Prolog Experiments in Discrete Mathematics, Logic, and Computability

James L. Hein
Portland State University

January 2005

Contents

Preface

This book contains programming experiments that are designed to reinforce the learning of discrete mathematics, logic, and computability. It is the outgrowth of the experimental portion of a one-year course in discrete structures, logic, and computability for sophomores at Portland State University. The course has evolved over the last ten years from a standard course into a course that uses programming experiments as part of the required activity. Most of the experiments are short and to the point, just like traditional homework problems, so that they reflect the daily classroom work. The experiments in the book are organized to accompany the material in *Discrete Structures*, *Logic*, *and Computability*, Second Edition, by James L. Hein.

In traditional experimental laboratories, there are many different tools that are used to perform various experiments. The Prolog programming language is the tool used for the experiments in this book. Prolog has both commercial and public versions. The language is easy to learn and use because its syntax and semantics are similar to that of mathematics and logic. So the learning curve is steep and no prior knowledge of the language is assumed. In fact, the experiments are designed to introduce language features as tools to help explore the problems being studied.

The instant feedback provided by Prolog's interactive environment can help the process of learning. When students get immediate feedback to indicate success or failure, there is a powerful incentive to try and get the right solution. This encourages students to ask questions like, "What happens if I do this?" This supports the idea that exploration and experimentation are keys to learning.

The book builds on the traditional laboratory experiences that most students receive in high school science courses. i.e., experimentation, observation, and conclusion. Each section contains an informal description of a topic—with examples as necessary—and presents a list of experiements to perform. Some experiements are simple, like using a program to check answers to hand calculations, and some experiements are more sophisticated, like checking whether a definition works, or constructing a small program to

explore a concept.

Laboratory Notebook

The book has been used for several years at Portland State University for the laboratory portion of courses in logical structures and computational structures. Here is a sample of the requirements for the laboratory portion of a course.

- Keep a laboratory notebook that has a hard cover and permanent pages that are either blank or lined, but not cross hatched.
- Keep a detailed written account of each lab assignment. If you print, use lower case. Cut and tape computational results into the prose. Write on both sides of each page.
- Keep two pages at the beginning of the laboratory notebook for a table of contents that lists the name and page number of each lab assignment with subheadings for the experiments performed. Keep it up-to-date as each experiment is written up.
- Start the write-up for each lab assignment on a new page. The write-up should consist of three parts—**Name, Description,** and **Conclusion.** Underline the words so that they stand out. Here is a description of the parts.

  Name. Write down the name of the assignment. e.g., Truth tables.

  Description. This is where you describe what was asked for and what happened. There may be several experiments to perform. There will be items such as examples, questions, tests, calculations, and program definitions. Of course, there will be corresponding computational results for most of these items. Cut and tape appropriate computational results into the prose and be sure to comment on their meaning and significance. You are encouraged to explore new ideas that come to mind. Describe these too.

  Conclusion. Write down your general observations about the results of the experiments performed. Write about what you learned, what ideas were clarified, what problems arose, what questions arose, what was interesting, what was exciting, and so on.

Notes

- It is not necessary to cut and tape every bit of a computer session into your lab notebook. For example, you don't need to cut and tape the example computations that are used to introduce a topic. However, you may wish to include some of them so that you can refer back to them.

When asked for a program definition, be sure to cut and tape the definition along with any input data, commands, and output used in testing. The computational results must be cut small enough to fit onto part of a page and still have room on the page to continue writing about what is going on. If some computational results are too large and can't be cut up into smaller parts (e.g., a graph), then reduce them on a copy machine. Tape each item along the entire length of each of its four edges. Foldouts or loose papers are not acceptable.

- Computational results must be accompanied by appropriate comments about their meaning and significance. Be sure to place the computational results within the prose description so that the reader can see them without moving back and forth to different pages. Don't separate the computational results from the prose that describes them. In other words, the prose description will be interspersed with neatly taped displays of computational results that fit into the prose.
- Write up each lab assignment as a very short story that someone else can read and understand without having to use supplementary material. You must use complete sentences.

1

Introduction to Prolog

The Prolog language allows us to explore a wide range of topics in discrete mathematics, logic, and computability. Prolog's powerful pattern-matching ability and its computation rule give us the ability to experiment in two directions. For example, a typical experiment might require a test of a definition with a few example computations. Prolog allows this, as do all programming languages. But the Prolog computation rule also allows a definition to be tested in reverse, by specifying a result and then asking for the elements that give the result. From a functional viewpoint this means that we can ask for domain elements that map to a given result.

After a brief introduction to Prolog we'll start right in doing experiments. To keep the emphasis on the discrete mathematics, logic, and computability, we'll introduce new Prolog tools in the experiments where they are needed.

1.1 Getting Started

This section introduces a few facts to help you get started using Prolog. To start the Prolog interpreter in a UNIX environment type *prolog* (or *sicstus* for those using SICStus Prolog) and hit return. Once Prolog has started up it displays the prompt

```
| ?-
```

which indicates that the interpreter is waiting for a command from the user. All commands must end with a period. For example, the command

```
| ?- integer(3.4).
```

returns the answer no because 3.4 is not an integer. A command is usually called a *goal* or a *query*. To exit the interpreter type control D—press the control key and the D key at the same time.

Before we go any further, we're going to go through an introductory example to get the look and feel of Prolog. After the example, we'll present some useful programming tools.

1.2 An Introductory Example

A Prolog program is a set of facts or rules called *definite clauses.* We'll usually refer to them as *clauses*. The example program that follows describes some family relationships. We'll use the predicates "par" and "grand" with the following meanings.

par(X, Y) means that X is a parent of Y.
grand(X, Y) means that X is a grandparent of Y.

Now we'll list the program, which consists of some parent facts together with a rule that defines the grandparent relationship in terms of parents. Note that a comment is signified by the character % followed by any sequence of characters up to the end of the line. Another way to comment is to place any sequence of characters, including new lines, between the symbols /* and */.

```
% Here is a set of facts describing parental relationships.

par(lloyd, james).
par(lloyd, janet).
par(ruth, james).
par(ruth, janet).
par(emma, lloyd).
par(katherine, ruth).
par(adolph, lloyd).
par(edgar, ruth).

% The grandparent relationship. Any rule of the form
% A :- B, C is read, "A is true if B is true and C is true."

grand(X, Z) :- par(Y, Z), par(X, Y).
```

Now, suppose that you have entered this program into a file named *familyTree*. To read in the program type the following command.

```
|?- [familyTree].
```

Once the program has been read in it won't do anything until it is presented with a goal. We'll give some example goals that ask questions about children and grandparents.

Finding the Children of a Person

Suppose that we want to find the children of ruth. We can find them by typing the following goal, where the letter C stands for a variable.

```
|?- par(ruth, C).
```

Prolog will search the program statements from top to bottom until it can match the goal with some fact or the left part of a rule. In this case, the goal matches par(ruth, james) by identifying C with james. Prolog responds with

```
C = james ?
```

At this point, we can hit return and Prolog will answer

```
Yes.
```

But if we hit a semicolon followed by return, then Prolog will backtrack and continue to search for another match for the goal par(ruth, C). The goal matches par(ruth, janet) by identifying C with janet. Prolog responds with

```
C = janet ?
```

If we hit a semicolon followed by return, then Prolog will continue to search for another match. It doesn't find any and lets us know with the statement

```
no.
```

So we can conclude that the two children of ruth are james and janet.

Finding the Grandparents of a Person

Suppose that we want to find all the grandparents of james. In this case, we can enter the goal

```
|?- grand(A, james).
```

Prolog matches this goal with grand(X, Z) in the rule

```
grand(X, Z) :- par(Y, Z), par(X, Y).
```

It identifies X with A and Z with james. Now Prolog attempts to find matches for the two goals on the right side of the grandparent rule:

```
par(Y, james)     and     par(A, Y).
```

It tries par(Y, james) first, and finds a match with par(lloyd, james) by identifying Y with lloyd. With this identification it tries to find a match for the second goal par(A, lloyd). This goal matches the fact par(emma, lloyd) by identifying A with emma. So Prolog outputs the answer

A = emma?

If we hit semicolon followed by return, then Prolog will try to find another match for the goal par(A, lloyd). This goal matches the fact par(adolph, lloyd) by identifying A with adolph. So Prolog outputs the answer

A = adolph?

If we hit semicolon followed by return, then Prolog will not find another match for the goal par(A, lloyd). So it will backtrack and try to find a different match for the first of the two goals

par(Y, james) and par(A, Y).

The goal par(Y, james) matches the fact par(ruth, james) by identifying Y with ruth. With this identification it tries to find a match for the second goal par(A, ruth). This goal matches the fact par(katherine, ruth) by identifying A with katherine. So Prolog outputs the answer

A = katherine?

If we hit semicolon followed by return, then Prolog will try to find another match for the goal par(A, ruth). This goal matches the fact par(edgar, ruth) by identifying A with edgar. So Prolog outputs the answer

A = edgar?

If we hit semicolon followed by return, then Prolog will not find another match for the goal par(A, ruth). When it backtracks, it won't find any new matches for the goals par(Y, james) and par(A, Y). So it backtracks to the original goal grand(A, james). There are no other matches for this goal, so Prolog outputs the answer

no.

Thus the four grandparents of james are emma, adolph, katherine, and edgar.

1.3 Some Programming Tools

We'll record here several Prolog programming tools that should prove useful in doing the experiments.

Loading Information

To read in the contents of a file named *filename* type

```
|?- [filename].
```

and hit return. If the file name contains characters other than letters or digits, then put single quotes around the filename. For example, if the name of the file is *file.p*, then type

```
|?- ['file.p'].
```

will load the file named *file.p*.

You can read in several files at once. For example, to read in files named *foo*, *goo*, and *moo* type

```
|?- [foo, goo, moo].
```

Sometimes it may be useful to enter a few clauses directly from the a terminal to test something or other. In this case you must type the command

```
|?- [user].
```

and hit return. The prompt

```
|
```

will appear to indicate that the interpreter is waiting for data. To exit the entry mode type *Control D*, which we'll signify by writing ^D. For example, to enter the two statements p(a, b) and q(X, Y) :- p(X, Y) type the following statements.

```
|?- [user].
| p(a, b).
| q(X, Y) :- p(X, Y).
|^D
```

Listing Clauses

To list the clauses of a Prolog program type the command

```
|?- listing.
```

To list only clauses beginning with predicate p type the command

```
|?- listing(p).
```

To list clauses beginning with predicates p, q, and r type the command

```
|?- listing([p, q, r]).
```

Using Unix Commands

To execute UNIX commands from SICStus Prolog, first load the system library package with the command

```
|?- use_module(library(system)).
```

This goal can be automatically loaded and executed by placing the following command in the *.sicstusrc* file.

```
:- use_module(library(system)).
```

Then UNIX commands can be executed using the *system* predicate. For example, to edit the file named filename with the vi editor, type

```
|?- system('vi filename').
```

Tracing Note

To interactively trace each step of a computation type the trace command.

```
|?- trace.
```

Now the execution of any goal will stop after each step. The names of the computation steps are from the set {call, exit, redo, fail}. To continue the computation you must react in one of several ways. For example, here are some of the options that are available.

To "creep" to the next step hit return.

To "leap" to the end of the computation, type *l* and hit return.

To list the menu of available options type *h* and hit return.

You can switch off tracing by typing the notrace command.

```
|?- notrace.
```

Spying Note

It is usually not necessary to creep through every execution step of a program. Spy-points make it possible to stop the execution at predicates of your choice. For example, to stop the execution at each use of the predicate *p*, type the goal

```
|?- spy p.
```

You can set more than one spy-point. For example, if you want to set spy-points for predicates *p*, *q*, and *r*, type the goal

```
|?- spy [p, q, r].
```

Now you can "leap" between uses of spy-points or you can still creep from step to step. To "leap" to the next use of a spy-point, type *l* and hit return.

You can remove spy-points too. For example, to remove *p* as a spy-point, type the nospy goal

```
|?- nospy p.
```

To remove *p*, *q*, and *r* as spy-points type the goal

```
|?- nospy [p, q, r].
```

To remove all spy-points type the goal

```
|?- nospyall.
```

2

Beginning Experiments

This chapter contains some simple experiments that are designed to introduce some of the basic ideas of programming in Prolog.

2.1 Variables, Predicates, and Clauses

In this experiment we'll see how to represent variables, predicates, clauses, and goals. We'll also introduce the computation rule used to execute Prolog programs.

Variables

A variable may be represented by a string of characters made up of letters or digits or the underscore symbol _, and that begins with either an uppercase letter or _. For example, the following strings denote variables:

X, Input_list, Answer, _,

A variable name that begins with the underscore symbol represents an unspecified (or anonymous) variable.

Predicates

A *predicate* is a relation. In Prolog the name of a predicate is an alphanumeric string of characters (including _) that begins with a lowercase letter. For example, we used the predicate "par" in the introductory example to denote the "is parent of" relation. The number of arguments that a predicate has is called the *arity* of the predicate. For example, par has arity 2. If a predicate q has arity n, then we sometimes write q/n to denote this fact. For example, par/2 means that par is a predicate of arity 2.

An expression consisting of a predicate applied to arguments is called an *atomic formula* Atomic formulas are the building blocks of Prolog programs. For example, the following expressions are atomic formulas.

```
p(a, b, X),
capital_of(salem, oregon).
```

Clauses

The power of Prolog comes from its ability to process clauses. A *clause* is a statement taking one of the forms

```
head.
```

or

```
head :- body.
```

where *head* is an atomic formula and *body* is a sequence of atomic formulas separated by commas. For example, the following statements are clauses.

```
capital–of(salem, oregon).
q(b).
p(X) :- q(X), r(X), s(X).
```

The last clause has head p(X) and its body consists of the atomic formulas q(X), r(X), and s(X).

Now let's look at the meaning that Prolog gives to clauses. The meaning of a clause of the form

```
head.
```

is that head is true. A clause of the form

```
head :- body.
```

has a declarative meaning and a procedural meaning. The declarative meaning is that the head is true if all of the atomic formulas in the body are true. The procedural meaning is that for head to succeed, each atomic formula in the body must succeed. For example, suppose we have the clause

```
p(X) :- q(X), r(X), s(X).
```

From the declarative point of view this means that

```
for all X, p(X) is true if q(X) and r(X) and s(X) are true.
```

From the procedural point of view this means that for all X, p(X) succeeds if q(X), r(X), and s(X) succeed.

