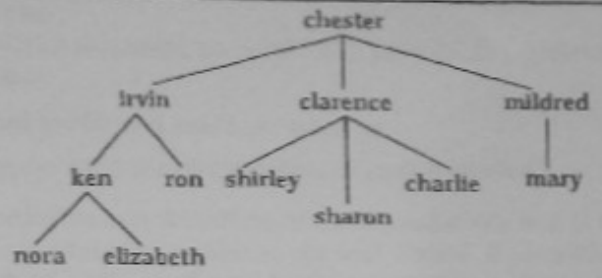

Figure A.3: A Family Tree

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".