Or Clauses

Prolog also allows us to represent the "or" operation in two different ways. For example, suppose that we have the following two Prolog clauses to define the parentOf relation in terms of the motherOf and fatherOf relations.

```
parentOf(X, Y) :- motherOf(X, Y).
parentOf(X, Y) :- fatherOf(X, Y).
```

We can think of the two clauses as having the following form.

$$c \text{ :- } a.$$
$$c\text{:- } b.$$

From a logical viewpoint, these two clauses represent the conjunction of two conditionals of the following form.

$$(a \rightarrow c) \wedge (b \rightarrow c).$$

Now recall that the following equivalence holds.

$$(a \rightarrow c) \wedge (b \rightarrow c) \equiv a \vee b \rightarrow c.$$

The right side of the equivalence can be represented in Prolog as the following or-clause, where the semi-colon denotes disjunction.

$$c \text{ :- } a;b.$$

For example, the original two clauses that we used to define the parentOf relation can also be expressed as the following or-clause.

```
parentOf(X, Y) :- motherOf(X, Y) ; fatherOf(X, Y).
```

Experiments to Perform

1. Input the program consisting of the two clauses p(a) and p(b). Then ask the following questions:

```
| ?- p(a).
| ?- p(b).
| ?- p(c).
```

Use backtracking to find all possible answers to the following question.

```
|?- p(X).
```

2. Input the single clause q(_). Then ask the following questions:

```
|?- q(a).
|?- q(b).
|?- q(c).
```

Describe what happens when you try to find all possible answers to the following question.

```
|?- q(X).
```

3. Input the following three clauses.

```
r(a).
r(b).
s(X) :- r(X).
```

Now ask the following questions:

```
|?- s(a).
|?- s(b).
|?- s(c).
```

Use backtracking to find all possible answers to the following questions.

```
|?- s(A).
|?- r(A).
```

4. Verify that the conjunction of clauses with the same head can be represented by a single clause using the "or" construction by doing the following tests. For each input the given data and then ask the question

```
|?- p(a).
```

Perform each test separately with only the given clauses as input.

a.
```
p(b).
p(a) :- p(b).
p(a) :- p(c).
```

b.
```
p(c).
p(a) :- p(b).
p(a) :- p(c).
```

c.
```
p(b).
p(a) :- p(b) ; p(c).
```

d.
```
p(c).
p(a) :- p(b) ; p(c).
```

2.2 Equality, Unification, and Computation

Most of us will agree that any object is equal to itself. For example, b is equal to b, and p(c) is equal to p(c). We might call this "syntactic equality." It's the most basic kind of equality and Prolog represents it with the following symbol.

==

For example try out the following goals.

```
|?- b == b.
|?- p(a) == p(a).
|?- p(X) == p(b).
|?- 5 == 5.
|?- 2 + 3 == 1 + 4.
```

The expressions 2 + 3 and 1 + 4 are not syntactically equal, but they both denote the number 5. This might be called "numerical" or "semantic equality." Prolog represents this equality with the following symbol.

=:=

For example try out the following goals.

```
|?- b =:= b.
|?- p(a) =:= p(a).
|?- p(X) =:= p(b).
|?- 5 =:= 5.
|?- 2 + 3 =:= 1 + 4.
```

What about expressions like p(X) and p(b)? They are not syntactically equal and they are not semantically equal. But if we consider X to be a variable, then we can say p(X) and p(b) are equal under the assumption that X stands for b. This is an example of "unification" and it is a basic ingredient in the matching process used for computation in logic programming.

Unification

Unification is the process of matching two expressions by attempting to construct a set of bindings for the variables so that when the bindings are applied to the two expressions, they become syntactically equal. Unification is

used as part of Prolog's computation rule. The following symbol is used for unification within a program.

=

If two expressions can be unified, then Prolog will return with corresponding bindings for any variables that occur in the expressions. For example, try out the following tests.

```
|?- b = b.
|?- p(a) = p(a).
|?- p(X) = p(b).
|?- 5 = 5.
|?- 2 + 3 = 1 + 4.
```

Computation

A Prolog program executes goals, where a *goal* is the body of a clause. In other words, a goal is one or more atomic formulas separated by commas. The atomic formulas in a goal are called *subgoals*. For example, the following expression is a goal consisting of two subgoals.

```
|?- par(X, james), par(Y, X).
```

The execution of a goal proceeds by unifying the subgoals with heads of clauses. The search for a matching head starts by examining clauses at the beginning of the program and proceeds linearly through the clauses. If there are two or more subgoals, then they are executed from left to right. A subgoal is true in two cases:

1. It matches a fact (i.e., the head of a bodyless clause).
2. It matches the head of a clause with a body and when the matching substitution is applied to the body, each subgoal of the body is true.

A goal is true if there is a substitution that when applied to its subgoals makes each subgoal true. For example, suppose we have the following goal for the introductory program example.

```
|?- par(X, james), par(Y, X).
```

This goal is true because there is a substitution {X=ruth, Y=katherine} that

when applied to the two subgoals gives the following subgoals, both of which are true.

```
par(ruth, james),  par(katherine, ruth).
```

Experiments to Perform

1. Try out some unification experiments like the following. First find the answers by hand. Then check your answers with Prolog.

```
|?- p(X) = p(a).
|?- p(X, f(Y)) = p(a, Z).
|?- p(X, a, Y) = p(b, Y, Z).
|?- p(X, f(Y, a), Y) = p(f(a, b), V, Z).
|?- p(f(X, g(Y)), Y) = p(f(g(a), Z), b).
```

2. An algorithm that tries to match terms is called a unification algorithm. These algorithms have an "occurs check" that stops the process if an attempt is made to unify a variable X with a non-variable term in which X occurs. For example, X and p(X) do not unify. However, most versions of Prolog do not implement the occurs check to save processing time. Try the following tests to see whether Prolog implements the occurs check.

```
|?- p(X) = p(g(X)).
|?- p(f(a, X)) = p(X).
|?- f(X) = X.
|?- [a|X] = X.
```

3. The international standard ISO Prolog has a predicate for unification with the occurs check. The name of the predicate is

```
unify_with_occurs_check.
```

In SICStus Prolog the predicate is in the "terms" library. So the following command must be executed first.

```
|?- use_module(library(terms)).
```

For example, to unify p(X) and p(a) we type the goal

```
|?- unify_with_occurs_check(p(X), p(a)).
```

which returns X = a. But the goal

```
|?- unify_with_occurs_check(p(X), p(g(X))).
```

returns no. Try out the predicate with the examples given in the preceding experiments. Compare the results. To simplify typing input the following definition.

```
u(X, Y) :- unify_with_occurs_check(X, Y).
```

2.3 Numeric Computations

Prolog has a built-in predicate "is" that is used to evaluate numerical expressions. The predicate is infix with a variable on the left and a numerical expression on the right. For example, try out the following goals.

```
|?- X is 5 + 7.
|?- X is 5 - 4 + 2.
|?- X is 5 * 45.
|?- X is log(2.7).
|?- X is exp(1).
|?- X is 12 mod 5.
```

The expression on the right must be able to be evaluated. For example, try out the goal

```
|?- X is Y + 1.
```

Now try out the goal

```
|?- Y is 5, X is Y + 1.
```

SICStus Prolog has a rich set of numerical operations that includes all of the ISO operations:

Binary operators

+, –, *, /, //, rem, mod, sin, cos, atan.

Unary operators

+, –, abs, ceiling, floor, float, truncate, round, exp, sqrt, log.

Numeric Comparison. Numerical expressions can be compared with binary comparison operators. The six operators are given as follows:

=:=, =\=, <, >, =<, >=.

For example, try out the following goals.

```
|?- 5 < 6 + 2.
|?- 5 =< 6 + 2.
|?- 5 =:= 6 - 1.
```

Experiments to Perform

1. Test each of the numeric binary, unary, and comparison operations.
2. If we don't want to type goals of the form "X is expression" we can define a predicate to do the job. For example, a predicate to evaluate and print a numeric expression can be defined as follows:

   ```
   eval(X) :- A is X, write(A).
   ```

   Try it out on several expressions. For example, try the goal

   ```
   |?- eval(sin(5)*sin(5) + cos(5)*cos(5)).
   ```

2.4 Type Checking

Prolog has several built-in predicates for type checking. For example, try out the following goals.

```
|?- integer(25).
|?- integer(25.0).
```

The ISO type checking predicates are listed as follows.

var, nonvar, integer, float, number, atom, atomic, compound.

We can create our own type checkers too. For example, suppose we let nat(X) mean that X is a natural number. We can define nat as follows.

```
nat(X) :- integer(X), X >= 0.
```

Experiments to Perform

1. Test each of the type checker predicates.

2. Construct a type checker for each of the following sets.

a. The non-negative numbers.

b. The even integers.

c. {0, 1, 2, 3,4, 5, 6, 7, 8}.

3. We can solve for any of the three variables in X + Y = Z if the other two arguments are given as follows:

```
sum(X, Y, Z) :- nonvar(X), nonvar(Y), Z is X + Y.
sum(X, Y, Z) :- nonvar(X), nonvar(Z), Y is Z - X.
sum(X, Y, Z) :- nonvar(Z), nonvar(Y), X is Z - Y.
```

a. Check it out.

b. Write a solver for the linear equation A*X + B = 0. Let the predicate linear(A, B, X) return the root X of the equation.

2.5 Family Trees

In this experiment we'll continue working with a family tree by examining a few of the many family relations. To keep things short and concise, let

p(X, Y) mean that X is a parent of Y,

and let

g(X, Y) mean that X is a grandparent of Y.

Experiments to Perform

1. Enter the following program into a file.

```
p(a, b).
p(a, c).
p(a, d).
p(b, e).
p(b, f).
p(c, g).
p(d, h).
p(e, i).
p(g, j).
p(h, k).
g(X, Y) :- p(X, Z), p(Z, Y).
```

a. Draw a graph to represent the "is a parent of" relation and orient the graph so that parents are above their children.

b. Load the program and then find all possible answers to each of the following goals.

```
|?- p(a, e).
|?- p(X, e).
|?- p(a, X).
|?- g(a, T).
|?- g(M, e).
|?- g(U, V).
```

For each of the preceding goals and answers write down (1) an informal description of the goal and (2) an informal description of the answer in terms of parents and grandparents.

2. Continue the experiment by adding definitions for the following relationships to the program and then testing them.

a. ch(X, Y) means that X is a child of Y.

b. gch(X, Y) means that X is a grandchild of Y.

3. Let sib(X, Y) mean that "X is a sibling of Y." We can define sib as follows by using the inequality operation.

```
sib(X, Y) :- p(Z, X), p(Z, Y), X \== Y.
```

Continue the experiment by adding this definition to the program and then testing it. Find definitions for the following relationships and then test them.

a. co(X, Y) means that X is a cousin of Y, which means that their parents are siblings.

b. sco(X, Y) means that X is a second cousin of Y, which means that their parents are cousins.

4. Construct a Prolog program for a real family with facts of the form parent(a, b), male(x), and female(y).

a. Define and test the relations son, daughter, mother, father, brother, sister, grandmother, grandfather, grandson, and granddaughter.

b. Define and test the relations aunt, uncle, niece, nephew, and the maternal and fraternal versions of grandmother and grandfather.

2.6 Interactive Reading and Writing

In this experiment we'll construct a simple interactive Prolog program. Since interactive programs need to read and write, we'll discuss writing to the screen and reading from the keyboard. Try out the following goals to get familiar with the "write" predicate.

```
|?- write('hello world').
|?- write(hello), write(' '), write(world).
|?- write(hello), tab(10), write(world).
|?- write(hello), nl, write(world).
|?- write(hello world).
|?- write(X).
|?- write(x).
```

The operation nl means "start a new line". The operation tab(10) means "tab 10 spaces". Be sure to place single quotes around text if it contains spaces, punctuation marks, etc. If a single quote is part of the text, then write two single quotes.

It is easy to read a Prolog term from the keyboard as long as the term ends with a period followed by a return. Try out the following goals to get the familiar with the "read" predicate.

```
|?- read(X).
|?- read(yes).
|?- read(hello).
```

Note that read(yes) and read(hello) will only succeed if the terms typed are yes and hello, respectively. Similarly, any read statement with a nonvariable argument will succeed only if that argument is typed.

Now we're in position to give an example of a simple interactive program. This example allows a user to ask for the name of the capital of a state. We'll only use a partially complete knowledge base consisting of the following six facts.

```
capital(salem, oregon).
capital(olympia, washington).
capital(boise, idaho).
capital(juneau, alaska).
capital(honolulu, hawaii).
capital(sacramento, california).
```

The program begins by asking the user to type a state. Then it reads the input, gets the capital from the knowledge base, and writes out the answer. Here is the definition.

```
start :- write('For what state do you want to know the capital?'), nl,
        write('Type a state in lowercase followed by a period.'), nl,
        read(A),
        capital(B, A),
        write(B), write(' is the capital of '), write(A), write('.'), nl.
```

Here is a typical interactive session, where bold face printing indicates typing by the user.

```
|?- start.
For what state do you want to know the capital?
Type a state in lowercase followed by a period.'
|: oregon.
salem is the capital of oregon.
```

Suppose now that we want to modify the program to find either the state of a capital or the capital of a state. We can do this by changing the written text appropriately and then defining a predicate to find either the capital of a state or the state of a capital. Here is the code.

```
start :-
        write('What state"s capital or capital"s state do you wish to know?'), nl,
        write('Type a state or a capital in lowercase followed by a period.'), nl,
        read(A),
        process(A).

process(A) :- capital(B, A), output(B, A).
process(A) :- capital(A, B), output(A, B).

output(X, Y) :-   write(X), write(' is the capital of '), write(Y), write('.'), nl.
```

Experiments to Perform

1. Put the sample data and the modified program in a file. Then make the following tests.
   a. Make three tests by typing cities and three tests by typing states.

b. Test the program by typing an uppercase letter followed by a period. Trace the computation to see what happens.

2. Write an interactive program to find information in a knowledge base of your choice. Use a database of at least ten elements. If you can't think of one, here are a couple of examples:

a. (Chemistry) Use some facts that associate each chemical element with its notation. For example, here are three facts about elements.

```
element(iron, fe).
element(hydrogen, h).
element(helium, he).
```

b. (Language Translation) Use some facts that associate words of two different languages. For example, here are three facts that relate Spanish to English.

```
translate(adios, goodby).
translate(bueno, good).
translate(porque, because).
```

2.7 Adding New Clauses

In this experiment we'll see how to add new clauses (i.e., with new predicate names) to the program by using the backtracking feature of Prolog. We'll introduce the idea with the familiar family tree example. Assume that the following facts have been input from a file named *familyTree*.

```
p(a, b).
p(a, c).
p(a, d).
p(b, e).
p(b, f).
p(c, g).
p(d, h).
p(e, i).
p(g, j).
p(h, k).
```

Suppose that we want to add all possible sibling relationships to the file. We'll assume that siblings are people who share a common parent and no one is a self-sibling. In other words, we'll use the definition

```
sib(X, Y) :- p(Z, X), p(Z, Y), X \== Y.
```

To add all possible siblings to the program we can type the following goals

```
|?- p(Z, X), p(Z, Y), X \== Y, assertz(sib(X, Y)), fail.
```

The predicate "assertz" adds a clause to the program putting it after any other clauses whose heads have the same predicate name. The goal "fail" causes backtracking to occur, which in turn causes a search for new siblings to be asserted. We can list the predicates p and sib to see that they have indeed been added to the program as follows:

```
|?- listing([p, sib]).
```

Suppose that we want to save the data for predicates p and sib in a file named x. We can do this by opening x for writing with the "tell" predicate, listing the data that we want to go into the file, and then closing the file with the "told" predicate. Here is the goal.

```
|?- tell(x), listing([p, sib]), told.
```

Experiments to Perform

1. Verify the examples given by starting with a file that contains the "p" facts and ending with a file that contains the "p" and "sib" predicates. Do a trace to observe how siblings are found by backtracking.
2. Use the same technique to add all grandchild facts to the file, where grandChild(X, Y) means that X is a grandchild of Y.

2.8 Modifying Clauses

In this experiment we'll see how to modify clauses by declaring their predicates to be dynamic. We'll introduce the idea with the familiar family tree example. Assume that the following facts have been loaded into the program from a file named *familyTree*.

```
p(a, b).
p(a, c).
p(a, d).
```

```
p(b, e).
p(b, f).
p(c, g).
p(d, h).
p(e, i).
p(g, j).
p(h, k).
sib(X, Y) :- p(Z, X), p(Z, Y), X \== Y.
```

Suppose that we want to add all possible sibling relationships to the file. We might try the following goal.

```
|?- sib(X, Y), assertz(sib(X, Y)), fail.
```

This will not work because sib is a predicate that is already used in the "static" file familyTree. If we wish to modify the predicates of a file we must declare the predicates to be "dynamic." We can do this by placing the following declaration at the beginning of the predicates that we want to be dynamic.

```
:- dynamic p/2, sib/2.
p(a, b).
p(a, c).
p(a, d).
p(b, e).
p(b, f).
p(c, g).
p(d, h).
p(e, i).
p(g, j).
p(h, k).
sib(X, Y) :- p(Z, X), p(Z, Y), X \== Y.
```

Now we can add all the sibling relations to the program as follows:

```
|?- sib(X, Y), assertz(sib(X, Y)), fail.
```

The goal "fail" causes backtracking to occur, which in turn causes a search for new siblings to be asserted. We can list the predicates p and sib to see that they have indeed been added to the program as follows:

```
|?- listing([p, sib]).
```

Suppose that we want to save the data for predicates p and sib in a file named x. We can do this by opening x for writing, listing the data that we want to go into the file, and then closing the file. Here is the goal.

```
|?- tell(x), listing([p, sib]), told.
```

If we want the predicates in file x to be dynamic then we must put the appropriate dynamic declaration at the beginning of the file. We can do this as follows.

```
|?- tell(x), write(':- dynamic p/2, sib/2.'), listing([p, sib]), told.
```

Experiments to Perform

1. Verify the examples given by starting with a file that contains the given dynamic declaration of p and sib together with the "p" facts and the definition of the sib predicate.

2. Use the same technique to add all grandchild facts to the file. In other words, place the definition for grandChild(X, Y) into the file and declare grandChild/2 to be a dynamic predicate.

2.9 Deleting Clauses

In this experiment we'll see how to delete clauses from the program. We'll introduce the idea with the familiar family tree example. In previous experiments we've seen that backtracking is a powerful tool for finding information. But sometimes we get more information than we want or need.

For example, if b and c are siblings, then backtracking produces sib(b, c) as well as sib(c, b). We can delete unwanted clauses from the program with the retract and abolish predicates. For example, suppose that we have the following clauses in the program.

```
p(b, a).
p(a, c).
p(b, d).
p(d, f).
p(e, b).
```

```
p(c, g).
p(f, h).
p(c, i).
```

We can add all siblings to the program with the following goal.

```
|?- p(X, Y), p(X, Z), Y \== Z, assertz(sib(Y, Z)), fail.
```

Now a listing of the program will produce the following clauses.

```
p(b, a).
p(a, c).
p(b, d).
p(d, f).
p(e, b).
p(c, g).
p(f, h).
p(c, i).
sib(a, d).
sib(d, a).
sib(g, i).
sib(i, g).
```

Since sib is a symmetric relation, the backtracking has produced some repetition that we may not want. Suppose we want to keep just one sib clause for each pair of siblings. The following goal will do the job.

```
|?- sib(X,Y), sib(Y,X), retract(sib(X,Y)), fail.
```

The retract predicate deletes the first occurrence of the clause in its argument. As we have already seen, the "fail" predicate automatically forces backtracking to look for other unwanted clauses. Now a listing of the clauses will produce

```
p(b, a).
p(a, c).
p(b, d).
p(d, f).
p(e, b).
p(c, g).
```

```
p(f, h).
p(c, i).
sib(a, d).
sib(g, i).
```

Note: The retract predicate works only for dynamic clauses. So we can't retract any of the p clauses of our example.

The "abolish" predicate can be used to delete clauses that are static or dynamic. But it deletes all clauses that have a head that matches the given predicate.
For example, the goal

```
|?- abolish([p, sib]).
```

will delete all p clauses and sib clauses in the program, whatever their arity. For example, the goal

```
|?- abolish(p/3).
```

will only abolish p clauses that take three arguments.

Experiments to Perform

1. Start with a family tree (your own or an example) consisting of parent relations in the program. Make sure that the tree is large enough to have several cousin relationships. Next, write a cousin predicate and use it to add all cousin relations to the program. Then experiment with the retract operation by retracting cousin(X, Y) if cousin(Y, X) is in the program.
2. Test the abolish operation on predicates that are dynamic, predicates that are static, and predicates with the same head name, but with different arities.

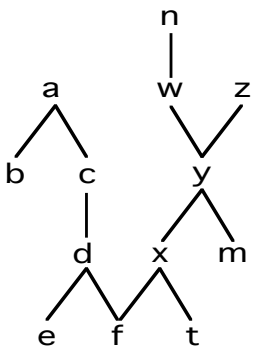
3

Recursive Techniques

This chapter introduces some recursive techniques to explore problems. The use of recursion makes it possible to write very short programs with a high degree of accuracy. This allows one to spend most of the time thinking about the experiment rather than the programming. Each problem includes an experiment that uses the trace and spy debugging tools to observe how recursive computations work. All the remaining sections contain problems that use recursion.

3.1 The Ancestor Problem

An ancestor is a person from whom one is descended. We'll write a predicate for the ancestor relation, where ancestor(A, B) means that A is an ancestor of B. If we assume that the knowledge base contains parent relations, then we can easily define the ancestor relation. Clearly a parent is an ancestor of a child, and any ancestor of a parent is also an ancestor of the child. For example, suppose we have the following graph representing some parent relations, where the orientation is such that parents are directly above their children



We can represent the graph in Prolog as the following list of parent facts, where p(X, Y) means that X is a parent of Y.

```
p(a, b).
p(a, c).
p(c, d).
p(d, e).
p(d, f).
p(n, w).
p(w, y).
p(z, y).
p(y, x).
p(y, m).
p(x, f).
p(x, t).
```

Who are the ancestors of e? From observation we can see that they are d, c, and a. Our observation started by looking for the parents of a and then looking for their ancestors. This provides us with the basis case (an ancestor is a parent) and the recursive case (an ancestor of a parent is an ancestor) for a definition of the ancestor predicate.

```
ancestor(A, B) :- p(A, B).                    % a parent is an ancestor
ancestor(A, B) :- p(X, B), ancestor(A, X).% ancestor of parent is ancestor
```

Suppose we type the following goal and then continue backtracking as long as possible.

```
|?- ancestor(A, e).
```

The results with backtracking should be: A = d; A = c; A = a; no.

Experiments to Perform

1. Implement the ancestor program along with the given knowledge base. Then perform the following tests and discuss the results in each case.

   a. Try several different goals, and backtrack as much as possible. Include goals to find the largest list of ancestors, and goals to find no ancestors.

b. Try goals with a variable in either or both argument positions and backtrack as much as possible. Are there any interesting results?

2. (*Tracing and Spying*) Do each of the following tracing tests on the ancestor program. In parts (b) and (c) you can observe the recursive calls to the ancestor predicate.

a. |?- ancestor(a, e).

b. |?- trace, ancestor(a, e).

c. |?- spy ancestor, ancestor(a, e).
(Use l to leap to each use of ancestor.)

d. |?- nospy ancestor, spy p, ancestor(a, e).
(Use l to leap to each use of p.)

3. Modify the definition of the ancestor predicate as indicated in each of the following cases. Test each new definition and discuss the results.

a. Swap the two clauses so that the recursive clause is listed before the basis clause.

b. Interchange the two predicates in the body of the recursive clause to obtain the following recursive clause.

```
ancestor(A, B) :- ancestor(A, X), p(X, B).
```

4. Modify the ancestor predicate to return the number of generations between the two people. For example the goal

```
|?- ancestor(a, e, N).
```

will return N = 3. Test your predicate on several examples, including cases where either or both of the first two arguments are variables. Try backtracking and observe the results.

3.2 Writing and Summing

Suppose that for any natural number n we wish to write out the sequence of consecutive numbers from 0 to n. To discover a program to do the job we can make a couple of observations. First, we can observe that if the input is 0, then we should write out 0. On the other hand, if n > 0, then the task will be accomplished if we write out the sequence from 0 to n – 1 and then write out n. These observations provide us with the basis case and the recursive case for a predicate "seq" to do the job. For example, the goal

```
|?- seq(3).
```

should print out the sequence 0 1 2 3. A definition for the seq predicate can be written as follows, where we've included a check to see whether the input is a natural number. We'll write out the sequence as a column of numbers.

```
seq(0) :- write(0), nl.
seq(N) :-  nat(N),              % check to see if N is a natural number
           M is N – 1, seq(M), % write the sequence 0 to N – 1
           write(N), nl.        % write N

nat(X) :- integer(X), X >= 0.   % type checker for natural numbers.
```

For another example with a similar style of programming, suppose that for any natural number n we want to compute the sum $0 + 1 + \cdots + n$. You may recall that this sum has a simple formula given by $n(n + 1)/2$. But we're trying to do some examples of recursive programming. So we'll forget about the formula for now and try to write a recursive predicate to do the job. We can make some observations about the sum to help construct the program. First, we can observe that if the input is 0, then we should return 0. On the other hand, if $n > 0$, then the task will be accomplished if we can calculate the sum $0 + 1 + \cdots + (n – 1)$ and then add n to the result. These observations provide us with the basis case and the recursive case for a predicate "sum" to do the job. For example, the goal

```
|?- sum(3, Out).
```

should return Out = 6. A definition for the sum predicate can be written as follows

```
sum(0, 0).
sum(N, S) :- nat(N), K is N - 1, sum(K, T), S is T + N.
```

Experiments to Perform

1. (*Tracing and Spying*) Implement the program for the seq predicate and try it out on several different numbers. Then do each of the following tracing tests. In parts (b) and (c) you can observe the recursive calls to the seq predicate.
   - **a.** |?- seq(3).
   - **b.** |?- trace, seq(3).
   - **c.** |?- spy seq, seq(3).
     (Use l to leap to each use of seq.)

d. |?- nospy seq, spy nat, seq(3).
(Use l to leap to each use of nat.)

e. |?- trace, seq(3).
(Try out g to see "ancestor" goals at various points.)

2. Make each of the following modifications to the definition of the seq predicate.

a. Interchange the two seq clauses so that the basis case comes after the recursive case. Test the new definition to see whether anything happens. Why or why not?

b. Starting with the modification to the seq predicate made in part (a), delete the call to nat(N) in the recursive clause. Test the new definition to see whether anything happens. Why or why not?

3. Modify the original definition of the seq predicate so that numbers are printed in reverse order. For example, the goal

|?- seq(3).

should print the sequence 3 2 1 0.

4. (*Tracing and Spying*) Implement the program for the sum predicate and try it out on several different numbers. Then do each of the following tracing tests. In parts (b) and (c) you can observe the recursive calls to the sum predicate.

a. |?- sum(3, X).

b. |?- trace, sum(3, X).

c. |?- spy sum, sum(3, X).
(Use l to leap to each use of sum.)

d. |?- nospy sum, spy is, sum(3, X).
(Use l to leap to each use of is.)

e. |?- nospy is, sum(3, X).
(Try out g to see "ancestor" goals at various points.)

f. |?- trace, sum(N, 2).
Explain why the answer is no.

5. Make each of the following modifications to the definition of the sum predicate.

a. Interchange the two sum clauses so that the basis case comes after the recursive case. Test the new definition to see whether anything happens. Why or why not?

b. Starting with the modification to the sum predicate made in part (a), delete the call to nat(N) in the recursive clause. Test the new definition to see whether anything happens. Why or why not?

c. The order of formulas in the body of a clause is important because subgoals are executed from left to right. To see this, reorder the body of the second clause to obtain the following program.

```
sum(0, 0).
sum(N, S) :- nat(N), sum(K, T), K is N - 1, S is T + N.
```

Now trace the goal sum(3, X) to see how computation proceeds.

3.3 Switching Pays

Suppose there is a lottery in which the winner will be chosen from among a set of three numbers $\{x, y, z\}$. We choose one of the three numbers, say x. Later, after the winning number has been drawn, but not yet made public, we are given the additional information that one of the other numbers, say y, is not a winner. Then we are given the opportunity to switch our choice from x to z. What should we do? We should switch.

To see this, notice that once we pick a number, the probability that we did not pick the winner is 2/3. In other words, it is more likely that one of the other two numbers is a winner. So when we are given one of the other numbers and told that it is not the winner, it follows that the remaining other number has probability 2/3 of being the winner. So go ahead and switch.

We can write an experiment to test the claim by using a random number generator. The SICStus random number package can be loaded by placing the following statement within your program.

```
:- use_module(library(random)).
```

The random predicate returns a random integer within a specified range. For example, suppose we type the goal

```
| ?- random(5, 88, Out).
```

The variable Out will be instantiated to a random integer in the range

$$5 \leq \text{Out} < 88.$$

Now we're in position to make a definition for the trial predicate, which will perform one trial of the experiment.

```
trial :-
    random(1, 4, Guess),          %  guess a number from the set {1, 2, 3}
    random(1, 4, Winner),    %  pick a winner from the set {1, 2, 3}
    check(Guess, Winner).

check(A, B) :- A = B, write(A), write(B), write(' Gave away a winner.').
check(A, B) :- write(A), write(B), write(' Switching paid off.').
```

To test the theory we need to observe what happens over many trials. Here is a recursively defined predicate to run one or more trials.

```
trials(0) :- write(' Done with trials.'), nl.
trials(N) :-
    N >= 1,
    trial,
    K is N - 1,
    trials(K).
```

Experiments to Perform

1. Test the claim that switching pays by doing several trials. Switching should pay off about two thirds of the time. So make several sets of trials with sizes that are divisible by 3. For example, execute goals like trials(3), trials(6), trials(9), and so on. And do each of these several times. Make a table to record the statistics.

2. (*Tracing and Spying*) Do each of the following tracing tests on the trials program. In parts (b) and (c) you can observe the recursive calls to the trials predicate.

   a. |?- trials(3).

   b. |?- trace, trials(3).

   c. |?- spy trials, trials(3).
   (Use l to leap to each use of trials.)

   d. |?- nospy trials, spy random, trials(3).
   (Use l to leap to each use of random.)

3. Another way to see that switching is the best policy is to modify the problem to a set of 50 numbers and a 50-number lottery. If we pick a number, then the probability that we did not pick a winner ii 49/50. Later we are told that 48 of the remaining numbers are not winners, but we are given the chance to keep the number we picked or switch and

choose the remaining number. What should we do? We should switch because the chance that the remaining number is the winner is 49/50.

Modify the trial predicate for a 50 number lottery and do several trials to confirm that switching pays off almost all of the time.

4. Write a definition for a predicate "rands" to call the random number generator one or more times to write out random numbers from a given interval. For example, rands(A, B, N) means that

random(A, B, Out), write(Out)

should be executed N times. Use the trials predicate as a prototype for your definition. Test your definition on several different intervals and for several different repetitions.

3.4 Inductively Defined Sets

A set S that is inductively defined if it can be described by naming some specific elements that are in S (the basis case) and giving one or more rules to construct new elements of S from existing elements of S (the induction case). It is also assumed that S contains only the elements constructed by these cases. For example, let S be defined as follows.

Basis: $2 \in S$.
Induction: If $x \in S$ then $x + 3 \in S$.

If we let inS(x) stand for a test to see whether $x \in S$, then we can define the inS predicate in Prolog as follows.

```
inS(2).
inS(X) :- Y is X - 3, Y >=2, inS(Y).
```

We can form inductive sets from any operation we wish if we have some basis to start from. Such a set together with its operations forms a special kind of algebra called an induction algebra. For example, suppose that we have the induction algebra $\langle A; g, e \rangle$ where g is a unary operator on A and $e \in A$ such that

$$A = \{e, g(e), g(g(e)), ..., g^n(e), ... \}.$$

Since the set is infinite we can't store it anywhere. But we can certainly

compute with its elements. Let the predicate inA(X) mean that X ∈ *A*. So we can represent the fact that e is a constant of A by writing the Prolog fact

inA(e).

Since g is a unary operation on A, we know that g(X) ∈ A if X ∈ A. In other words, we know that inA(g(X)) is true if inA(X) is true. In Prolog this becomes the recursive clause

inA(g(X)) :- inA(X).

So we have the following recursive definition for inA.

```
inA(e).
inA(g(X)) :- inA(X).
```

Now we can test for any element of A. For example, to see whether g(g(e)) is an element of A we can type the goal

```
|?- inA(g(g(e))).
```

For this algebra, we can list "all" the elements of A (if we have the time) by typing the following goal and then forcing the system to continually backtrack.

```
|?- inA(X).
```

Experiments to Perform

1. Write down a description of the set S from the example. Then implement the definition for the inS predicate. Perform some tests to see whether the inS predicate does indeed test for membership in the set S that you described. Try some arguments that are not in S. Try some non integers too.

2. (*Tracing and Spying*) Do each of the following tracing tests on the inS predicate. In parts (b) and (c) you can observe the recursive calls to the inS predicate.

   a. |?- inS(5).
   b. |?- trace, inS(5).
   c. |?- spy inS, inS(11).
   (Use l to leap to each use of trials.)

d. Find out what happens when a variable is used as an argument to the inS predicate. What about backtracking? Why or why not?

3. For each of the following inductive definitions, write down a set description of A. Then give a Prolog definition for the inA predicate, where inA(x) means that x ∈ A. Test your definition against your set description of A and perform tests similar to those of Experiment (2).

a. Basis: 3 ∈ A.
Induction: if x ∈ A then 2x + 1 ∈ A.

b. Basis: 0 ∈ A.
Induction: if x ∈ A then x + 4 ∈ A and x + 9 ∈ A.

4. Implement the definition for the inA predicate to test for elements of A in the sample induction algebra ⟨A; g, e⟩. Perform each of the following tests.

a. Test the definition on several terms, some of which are in A and some that are not.

b. Do a trace of the goal |?- inA(g(g(e))).

c. See if you can generate all the elements of the induction algebra with the following goal and backtracking (if you have an infinite amount of time).

```
|?- inA(X).
```

5. Suppose that we have the induction algebra ⟨T; *f*, a, b⟩ where *f* is a unary operation on T and a, b ∈ T. Write a definition for the inT predicate, where inT(X) means that X ∈ T. Then perform each of the following tests.

a. Test the definition on several terms, some of which are in T and some that are not.

b. Do a trace of the goal |?- inT(f(f(b))).

c. See if you can generate all the elements of the induction algebra with the following goal and backtracking (if you have an infinite amount of time).

```
|?- inT(X).
```

Can backtracking generate all the elements of T? Why or why not?

6. Suppose that we have the induction algebra ⟨U; h, a, b⟩ where h is a binary operation on U and a and b are constants in U. Write a definition

for the inT predicate, where inU(X) means X ∈ U. For example, the expressions h(a, b), h(h(b, a), b) and h(h(a, a), h(b, b)) represent elements of U. Then perform each of the following tests.

a. Test the definition on several terms, some of which are in U and some that are not.

b. Do a trace of the goal |?- inU(h(h(b, a), b)).

c. See if you can generate all the elements of the induction algebra with the following goal and backtracking (if you have an infinite amount of time).

```
|?- inU(X).
```

Can backtracking generate all the elements of U? Why or why not?

4

Logic

This chapter contains experiments that use Prolog to explore some basic ideas about logic.

4.1 Negation and Inference Rules

In this experiment we'll examine relationships between Prolog and logic by examining negation and some inference rules. We know from logic that if A is a statement, then the negation of the A is the statement ¬ A, which is opposite in truth value from that of A. The modus ponens inference rule says that from the two statements A and A → B we can infer B. The hypothetical syllogism inference rule says that from the two statements A → B and B → C we can infer A → C.

In SICStus Prolog the "not" operation is denoted by the two-character string \+. For example, the expression \+ p should be written in place of the expression not p. As the experiments will show, the not operation in Prolog is better thought of as meaning "not provable from the program facts".

Experiments to Perform

1. (Negation) Enter the following fact into the program.

```
p(a).
```

Try the following two questions.

```
| ?- p(a).
| ?- \+ p(a).
```

Now try the following two questions.

```
|?- p(b).
|?- \+ p(b).
```

What does this experiment have to do with the formal idea of logical negation?

2. (If-then-else) We want to experiment with the statement "If c(X) then p(X) else q(X)" where c, p, and q are predicates. First, enter the following facts in the program:

```
c(b).
p(X) :- c(X).
q(X) :- \+ c(X).
```

Try out the following two questions:

```
|?- p(a).
|?- q(a).
```

Now ask the two questions:

```
|?- p(b).
|?- q(b).
```

What does this experiment have to do with logical if-then-else?

3. (If-then-else) For each of the following experiments enter the given data in the program and then ask the following questions.

```
|?- s(a).
|?- s(b).
|?- s(c).
|?- s(X).
```

Use backtracking whenever possible to find alternative answers. Each experiment is an attempt to define s(X) as the statement, "if p(X) then q(X) else r(X)." Compare the two tests and discuss whether one definition is preferable to the other.

a.
```
p(a).
p(b).
q(a).
r(b).
s(X):- p(X), q(X).
s(X):- \+ p(X), r(X).
```

b.
```
p(a).
p(b).
q(a).
r(b).
s(X):- p(X), q(X).
s(X):- r(X).
```

4. (Modus ponens) Enter the following two facts into the program.

```
p(a).
q(X) :- p(X).
```

Ask the question

```
|?- q(a).
```

Discuss how this experiment is related to the modus ponens inference rule.

5. (Hypothetical syllogism) Enter the following three facts into the Prolog program:

```
r(X) :- q(X).
q(X) :- p(X).
p(a).
```

Now ask the question

```
|?- r(a).
```

Discuss how this experiment is related to the hypothetical syllogism inference rule.

4.2 The Blocks World

This experiment introduces the blocks world, which consists of a set of building blocks arranged in some way. We'll examine some questions about the blocks world and see how negation can be used to describe some typical predicates.

To describe the fact that one block is on top of another block we'll use the "on" predicate. For example, to say that block *a* is sitting on top of block *b*, we'll write the Prolog fact

on(*a, b*).

Once we have a blocks world described by a set of these facts, there are many questions that can be asked and operations that can be performed. We'll describe two such predicates. The "blocked" predicate tells us whether there is a block above a given block. It can be defined by the following Prolog clause.

```
blocked(X) :- on(Y, X).
```

For example, if on(a, b) is in the program, then the following goal will succeed.

```
|?- blocked(b).
```

The "onTop" predicate tells us whether a block is on the top of a stack of one or more blocks. It can be defined by the following Prolog clause.

```
onTop(X) :- \+ blocked(X).
```

For example, if on(a, b) is in the program, then the following goal will succeed.

```
|?- onTop(a).
```

Experiments to Perform

1. Load the following program into the program.

```
on(a, b).
blocked(X) :- on(Y, X).
onTop(X) :- \+ blocked(X).
```

For each of the following goals, give the answer that you might expect, and then execute the goal to see whether Prolog agrees with your answer.

```
|?- onTop(a).
|?- onTop(b).
|?- onTop(X).
|?- \+ onTop(a).
|?- \+ onTop(b).
|?- \+ onTop(X).
```

2. Let's extend the blocks program to allow us to move blocks around. For example, we'll let move(*a, b*) mean that block *a* is to be moved onto block *b* if they are both on top. To accomplish this we'll need to test whether the two blocks are on top and, if so, to retract one clause and to assert another clause. We'll modify the program so that on(*a*, [ ]) means that *a* is on the bottom. Here is the extended program, which includes a set of blocks to use in testing.

```
:- dynamic on/2.
on(a, b).
on(b, c).
on(d, [ ]).
```

```
on(c, [ ]).
onTop(X) :- \+ blocked(X).
blocked(X) :- on(Y, X).
move(X, [ ]) :- onTop(X), retract(on(X, Y)), assertz(on(X, [ ])).
move(X, Y) :- onTop(X), onTop(Y), retract(on(X, Z)), assertz(on(X, Y)).
```

a. Draw a picture of the blocks as given. Test the "move" predicate by constructing a single stack of the blocks so that from top to bottom they read *a*, *b*, *c*, *d*. Draw a picture of the blocks after each call of move. List the program of facts to confirm the result.

b. Implement and test the bottom predicate to find the bottom of a stack of blocks. For example, if a stack of blocks has *a* on top and *b* on the bottom, then the query bottom(*a*, *X*) returns *X* = *b*.

c. Implement and test the move_ordered predicate, where move_ordered(*a, b*) means move the stack of blocks with *a* on the top to the top of block *b* such that the blocks of the stack keep their same relative order. For example, if *abc* and *de* represent two stacks of blocks with tops *a* and *d*, then move_ordered(*a*, *d*) would move *abc* onto the top of *d* resulting in *abcde*.

d. Implement and test the move_reversed predicate, where move_reversed(*a, b*) means move the stack of blocks with *a* on the top to the top of block *b* such that the blocks of the stack are reversed in relative order. For example, if *abc* and *de* represent two stacks of blocks with tops *a* and *d*, then move_reversed(*a*, *d*) would move *cba* onto the top of *d* resulting in *cbade*.

4.3 Verifying Arguments in First-Order Logic

In this experiment we'll see if we can get Prolog to verify an argument in first-order predicate calculus. We'll start with an example. Suppose we're given the following argument in English:

> Every dog likes people or hates cats or both. Rover is a dog. Rover loves cats. Therefore some dog likes people.

Let $d(x)$ mean that x is a dog, $lp(x)$ mean that x likes people, $lc(x)$ mean that x loves cats, and let a = Rover. Then the argument can be formalized as follows:

$$\forall x\,(d(x) \rightarrow lp(x) \vee \neg lc(x)),\ d(a),\ lc(a) \vdash \exists x\,(d(x) \wedge lp(x)).$$

A formal proof of this statement can be given as follows:

Proof:

1.	$\forall x\,(d(x) \rightarrow lp(x) \wedge \neg lc(x))$	*P*
2.	$d(a)$	*P*
3.	$lc(a)$	*P*
4.	$d(a) \rightarrow lp(a) \wedge \neg lc(a)$	1, UI
5.	$lp(a) \wedge \neg lc(a)$	2, 4, MP
6.	$lp(a)$	3, 5, DS
7.	$d(a) \wedge lp(a)$	2, 6, Conj
8.	$\exists x\,(d(x) \wedge lp(x))$	7, EG
	QED	1, 2, 3, 8, CP.

To see whether Prolog will verify such an argument, we need to give Prolog the three premises on lines 1, 2, and 3 and then give Prolog the goal on line 8. Before we can write line 1 as a Prolog statement, we need to do some rewriting:

$$\begin{aligned} \forall x\,(d(x) \rightarrow lp(x) \wedge \neg lc(x)) &\equiv \forall x\,(\neg d(x) \vee lp(x) \wedge \neg lc(x)) \\ &\equiv \forall x\,(\neg (d(x) \wedge lc(x)) \wedge lp(x)) \\ &\equiv \forall x\,(d(x) \wedge lc(x) \rightarrow lp(x)). \end{aligned}$$

The latter wff can now be written in Prolog as the following clause.

```
lp(X) :- d(X), lc(X).
```

The second and third premises can be written as the following Prolog facts.

```
d(a).
lc(a).
```

The goal corresponding to the conclusion on line 8 can be written in Prolog as follows:

```
|?- d(X), lp(X).
```

Prolog returns the answer X = a, yes.

Note: If there is an existentially quantified wff as a premise, Prolog can't handle it. So we need to use the result of EI as the premise. For example, if $\exists x\,p(x)$ is a premise, then use the premise p(*c*) for a new constant *c*.

Experiments to Perform

1. Use Prolog to verify the following argument:

Every committee member is rich and famous. Some committee members are old. Therefore some committee members are old and famous.

2. Use Prolog to verify the following argument:

No human beings are quadrupeds. All men are human beings. Therefore no man is a quadruped.

3. Use Prolog to verify the following argument:

Some freshmen like all sophomores. No freshman likes any junior. Therefore no sophomore is a junior.

4.4 Equality Axioms

Suppose we have the following two axioms for an equality theory.

EA Axiom: $\forall x\ (x = x)$
EE Axiom: $t = u \rightarrow p(\ldots t \ldots) \rightarrow p(\ldots u \ldots)$.

The EA Axiom together with universal instantiation implies the reflexive property $t = t$ for any term t. We'll call this result the EA Axiom too.

EA Axiom: $t = t$

We can use these axioms to prove the following symmetric and transitive properties for terms:

$$t = u \rightarrow u = t$$
$$t = u \rightarrow u = v \rightarrow t = v.$$

For example, we have the following proof of the symmetric property:

Proof:

1.	$t = u$	*P*
2.	$t = t$	EA Axiom
3.	$t = u \rightarrow t = t \rightarrow u = t$	EE Axiom
4.	$t = u \rightarrow t = t$	1, 2, Conj
5.	$u = t$	3, 4, MP
6.	$t = u \rightarrow u = t$	1, 5, CP

QED.

To see if Prolog validates this argument, enter the following facts in the program, where *e*(*a, b*) means $a = b$.

```
e(t, u).
e(X, X).
e(U, T) :- e(T, U), e(T, T).
```

Now ask the following question.

| ?- *e*(*u, t*).

Experiments to Perform

1. Why does this Prolog experiment verify the six line proof of symmetry?

2. Consider the following proof of transitivity.

1.	$t = u$	*P*
2.	$u = v$	*P*
3.	$u = v \rightarrow t = u \rightarrow t = v$	EE Axiom
4.	$u = v \rightarrow t = u$	1, 2, Conj
5.	$t = v$	3, 4, MP
6.	$t = u \rightarrow u = v \rightarrow t = v$	1, 2, 5, CP

QED.

a. Construct a Prolog experiment to verify this proof.

b. Explain how your Prolog experiment verifies the transitivity proof.

4.5 SLD-Resolution

Let's introduce some terminology regarding computations of logic programs. "SLD-resolution" is name of the inference rule—which is a special case of the resolution inference rule—that is used to perform computation in logic programs. The rule is always applied to a goal atom and a clause whose head unifies with the goal atom. An SLD-derivation is a sequence of applications of the SLD-resolution rule. An SLD-refutation is a finite SLD-derivation that ends with the empty clause. As an example, let's consider the logic program *P* consisting of the following clauses.

```
p(a, b).
p(a, c).
p(b, d).
p(c, e).
g(X, Y) :- p(X, Z), p(Z, Y).
```

Let *G* be the goal

| ?- g(a, W).

Here's an SLD-refutation for $P \cup \{G\}$. To start, we can resolve the goal with

the 5th clause using the most general unifier σ_1 = {X/a, W/Y}. The result is the goal

|?- p(a, Z), p(Z, Y).

We can resolve the leftmost atom of this goal with the 1st clause using the most general unifier σ_2 = {Z/b}. The result is the goal

|?- p(b, Y).

We can resolve this goal with the 3rd clause using the most general unifier σ_3 = {Y/d}. The result is the empty clause

□.

So we have an SLD-refutation of $P \cup \{G\}$. Here the refutation as a formal proof.

1.	p(a, b)	*P*
2.	p(a, c)	*P*
3.	p(b, d)	*P*
4.	p(c, e)	*P*
5.	g(X, Y) :- p(X, Z), p(Z, Y)	*P*
6.	|?-g(a, W)	*P* (the goal)
7.	|?-p(a, Z), p(Z, Y)	5, 6, *R*, σ_1 = {X/a, W/Y}.
8.	|?-p(b, Y)	1, 7, *R*, σ_2 = {Z/b}
9.	□	3, 8, *R*, σ_3 = {Y/d}.

Applying the composition of the three unifiers to the given goal atom, we obtain g(a, W)$\sigma_1\sigma_2\sigma_3$ = g(a, d), which is a logical consequence of the program *P*.

Experiments to Perform

1. For the given program, trace the execution of the goal *G*. Then observe the relationships between the calls of the trace and the clauses on lines 6, 7, 8, and 9 of the SLD-refutation.

2. For the given example there is another SLD-refutation. All SLD-refutations of are contained as paths in the computation tree for $P \cup \{G\}$.

   a. Draw a picture of the computation tree (i.e., the SLD-tree) for the program and goal $P \cup \{G\}$. Be sure to label each success leaf of the tree with the computed answer.

b. Verify your answers by tracing the execution of the goal and by using backtracking.

3. We can define addition and multiplication of natural numbers as follows, where the natural numbers are represented as 0, *s*(0), *s*(*s*(0)), and so on.

```
add(0, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).

mult(0, Y, 0).
mult(s(X), Y, Z) :- mult(X, Y, W), add(Y, W, Z).
```

For example, the goal

|?- add(*s*(*s*(0)), *s*(0), *A*).

computes the value of the expression 2 + 1. Similarly, the goal

|?- mult(*s*(*s*(0)), *s*(*s*(0)), *A*).

computes the value of the expression 2*2.

a. Construct an SLD-refutation for the goal |?- add(*s*(*s*(0)), *s*(0), *A*). Trace the execution of the goal to help verify that the refutation is correct.

b. Construct an SLD-refutation for the goal |?- mult(*s*(*s*(0)), *s*(*s*(0)), *A*). Trace the execution of the goal to help verify that the refutation is correct.

4.6 The Cut Operation

The "cut" is an operation that is used to cut off backtracking when it is not wanted. For example, consider the following logical statement, where s, p, q, and r are all predicates.

s(X) = if p(X) then q(X) else r(X).

Suppose we implement the statement in Prolog with the following two clauses.

```
s(X) :- p(X), q(X).
s(X) :- r(X).
```

Now suppose we have a goal of the form

```
|?- s(X).
```

If p(X) is true, then we want q(X) to execute and we want the truth value of s(X) to be that of q(X). If p(X) is false, then we want r(X) to execute and we want the truth value of s(X) to be that of r(X). Everything looks fine. But suppose that p(X) is true and q(X) returns a no answer. Then backtracking may cause the computation to proceed to the second clause, which gives s(X) the value of r(X) when at the same time p(X) is true.

We can avoid this situation by inserting a cut, symbolized by the exclamation mark, !, after p(X) in the first clause to obtain the following implementation of "if p(X) then q(X) else r(X)."

```
s(X) :- p(X), !, q(X).
s(X) :- r(X).
```

The cut is a nullary predicate that always succeeds as a goal and if it is encountered on backtracking, then it causes the backtracking to skip all clauses beginning with s(X).

Now suppose that the goal s(X) is given and p(X) is true. Then the cut will succeed and q(X) will be executed. If q(X) succeeds, then s(X) will be given its value as desired. And if q(X) fails, then backtracking will reach the cut and the cut will force backtracking to skip the second s clause. Thus s(X) fails too, just as q(X) did, as we desired.

Let's consider the logic program consisting of the following clauses. Notice that the two clauses with "b" at the head form an "if d then e else f" statement.

```
a :- b, c.
a :- d, f.
b :- d, !, e.
b :- f.
c.
d :- g, h.
d.
e :- i, j.
e :- k.
f.
```

Experiments to Perform

1. Draw a picture of the SLD-tree for the goal |?- *a*.

2. Draw a closed curve around the subtree that will be pruned if the cut is

encountered on backtracking.

3. Test the program with different goals to see that the cut performs as expected. For example, to confirm the results of parts (1) and (2) above, try out the goal

```
|?- a.
```

Try some other goals too. For example, try the following goals.

```
|?- b.    and    |?- c.
```

Use trace to see what is happening. Make some observations.

NOTE: In sicstus a goal atom will cause the computation to abort with an existence error message if the predicate name of the atom does not appear as the head of any clause in the program. To get a failure rather than an abort, replace each occurrence of *g, h, i, j,* and *k* in the program with $g(a)$, $h(a)$, $i(a)$, $j(a)$, $k(a)$. Then add the following facts to the program: $g(b)$, $h(b)$, $i(b)$, $j(b)$, $k(b)$. Now a goal such as $g(a)$ will fail but the computation will continue.

5

List Structures

Lists are fundamental structures for representing information. In this chapter we'll see how to represent and access lists and we'll see a variety of ways that lists are used to represent sets and binary trees.

5.1 List and String Notation

A list is represented in Prolog by separating the elements with commas and placing square brackets at the ends. The empty list is denoted by [ ]. Some examples of lists are

```
[a, b, c]
[17, 13, 11, 7, 5, 3, 2]
[orange, orange, apple, banana]
[X, f(A), hello, 3, -9.4]
[a, [a, a], [a, [a, a]], a]
[the, [ ], has, nothing, in, it]
```

A list can be constructed or taken apart by using the symbol | to separate the head from the tail. For example, try the following goal.

```
| ?- [H | T] = [a, b, c].
```

Notice that H represents the head and T represents the tail of the list. We can also construct lists. For example, try the following goal.

```
| ?- A = a, B = [b, c], C = [A | B].
```

Next we'll look at two goals that should confirm something that we know

about lists. Try out the following two goals only after you have thought about what the answer should be.

```
|?- [a] = [H|T].
|?- [ ] = [H|T].
```

If a string of characters is enclosed within double quotes, then Prolog treats the string internally as a list of natural numbers representing the ASCII codes for the characters in the string. For example, the string "abcd" and the list [97, 98, 99, 100] are treated the same by Prolog. So strings can be processed as lists. Try out the following goals to get the idea.

```
| ?- "abcd"=X.
|?- "abcd" = [97, 98, 99, 100].
|?- "abcd" = [H|T].
|?- "X*(Y+Z)" = [H|T].
```

If a string of characters is not enclosed within double quotes, then Prolog treats the string as an atom that can't be manipulated. Try out the following goals to get the idea.

```
| ?- abcd = X.
| ?- abcd = [H|T].
| ?- a+b = X.
| ?- 'a+b' = X.
| ?- 'a+b' = [H|T].
```

Although atoms can't be manipulated, there is a Prolog predicate that transforms between atoms and strings. The predicate "name" does the job. Try out the following goals to get the idea.

```
| ?- name(abc, X).
| ?- name(X, [97, 98, 99]).
| ?- name(abc, [97, 98, 99]).
| ?- name(a+b, X).
| ?- name('a+b',X).
```

Experiments to Perform

1. Try out a few more examples to get used to the notation for lists. For example, try out the following goals.

```
|?- [A, B | C] = [1, 2, 3, 4, 5].
|?- [[a, b], [c, d]] = [[A, B] | C].
```

2. Suppose that we want to construct a test to see whether a list consists of an even number of a's. If we let p be the test predicate, then we should get yes answers on goals such as

```
|?- p([ ]).
|?- p([a, a]).
```

And we should get no answers to goals like

```
|?- p([b]).
|?- p([a, a, a]).
```

We can define p as follows.

```
p([ ]).
p([a, a|X]) :- p(X).
```

a. Test the definition on several lists.

b. To see what is going on, trace the goal |?- p([a, a]).

c. Generate the set of terms for which p is true with the following goal and backtracking.

```
|?- p(A).
```

3. Find a definition for the predicate q where q(t) is true if and only if t is a list that contains an odd number of b's.

a. Test the definition on several lists.

b. To see what is going on, trace the goal |?- q([b, b, b]).

c. Generate the set of terms for which q is true with the following goal and backtracking.

```
|?- q(B).
```

5.2 Sets and Bags of Solutions to a Query

In this experiment we'll examine some powerful tools for constructing sets and bags.

The Setof Predicate

When we work with sets in a programming language we normally work with lists that have no repeated elements. The Prolog language treats sets in this manner too and it uses list notation for sets. So, for example, when we think about a set like

{a, b, c},

we must use the notation

[a, b, c]

to represent it in Prolog.

We can often describe a set by describing a property (i.e., a predicate) that the elements of the set must satisfy. For example, if we let p(a, b) mean that "a is a parent of b," then the set S of all people who are parents can be described as

$$S = \{x \mid p(x, y) \text{ for some } y\}$$

In terms of formal logic, the statement "p(x, y) for some y" can be written as $\exists y\ p(x, y)$. So we can describe S as

$$S = \{x \mid \exists y\ p(x, y)\}.$$

Prolog provides a useful tool called the "setof" predicate that can be quite useful in calculating sets. We'll introduce it with some examples. Suppose we put the following four facts in the program.

```
p(a, b).
p(a, c).
p(b, d).
p(c, e).
```

Suppose that we want to find the set S of all letters that appear in the first argument of one of the facts. In other words, we want to calculate the set S that we defined above. Since there are only four facts in the program it is easy to see that

$$S = \{x \mid \exists y\ p(x, y)\} = \{a, b, c\}.$$

Let's try to use the setof predicate to find S. In Prolog the notation Y^ means "there exists Y." We can compute the set S with the following goal.

```
| ?- setof(X, Y^p(X, Y), S).
```

This goal returns S = [a, b, c].

For another example, suppose that we want to calculate the set of second arguments that have the letter *a* as a first argument of p. In other words, we

want to calculate the set

$$S = \{x \mid p(a, x)\} = \{b, c\}.$$

We can compute the set S with the following goal.

```
| ?- setof(X, p(a, X), S).
```

This goal returns S = [b, c].

For another example, suppose that we want to construct the set of "grandparent" relationships from p. If we let g(x, y) mean that x is a grandparent of y, then we can describe the set S of these relations as the following set.

$$S = \{g(x, y) \mid \exists z\ (p(x, z) \wedge p(z, y))\}.$$

We can compute the set S with the following goal.

```
| ?- setof(g(X, Y), Z^(p(X, Z), p(Z, Y)), S).
```

This goal returns S = [g(a, d), g(a, e)].

The Bagof Predicate

A bag (or multiset) is like a set except that repetitions of elements can occur in a bag. Bags are represented in Prolog as lists. The bagof predicate works just like the setof predicate except that repeated elements are kept. For example, to find the bag B of all letters that appear in the first argument of one of the facts in the preceding example we type the following goal.

```
| ?- bagof(X, Y^p(X, Y), B).
```

This goal returns B = [a, a, b, c].

The Findall Predicate

The findall predicate is similar to the bagof predicate. It finds a bag of elements by examining all solutions to a query without any need to use quantifiers. For example, to find the bag B of all letters that appear in the first argument of one of the facts in the preceding example we type the following goal.

```
| ?- findall(X, p(X, Y), B).
```

This goal returns B = [a, a, b, c].

Experiments to Perform

1. Try the sample goals for the given set of facts. Then, for each of the following cases, use the setof predicate to construct and test a goal to compute the given set.

a. C = {y | ∃x p(x, y)} (children).

b. R = {c(y, x) | p(x, y)} (child relations).

2. Modify the sample goals by eliminating quantified variables. For example, in place of the goal

```
|?- setof(X, Y^p(X, Y), S).
```

use the goal

```
|?- setof(X, p(X, Y), S).
```

Also try the goal

```
|?- setof(X, p(X, _), S).
```

In each test, be sure to backtrack as much as possible. Try to explain the difference in the outcomes.

3. The setof predicate returns no instead of the empty set. For example, using the given set of facts we have {x | ∃y p(x, a)} = ∅.

a. Do a test to verify this fact.

b. Define a new predicate "newsetof" that calls the setof predicate. But if the setof predicate returns no, then newsetof returns [ ]. Test newsetof on 3 nonempty sets and 3 empty sets.

4. Enter the following set of facts in the Prolog database.

```
p(0, 0, 0, 49).
p(0, 0, 1, 143).
p(0, 1, 0, 78).
p(0, 1, 1, 398).
p(1, 0, 0, 87).
p(1, 0, 1, 398).
p(1, 1, 0, 49).
```

p(1, 1, 1, 374).

For each of the following cases, use setof, bagof, or findall to construct the indicated set or bag.

a. The set of all facts of the form p(0, X, Y, Z) for some X, Y, and Z.

b. The bag of all numbers N such that p(X, Y, 0, N) for some X and Y.

c. The bag of all numbers N such that p(X, Y, Z, N) for some X, Y, and Z.

d. The bag of all pairs [Y, Z] such that p(X, Y, Z, N) for some X and N.

e. The set of all facts of the form p(X, Y, Z, N) where N < 100.

5.3 List Membership and Set Operations

Most versions of Prolog have a predicate to test whether an element is a member of a list. In SICStus Prolog the "member" predicate resides in the lists library. But it is easy to define in any case. Here is one definition that we can use.

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

For example, the goal

```
|?- member(a, [b, a, c, b]).
```

succeeds with the answer yes. We can use the backtracking feature of Prolog to generate all elements in a list. For example, the goal

```
|?- member(X, [b, a, c, b]).
```

succeeds with X = b. With backtracking the other elements of the list will also be found.

Subset and Equality Relations

Since a set is a list with no repeated elements, we can use the member predicate to test for membership in a set too. This gives us a tool that can be used to construct many useful operations that involve sets.

We'll start with the subset relation. Recall that $A \subseteq B$ if and only if every element of A is an element of B. It follows from this that the empty set is a subset of every set. This gives us the basis case for the following recursive definition of the subset predicate.

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

For example, the following goal returns yes.

```
|?- subset([a, b], [c, a, d, b]).
```

Now we can use the subset relation as a tool to define a test for equality of sets. Recall that equality of sets can be described in terms of subset as follows.

A = B if and only if A ⊆ B and B ⊆ A.

So we can easily define the equal predicate as follows.

```
equal(A, B) :- subset(A, B), subset(B, A).
```

For example, the following goal returns yes.

```
|?- equal([a, b, c], [c, a, b]).
```

Union, Intersection, and Difference of Sets

Now we'll look at a few standard operations to construct sets. We'll start with the union operation. The union S of two sets A and B is the set

$$S = A \cup B = \{x \mid x \in A \text{ or } x \in B\}.$$

We might try to define the union operation by using the setof predicate as the follows.

```
union(A, B, S) :- setof(X, (member(X, A); member(X, B)), S).
```

A problem with this definition, which we can observe with testing, is that the setof predicate will return a set only if the statement in the middle argument is true for some X. Otherwise a no answer will be returned. So a goal like

```
|?- union([ ], [ ], S).
```

will return a no answer instead of giving S the value [ ]. This problem is easily fixed by adding the fact union([ ], [ ], [ ]) to the definition. So the complete definition looks like the following.

```
union([ ], [ ], [ ]).
union(A, B, S) :- setof(X, (member(X, A); member(X, B)), S).
```

Next, we'll consider the intersection S of two sets A and B, which is the set

$$S = A \cap B = \{x \mid x \in A \text{ and } x \in B\}.$$

Suppose we attempt to define the intersection operation using the setof predicate as follows.

```
intersect(A, B, S) :- setof(X, (member(X, A), member(X, B)), S).
```

In this case, the middle statement of the setof predicate will be false if either of both of A and B are [ ]. So we need to add cases to handle this problem as follows to get a proper definition.

```
intersect([ ], _, [ ]).
intersect(_, [ ], [ ]).
intersect(A, B, S) :- setof(X, (member(X, A), member(X, B)), S).
```

Now we'll describe two other set operations and leave the Prolog definitions as experiments. The *difference* between A and B (also called the *relative complement* of B in A) is the set of elements in A that are not in B, and it is denoted by $A - B$. A natural extension of the difference $A - B$ is the *symmetric difference* of sets A and B, which is the union of $A - B$ with $B - A$ and is denoted by $A \oplus B$. For example, if $A = \{a, b, c\}$ and $B = \{c, d\}$, then we have

$$A - B = \{a, b\} \quad \text{and} \quad A \oplus B = \{\text{a, b, d}\}.$$

Experiments to Perform

1. Check each of the following statements by hand and then use "member" to confirm your answers:

a. x ∈ {a, b}. **b.** x ∈ {a, x}. **c.** a ∈ {a}.
d. ∅ ∈ {a, b}. **e.** ∅ ∈ ∅. **f.** ∅ ∈ {∅}.
g. {a, b} ∈ {a, b, c}. **h.** {a, b} ∈ {{a, b}, b, c}.

2. Use "member" and Prolog's backtracking feature to generate all elements in each of the following sets:

a. {a, b}. **b.** ∅. **c.** {∅}. **d.** {{a, b}, b, c}. **e.** {{a}, b, {{∅}}}.

3. Test the "member" predicate to see how it behaves when variables are placed in various positions in a goal.

a. Both arguments are variables. For example, try out the goal

```
|?- member(A, S).
```

Use backtracking to discover any patterns.

b. The first argument is a constant and the second is a variable. For example, try out the goal

```
|?- member(a, S).
```

Use backtracking to discover any patterns.

4. Test the "subset" and "equal" predicates. Use the trace command on one test for subset. Try a few tests with variables in different argument positions.

5. Perform each of the following tests for the union and intersection operations.

a. Test the definitions of the union and intersect predicates on a variety of sets, including [ ].

b. Remove the basis cases in the definitions of union and intersect predicates and observe what happens when either or both arguments is [ ].

c. Try out the following alternative definition for union that uses two predicates from the SICStus Prolog lists library.

```
union(A, B, S) :- append(A, B, R), remove_duplicates(R, S).
```

6. Define and test the operation "minus" for set difference. For example, the goal

```
|?- minus([a, b, c], [a, c, d], S).
```

should return S = [b]. Be sure to check the definition with [ ] as either or both of the first two arguments.

7. Recall that the symmetric difference of two sets A and B can be written in two ways:

$$A \oplus B = (A - B) \cup (B - A)$$

$$A \oplus B = (A \cup B) - (A \cap B).$$

Define two predicates, sym1 and sym2, to test each of these properties. For example, the goal

|?- sym2([a, b, c], [b, c, d], *S*).

should return S = [a, d]. Be sure to check the definitions with [ ] as either or both of the first two arguments.

8. The following two properties of sets relate the subset operation to the operations of intersection or union.

$A \subseteq B$ if and only if $A \cap B = A$.
$A \subseteq B$ if and only if $A \cup B = B$.

Define two predicates, sub1 and sub2, to test each of these properties. For example, the goal

|?- sub1([a, b], [c, b, a]).

should return yes. Be sure to check the definitions with [ ] for either or both of the arguments.

5.4 List Operations

This experiment focuses on defining and testing a variety of operations that use lists. The emphasis will be on short recursive definitions for predicates to implement the operations. We'll give some examples and then continue with more problems in the experiments.

For example, suppose we need to find the length of a list. Since most versions of Prolog already have a length predicate for lists, we'll use the name "lngth" for our definition. The definition can be constructed once we observe that the length of an empty list is 0 and the length of a nonempty list is 1 plus the length of its tail. This gives us the basis case (the length of the empty list is 0) and the recursive case (the length of [H|T] is 1 plus the length of T) that we can use to construct a definition for the lngth predicate.

```
lngth([ ], 0).
lngth([Head|Tail], L) :- lngth (Tail, N), L is 1 + N.
```

For another example, suppose we need to construct a predicate "cat" to concatenate two lists. For example, if [h, e] and [l, l, o] are two lists, then the goal

```
|?- cat([h, e], [l, l, o], A).
```

should return A = [h, e, l, l, o]. Most Prolog implementations have a predicate to concatenate lists called the append predicate. But we'll write our own definition for the cat predicate for practice. We'll start with a few observations about the concatenation of lists. Certainly the concatenation of the empty list [ ] with any other list X is just X. For the nonempty case, it can be observed that the concatenation of a nonempty list [H | T] with any other list X takes the form [H | Y], where Y is the result of concatenating T with X. This gives us the basis and recursive cases needed for our definition of cat.

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

Experiments to Perform

1. Test cat on a few lists. For example the goal

   ```
   | ?- cat ([a, b], [c, a], A).
   ```

   should return the answer A = [a, b, c, a]. Now try the same goal with the trace command. Try a few tests with variables in different argument positions.

2. Implement the following definition for the "last" predicate, which finds the last element of a non-empty list.

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

   Test the predicate on several lists. Use the trace command on one test. Try a few tests with variables in different argument positions.

3. Construct a recursive Prolog program for the "first" predicate, which removes the rightmost element of a nonempty list. For example, the goal

   ```
   | ?- first([a, b, c], X).
   ```

   returns X = [a, b]. Test the predicate on several lists. Use the trace command on one test. Try a few tests with variables in different argument positions.

4. Construct a recursive Prolog program for the "pairs" predicate, which takes two lists of equal length and outputs a list consisting of the corresponding pairs from the two input lists. For example, the goal

```
|?- pairs([a, b, c], [1, 2, 3], X).
```

returns X = [[a, 1], [b, 2], [c, 3]]. Test the predicate on several pairs of lists. Use the trace command on one test. Try a few tests with variables in different argument positions.

5. Construct a recursive Prolog program for the "dist" predicate, which takes an element and a list and outputs a list of pairs made up by distributing the given element with each element of the list. For example, the goal

```
|?- dist(x, [a, b, c], X).
```

returns X = [[x, a], [x, b], [x, c]]. Test the predicate. Use the trace command on one test. Try a few tests with variables in different argument positions.

6. Construct a recursive Prolog program for the "prod" predicate, which takes two lists and outputs the product of the two lists. For example, the goal

```
|?- prod([1, 2], [a, b, c], X).
```

returns X = [[1, a], [1, b], [1, c], [2, a], [2, b], [2, c]]. Test the predicate. Use the trace command on one test. Try a few tests with variables in different argument positions.

7. Construct a recursive Prolog program for the "replace" predicate, which replaces all occurrences of an element in a list. For example, the goal

```
|?- replace(a, o, [b, a, n, a, n, a], X).
```

returns X = [b, o, n, o, n, o]. Test the predicate. Use the trace command on one test. Try a few tests with variables in different argument positions.

8. Construct a recursive Prolog program for the "power" predicate, which computes the set of all subsets of a finite set, represented as a list. For example, the goal

```
|?- power([a, b], X).
```

should return X = [[ ], [a], [b], [a, b]] or some other ordering of the same sets. Test the predicate. Use the trace command on one test. Try a few tests with variables in different argument positions.

9. (*Printing a List*). Suppose that we wish to construct a recursive Prolog program for the "out" predicate, which outputs a list of elements with a given number of elements on each line. For example, if L is a list with 100 items, then the goal

```
|?- out(L, 15).
```

Should print nine lines with 15 items per line and the last line will have 10 items. The following definition uses a predicate outLine(L, N, K), which prints N elements from list L and returns the list K of elements from L that remain to be printed.

```
out(L, N) :- length(L, M), M > N, outLine(L, N, K), nl, out(K, N).
out(L, N) :- length(L, M), outLine(L, M, [ ]), nl.
```

Define and test the outLine predicate. Then test the out predicate on several lists of varying sizes with a different number of items per line.

Notes: The predicate nl means "start a new line". There is a predicate tab that might be useful. For example, tab(2) means "tab 2 spaces".

6

List Applications

This chapter contains a variety of experiments, all of which use recursive techniques to explore problems that use lists.

6.1 Binary Trees

Binary trees are inherently recursive in nature. In this experiment we'll see how binary trees can be created, searched, and traversed by simple recursive algorithms. SICStus Prolog has a package of binary tree operations that can be accessed by

```
| ?- use_module(library(trees)).
```

The empty binary tree is denoted by the letter

```
t.
```

A nonempty tree is represented by an expression of the form

```
t(Root, Left, Right),
```

where Root is the root of the tree and Left and Right are the left and right subtrees, respectively. Lists can be transformed into binary trees by the predicate list_to_tree. For example, try out the following goal.

```
|?-list_to_tree([a, b, c, d, e, f], Tree).
```

We can observe the way that the tree is constructed by examining the output and drawing a picture of the tree. But we can also build traversal algorithms.

For example, here is an algorithm to traverse a binary tree in preorder and return the result as a list of nodes.

```
pre(t, [ ]).
pre(t(A, B, C), [A|X]) :- pre(B, U), pre(C, V), append(U, V, X).
```

The append predicate concatenates two lists. It is in the SICStus Prolog library of list operations, which we can access as follows.

```
| ?- use_module(library(lists)).
```

The following goal will first transform the list [a, b, c, d] into a binary tree, and then, using our preorder predicate, construct the list of elements obtained by a preorder traversal of the tree.

```
| ?- list_to_tree([a, b, c, d], T), pre(T, Out).
```

Suppose that we did not have the binary tree library available. We can easily construct a binary search tree from a list of numbers as follows, using the same notation for binary trees as above.

```
build([ ], t).
build([H|T], Tree) :- build(T, U), insert(H, U, Tree).
```

The predicate "insert" takes a number and a binary search tree and returns a new binary search tree that contains the number.

```
insert(X, t, t(X, t, t)).
insert(X, t(A, L, R), t(A, New, R)) :- X =< A, insert(X, L, New).
insert(X, t(A, L, R), t(A, L, New)) :- X > A, insert(X, R, New).
```

Experiments to Perform

1. Do some testing to see whether list_to_tree constructs a binary tree of the smallest possible depth. Test the predicate on lists of length 3, 4, 7, and 8. For each test draw a picture of the corresponding tree.
2. The expression to represent a binary tree is not very inviting. To see the information in a binary tree we can traverse it by one of the standard methods, preorder, inorder, and postorder.

   a. Test the preorder predicate "pre" as follows

   ```
   | ?- list_to_tree([a, b, c, d], T), pre(T, Out).
   ```

b. Construct and test the predicate "in" to traverse a binary tree in inorder. Use the same tree as in part (a).

c. Construct and test the predicate "post" to traverse a binary tree in inorder postorder. Use the same tree as in part (a).

3. Test the "build" and "insert" predicates on several lists. Be sure to try lists that are sorted in either direction as well as unsorted lists. E.g., [1,2,3,4], [4,3,2,1], and [3,1,4,2].

4. Construct and test a Prolog predicate "isIn" to see whether a number is in a binary search tree.

6.2 Arranging Objects

The process of arranging things seems to be part of almost every kind of endeavor. In this section we'll look at arbitrary arrangements (permutations) and specific arrangements (sorting).

Permutations

Suppose we want to generate permutations of a list. For example, suppose "perm" is a predicate such that the goal

```
| ?- perm([a, b, c], A).
```

returns the permutation A = [a, b, c], and on backtracking will return the other five permutations of [a, b, c]. We can write a definition of "perm" as follows:

```
perm([ ], [ ]).
perm([H | T], [A | B]) :- delete(A, [H | T], X), perm(X, B).
```

where "delete" is the predicate that deletes the first occurrence of an element from a list. For example, the goal

```
| ?- delete(a, [b, a, c], X).
```

returns X = [b, c]. We can write a definition of "delete" as follows:

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

Sorting

Let's write a sorting predicate for a list of numbers. We'll use the idea of sorting by insertion, where the head of the list is inserted into the sorted version of the tail of the list. For the moment, we'll assume that "insert" does the job of inserting an element into a sorted list. We'll use the name "isort" because SICStus Prolog already has its own "sort" predicate.

```
isort([]) = []
isort(h :: t) = insert(h, isort(t))
```

Now we'll translate this definition into a Prolog program.

```
isort( [ ], [ ]).
isort([H | T], S) :- isort(T, A), insert(H, A, S).
```

Of course, we can't test this definition until we write the definition for the insert predicate. This predicate inserts an element into a sorted list by comparing the element with each member of the list until it reaches a larger element or the end of the list, at which time the element is placed in the proper position. Here's a definition for the insert predicate in if-then-else form:

```
insert(a, x)  =  if x = [] then [a]
                 else if a <= head(x) then a :: x
                 else hd(x) :: insert(a, tail(x)))
                 fi
```

Here's a Prolog definition for the insert predicate.

```
insert(A, [ ], [A]).
insert(A, [H | T], [A | [H | T]]) :- A =< H.
insert(A, [H | T], [H | S]) :- insert(A, T, S).
```

Now we can test both the insert predicate and the isort predicate.

```
| ?- insert(7, [1, 4, 9, 14], X).
| ?- isort([4, 9, 3, 5, 0], X);
```

Experiments to Perform

1. Test "perm" with both arguments as atomic lists (e.g., perm([a, b], [b, a])), both as variables, and both cases where one argument is a variable and the other an atomic list. Use backtracking to observe any patterns.

2. Test "delete" with various combinations of the three arguments as variables. Use backtracking to observe any patterns.

3. Perform several tests of insert and isort and do at least one trace for each predicate.

4. What happens if we insert an element into a list that is not sorted?

5. Modify the definition of insert by replacing =< with <. Try out some tests on lists that have repeated elements to see what happens. Is one version more efficient than the other?

6. Try backtracking with both isort and insert and notice the results. Find a way to modify the definition of insert so that both isort and insert return "no" on backtracking.

7. Try out the sort predicate that comes with the system. Notice whether it removes repeated occurrences of elements. Write a second version of insert so that repeated occurrences of elements are removed.

8. (*Slow Sorting*). The following slowsort program sorts a list X of numbers by generating permutations of X until a sorted version is found.

```
slowsort(X, Y) :- perm(X, Y), sorted(Y).
sorted([ ]).
sorted([X]).
sorted([X|[Y|Z]]) :- X =< Y, sorted([Y|Z]).
perm([ ],[ ]).
perm([X|Y], [U|V]) :- delete(U, [X|Y], Z), perm(Z, V).
delete(X, [X|Y], Y).
delete(X, [Y|Z], [Y|W]) :- delete(X, Z, W).
```

a. Try out some simple examples to see whether slowsort works. For example, try the goal

```
|?- slowsort([4, 8, 2, 5], Y).
```

Do a trace of one goal to observe the order in which permutations are generated.

b. Try out slowsort with X free and Y bound and then with both X and Y free.

c. Suppose that we replace the slowsort predicate with slowsort2 defined as follows:

```
slowsort2(X, Y) :- sorted(Y), perm(X, Y).
```

The two atoms in the body of slowsort have been swapped so that sorted is called without Y bound to any list. Do some tests to find out what happens.

6.3 Simple Ciphers

This experiment involves ciphers for encoding or decoding messages. To get things started we'll construct a cipher that will encode or decode a string of text by means of a simple translation of the characters. For example, the message 'abc' translated by 5 letters becomes 'fgh'. We'll construct a predicate 'cipher' to do the job in such a way that the goal

```
|?- cipher(abc, 5, B).
```

will return the value B = fgh.

We'll assume that the messages use only the alphabet of lowercase letters a to z, which have ASCII codes 97 to 122. With this assumption the cipher is easy to write once we figure out how to wrap around the end of the alphabet. For example, to translate the letter z (i.e., 122) by 5 we need to come up with the letter e (i.e., 101). All we need to do is add the two numbers (z – 97) and 5 modulo 26. Then add 97 to the result to get back within the proper ASCII range. Since we want to be able to use any integer as a key, we need to make sure that for any integer n we have $0 \leq n \bmod 26 < 26$. Since the Sicstus mod function returns negative results if n is negative, we'll write a predicate "mod2" to implement the mod function that we need. Here is the program.

```
cipher(A, Key, B) :-
      name(A, X),
      change(X, Key, Y),
      name(B, Y).

change([ ], _, [ ]).
change([H|T], Key, [R|S]) :-
      mod2(H – 97 + Key, 26, A),
      R is A + 97,
      change(T, Key, S).

mod2(X, Y, Z) :- Z is integer(X - Y*floor(X/Y)).
```

The program can be used to encode or decode a message. For example, consider the following goals.

```
| ?- cipher(hello, 3, X).
X = khoor ?
yes

| ?- cipher(khoor, -3, X).
X = hello ?
yes
```

The cipher we've been talking about is called an *additive* cipher. The program can also be used as a tool by a cryptanalyst who doesn't know the cipher but thinks it might be an additive cipher. An additive cipher is an example of a *monoalphabetic* cipher, which is a cipher that always replaces any character of the alphabet by the same character from the cipher alphabet. In our example the two alphabets are the same and each letter is replaced by the third letter to its right.

Experiments to Perform

1. Implement the given program for an additive cipher. Perform the following tests to see whether the algorithm can be used for encoding and decoding.

   a. Perform tests on a message with two different keys. Then perform tests to decode the encoded messages. Do a test on messages that wrap around the left and the right ends of the alphabet.

   b. Make some tests on several different keys to be sure that a permutation of the cipher alphabet results. For example, try the goal

   ```
   |?- cipher(abcdefghijklmnopqrstuvwxyz, –2, X).
   ```

2. A *multiplicative* cipher is a monoalphabetic cipher that translates each letter by using a multiplier key.

   a. Modify the algorithm for an additive cipher to obtain an algorithm for a multiplicative cipher that does multiplication modulo 26. Perform tests on a message with two different keys.

   b. For a given key, will we get a permutation of the original alphabet? For example, try the following goal, and see whether the key 2 gives a permutation of the alphabet.

   ```
   |?- cipher(abcdefghijklmnopqrstuvwxyz, 2, X).
   ```

   Instead of inspecting the output X by hand to see whether we have 26

distinct letters, we can write a few lines of code to do the job. For example, we'll convert the output string back into a list, make it into a set and then let the length predicate find the number of elements in the set. We'll let distinct(X, L) mean that the string X has L distinct letters. Then we can write a simple definition for distinct as follows:

```
distinct(X, L) :-
    name(X, Y),
    setof(A, member(A, Y), Z),
    length(Z, L).
```

For example, to check whether the number 2 is a good choice for a key, we can write the goal

```
|?- cipher(abcdefghijklmnopqrstuvwxyz, 2, X), distinct(X, L).
```

which might return something like

```
L = 13,
X = acegikmoqsuwyacegikmoqsuwy.
```

This tells us that X has only 13 distinct letters, so we know that 2 is not a good choice for a key.

Find which of the 26 keys 1, ... 26 yield permutations of the letters a...z. You could execute the preceding goal 26 times, once for each of the 26 keys. But first try to write a loop to do the job automatically.

c. Which keys act as an identity (they don't change the message)? Is there always one letter that never changes no matter what the key? Do fractions work as keys? What about decoding (i.e., deciphering) a message? Do you need a new deciphering algorithm?

3. An *affine* cipher is a monoalphabetic cipher that translates each letter by using two kinds of translation. For example, let A and M be the keys for an additive and a multiplicative cipher. Then we can transform an input string by first applying the additive cipher with key A to get an intermediate result. Then apply the multiplicative cipher with key M to obtain an output string. Let "affine" be the predicate for the cipher. A typical goal might look like

```
|?- affine(hello, 3, 4, X).
```

Write a program for affine and test it thoroughly. Are there any restrictions on the value of the keys? What about decoding (i.e.,

deciphering) a message? Do you need a new deciphering algorithm?

6.4 The Birthday Problem

The "birthday problem" illustrates that some coincidences are actually probable events. This experiment is designed to reinforce this idea. For example, we know that if we choose 23 numbers (e.g., birthdays) at random out of 365 possible numbers (e.g., the days of the year), then the probability that two of the chosen numbers will be the same is 0.507. For 30 numbers the probability is 0.706, and for 40 numbers the probability is 0.891. Consider the following Prolog program to generate a list of random numbers in the interval 1 to 365.

```
birth(0, [ ]).
birth(N, [Out | T]) :-  random(1, 366, Out),
                        K is N - 1,
                        birth(K, T).
```

The random number package can be loaded by the following statement:

```
| ?- use_module(library(random)).
```

After the program has been loaded, a goal such as

```
| ?- birth(23, L).
```

will return a list L of 23 random integers in the range 1 to 365.

Experiments to Perform

1. It is hard for our eyes to find duplicates in a list of random numbers. But it is not hard to write a program that for any given list of numbers returns a list of any duplicates that occur. Let dup(L, D) mean that D is a list of duplicates that occur in the list L. For example, the goal

```
| ?- dup([2, 6, 2, 7, 2, 6, 9], D).
```

should return D = [2, 6]. Now we can let trial(N, D) mean that D is a list of duplicates that occur in a list of N random numbers in the interval 1 to 365. The definition for trial is easy.

```
trial(N, D) :- birth(N, L), dup(L, D).
```

Your task is to construct and test the predicate "dup" so that trial works as desired.

2. Test the birthday paradox by doing 10 trials for each of the following values of N. In each case, observe how close the results of the 10 trials come to the actual probabilities.

a. 23 numbers.

b. 30 numbers.

c. 40 numbers.

6.5 Predicates as Variables

In first-order predicate calculus, the only arguments allowed in a predicate are terms. If a predicate is allowed as an argument to another predicate, then the logic is of second or higher order. Prolog has a mechanism for passing predicates as arguments. The two Prolog operations that are needed to accomplish the task are

=.. and *call*.

These operations are used in a Prolog program whenever we want to process an arbitrary predicate that is passed as an argument to the program. For example, to process R(A, B), where R varies over different predicate names that have two arguments, we must write the following sequence of two statements.

```
P =.. [R, A, B], call(P)
```

For another example, to process R(A, B, C), where R varies over different predicate names that have three arguments, we must write the following sequence of two statements.

```
P =.. [R, A, B, C], call(P)
```

The following tests serve to introduce these two operations.

Experiments to Perform

1. Ask the following questions to get used to the =.. operation. The answer to the last question will be an error message.

```
| ?- A =.. [r, a, b].
| ?- A =.. [r, A, B, c].
```

```
| ?- p(a, b) =.. [A, B, C].
| ?- R = r, A =.. [R, a].
| ?- A =.. [R, a].
```

2. Enter s(a, b) into the data base. Then ask the following questions. The answer to the last question will be an error message.

```
| ?- s(a, b).
| ?- A =.. [s, a, b].
| ?- A =.. [s, a, b], call(A).
| ?- S = s, A =.. [S, a, b], call(A).
| ?- A =.. [S, a, b], call(A).
```

3. Discuss why the answers to the last question in each of the two tests are error messages.

4. (Reflexivity) Suppose we want to test whether a binary relation is reflexive. We'll let "reflex" be the predicate to do the job. For example, let *r* be the binary relation over the set {*a*, *b*, *c*} given by the following facts.

```
r(a, b).
r(b, a).
r(a, a).
```

Then the goal

```
| ?- reflex(r, [a, b, c]).
```

will return "no" since r is not reflexive. Here is a definition for reflex.

```
reflex(R, [ ]).
reflex(R, [H | T]) :- Q =.. [R, H, H], call(Q), reflex(R, T).
```

a. Enter the relation r and the program into the program and then perform the following tests for reflexivity.

```
| ?- reflex(r, [ ]).
| ?- reflex(r, [a]).
| ?- reflex(r, [b]).
| ?- reflex(r, [a, b]).
| ?- reflex(r, [a, b, c]).
```

b. Trace the execution of the following two goals to see how the computation proceeds.

```
|?- reflex(r, [a]).
|?- reflex(r, [a, b]).
```

c. Create a reflexive binary relation s over the set {a, b, c} and enter it into the program. Then perform at least the following tests for reflexivity.

```
|?- reflex(s, [a]).
|?- reflex(s, [a, b]).
|?- reflex(s, [a, b, c]).
|?- reflex(s, [a, b, c, d]).
```

6.6 Mapping Numeric Functions

The built-in numeric functions in Prolog are evaluated with the "is" predicate. For example, try out the following goals.

```
|?- X is floor(-3.1).
|?- X is ceiling(-3.1).
|?- X is truncate(-3.1).
|?- X is exp(2, 16).
|?- X is log(2, 16).
```

It is often convenient to examine a list of values for a function. For example, if f is a function and [a, b, c, d] is a list of elements in the domain of f, then we might want to examine the list [f(a), f(b), f(c), f(d)]. We'll define the predicate "mapf" to do the job for any Prolog numeric function of a single variable that is evaluated with the "is" predicate. For example, the goal

```
|?- mapf(floor, [-1.5, 2.4, 8.9], X).
```

returns X = [-2.0, 2.0, 8.0].

Here is the definition of mapf.

```
mapf(F, [ ], [ ]).
mapf(F, [H|T], [A|B]) :-  G =.. [F, H],
                          A is G,
                          mapf(F, T, B).
```

We can use the following "gen" predicate to generate an increasing sequence of numbers where each differs from its predecessor by one. For example, the goal

```
|?- gen(–2.4, 6, X).
```

returns X = [-2.4, -1.4, -0.4, 0.6, 1.6, 2.6]. Here is the definition of gen.

```
gen(_, 0, [ ]).
gen(S, N, [S|T]) :-  K is S + 1,
                     M is N – 1,
                     gen(K, M, T).
```

Now it's easy to map a function different lists. For example, to map the floor function onto the list [-2.4, -1.4, -0.4, 0.6, 1.6, 2.6], type the following goal.

```
|?- gen(-2.4, 6, X), mapf(floor, X, Y).
```

This goal returns Y = [-3.0, -2.0, -1.0, 0.0, 1.0, 2.0].

Experiments to Perform

1. Use gen and mapf to compare the following arithmetic functions over the rational numbers.

floor, ceiling, truncate, round.

How are they different? How are they alike? Find sets of rationals where they are equal/not equal.

2. Construct and test a program to define the predicate genSeq, where genSeq(S, I, N, L) means that L is a list of N numbers beginning with S such that succeeding numbers differ by interval I. Test the predicate by using it with mapf to explore values of the following functions.

a. round(X). **b.** sin(X). **c.** cos(X).

6.7 Mapping Predicates

If a predicate contains one or more input variables and an output variable, then we can map the predicate onto a list of input values to return a list of output values. We'll define a "map" predicate to do the job. For example, suppose the program contains the following facts.

```
q(a, x).
q(b, y).
q(c, z).
```

Then the goal

```
|?- map(q, [a, b, c], A).
```

will return

```
A = [x, y, z].
```

In the case of two or more input arguments, the map predicate will take as input the name of the predicate together with a list of inputs, where each input is a list of input arguments. For example, suppose the program contains the following facts.

```
p(a, b, x).
p(a, c, y).
p(b, c, z).
```

Then the goal

```
|?- map(p, [[a, b], [a, c], [b, c]], T).
```

will return T = [x, y, z].

Here is the definition for the map predicate.

```
% The basis case.
map(P, [ ], [ ]).

% The case for predicates with two or more input arguments.
map(P, [H|T], [X|R]) :-     is_list(H),
                            append(H, [X], C),
                            Q =.. [P|C],
                            call(Q),
                            map(P, T, R).

% The case for predicates with one input argument.
map(P, [H|T], [X|R]) :- Q =.. [P, H, X],
                        call(Q),
                        map(P, T, R).
```

Experiments to Perform

1. If a function is a composition of several numeric functions, then it cannot

be mapped with "mapf" because it does not have a name. So we must define it as a predicate and then use "map". For example, we know that among binary trees with n nodes, that the minimum depth of any tree is floor($\log_2$ n). Suppose that we want find out the value of the composition on the list [1, 2, 3, 4, 5, 5, 7, 8]. To do this we need to define a predicate that we can pass to the map function. For example, suppose we define

```
minDepth(N, X) :- X is floor(log(2, N)).
```

now we can map minDepth as follows

```
|?- map(minDepth, [1, 2, 3, 4, 5, 5, 7, 8], X).
```

Of course, we can use gen to generate lists of numbers. For example, if we want to map minDepth onto the list [1, 2, ..., 16], then the following goal will do the job.

```
|?- gen(1, 16, X), mapf(minDepth, X, Y).
```

Use map to test the following compositions and compare them over several ranges of values with minDepth.

a. ceiling(log(2, X)).

b. integer(log(2, X)).

c. truncate(log(2, X)).

2. The following two predicates provide alternative definitions for the mod function. The first uses the mod function provided by Prolog.

```
mod1(X, Y, Z) :- Z is X mod Y.
mod2(X, Y, Z) :- Z is integer(X - Y*floor(X/Y)).
```

Do they agree? If not, describe the differences between the two functions. Use map to test the two definitions over sets of integers. Hint: pick a modulus, say 5, and define

```
mod1_5(X, Y) :- mod1(X, 5, Y).
mod2_5(X, Y) :- mod2(X, 5, Y).
```

Then do some tests of mod1_5 and mod2_5 using gen and map. Be sure to include some negative integers in your tests. Also use different moduli.

6.8 Comparing Numeric Functions

Suppose that we want to compare two arithmetic functions over a set of

values. We'll write a predicate "comparef" to do the job. For example, to compare whether floor and truncate agree on the 5-element set

{–2.5, –1.5, –0.5, 0.5, 1.5}

we type the goal

```
|?- gen(–2.5, 5, X), comparef(floor, truncate, X, Answer).
```

The following results are returned.

```
X = [-2.5,-1.5,-0.5,0.5,1.5],
Answer = [false, false, false, true, true]
```

which indicate that floor and truncate do not agree on the set. Here is a definition of the comparef predicate.

```
comparef(F, G, [ ], [ ]).
comparef(F, G, [H|T], [true|S]) :-  A =.. [F, H], X is A,
                                    B =.. [G, H],Y is B,
                                    X =:= Y,
                                    comparef(F, G, T, S).
comparef(F, G, [H|T], [false|S]) :- comparef(F, G, T, S).

gen(_, 0, [ ]).
gen(S, N, [S|T]) :- K is S + 1, M is N - 1, gen(K, M, T).
```

Experiments to Perform

1. Use comparef to test pairs of the following functions.

floor, ceiling, truncate, round.

How are they different? How are they alike? Find sets of rationals where they are equal/not equal.

2. To compare functions that are compositions of Prolog numeric functions, we need to represent them as predicates. For example, consider the following two compositions.

```
minDepth(N, X) :- X is floor(log(2, N)).
alternateDepth(N, X) :- X is ceiling(log(2, N)).
```

But we cannot use comparef because the compositions are now predicates. But we can alter the definition of comparef to call predicates and then compare the results as follows

```
compare_cf(F, G, [ ], [ ]).
compare_cf(F, G, [H|T], [true|S]) :-  A =.. [F, H, V1], call(A),
                                      B =.. [G, H, V2], call(B),
                                      V1 =:= V2,
                                      compare_cf(F, G, T, S).
compare_cf(F, G, [H|T], [false|S]) :- compare_cf(F, G, T, S).
```

Compare the two functions on several lists to see whether they are equal. Also compare them with the following compositions.

a. integer(log(2, N)).
b. round(log(2, N))
c. truncate(log(2, N)).

6.9 Comparing Predicates

Problems can usually be solved in many different ways. So it is useful to be able to easily compare solutions. It would be nice if we had a tool to compare two predicates to see whether they agree over various domains. We'll construct a predicate "compare" to do the job. For example, suppose we have the two predicates "pop" and "quiz" defined as follows.

```
pop(N, X) :- X is floor(log(2, N)).

quiz(N, X) :- X is ceiling(log(2, N)).
```

To test the two predicates over the 6-element set {4, 5, 6, 7, 8, 9} we'll type the goal

```
|?- gen(4, 6, X), compare(pop, quiz, X, Result).
```

The goal should return

```
Result = [true, false, false, false, true, false].
```

This indicates that the two predicates agree on the set {4, 8} and they disagree on the set {5, 6, 7, 9}. Here is a definition for compare.

```
compare(F, G, X, Result) :-  map(F, X, A),
                             map(G, X, B),
```

```
compare_lists(A, B, Result).
```

The compare–lists predicate compares two lists of the same length, outputting a list of true/false values.

```
compare_lists([ ], [ ], [ ]).
compare_lists([H|S],[H|T], [true|U]) :- compare_lists(S, T, U).
compare_lists([H|S],[K|T], [false|U]) :- compare_lists(S, T, U).
```

Experiments to Perform

1. Suppose that we have two different definitions to test whether a number is even, given as "even1" and "even2" as follows.

```
even1(N, true) :- N mod 2 =:= 0.
even1(_, false).

even2(N, true) :- N =:= 2*floor(N/2).
even2(_, false).
```

Compare the two definitions over several sets of integers to see whether they agree.

2. The compare predicate returns a list of Boolean values that is not very interesting and can get cumbersome if we are testing two functions over a large set of integers. We can modify the program to return a subset of {true, false} by replacing the compare predicate with the following compare2 predicate, which uses the remove_duplicates predicate in the lists library.

```
compare2(F, G, X, Result) :-   map(F, X, A),
                               map(G, X, B),
                               compare_lists(A, B, C),
                               remove_duplicates(C, Result).
```

a. Use compare2 to test whether pop and quiz are equal over several different large ranges of integers.

b. Use compare2 to test whether even1 and even2 from Experiment 1 are equal over several different large ranges of integers.

3. Write two different definitions for a predicate to test whether a number is odd. Test your definitions to make sure that they agree on the numbers in the set {-1000, ... 1000}.

7

Languages and Expressions

This chapter contains experiments that use Prolog to explore some of the basic ideas of language parsing. We'll also explore associated semantic actions to evaluate arithmetic expressions.

7.1 Grammar and Parsing

How can we implement a parser for the language of a grammar? If the grammar is context-free and all left recursion has been removed, then it is quite easy to build a parser in Prolog.

Recall that each step in a leftmost derivation of a string consists of replacing the leftmost nonterminal of a sentential form with the right side of a production. So each nonterminal often derives only a proper substring of the given string. For this reason, each nonterminal will be associated with a predicate having two arguments, one for the given string and one to hold the rightmost portion of the string not derived by the nonterminal.

Let's do an example to help get the idea. Suppose we start with the following grammar.

$$S \rightarrow aST \mid \lambda$$
$$T \rightarrow aSb \mid c$$

Let s and t denote the predicates to be associated with nonterminals S and T, respectively. We'll represent strings as lists for ease of notation. Now we can give the Prolog definitions of s and t as follows.

```
s(X, Z) :- X = [a | R], s(R, U), t(U, Z).      % S → aST
s(X, Z) :- X = Z.                              % S → λ
t(X, Z) :- X = [a | R], s(R, U), U = [b | Z].  % T → aSb
t(X, Z) :- X = [c | Z].                        % T → c
```

A typical goal tests whether a string is in the language. For example, to see whether the string *aab* is in the language, type the goal

```
| ?- s([a, a, b], [ ]).
```

Notice that the second argument of the goal is always [ ], which represents the empty string λ. The reason for this is that the initial call to *s* asks whether there is a leftmost derivation of the entire input string. In other words, the goal asks whether there is a derivation $S \Rightarrow^{+} aab$.

Experiments to Perform

1. Test the example parser on five strings that are accepted and five strings that are not accepted. Then observe how computation takes place by tracing the goal

   ```
   | ?- s([a, a, b], [ ]).
   ```

2. Write and test a parser for each of the following grammars and observe in each case how computation takes place by tracing a goal.

   a. $S \rightarrow aSb \mid \lambda$.

   b. $S \rightarrow AB \mid abA \mid bBab$
   $A \rightarrow aA \mid \lambda$
   $B \rightarrow bB \mid c$.

7.2 A Parsing Macro

Prolog has a nice macro facility for constructing a parser for the language of a grammar. For example, suppose we have the following grammar that we used in Section 7.1.

$$S \rightarrow aST \mid \lambda$$
$$T \rightarrow aSb \mid c$$

To create a parser for the language of this grammar we simply write the following "macro" clauses.

```
s --> “a”, s, t | [ ].
t --> “a”, s, “b” | ”c”.
```

These two clauses can also be written as four separate clauses, one for each of the four productions.

```
s --> “a”, s, t.
s --> [ ].
t --> “a”, s, “b”.
t --> ”c”.
```

So nonterminals must be lowercase names, terminal letters are strings in quotes, and the empty string is represented by the empty list. The operation --> is macro operation that transforms the input clauses into other clauses that are used to do the parsing. The nonterminals are converted to predicates with two arguments. For example, to see if the string *aab* is derived by the grammar we write the following goal.

```
| ?- s(“aab”, [ ]).
```

If we are going to be doing much testing it can be tedious to always have to type the double quotes and the empty list. We can avoid this tedium by defining our own predicate to parse strings of the grammar. For example, consider the following definition for a predicate *p*:

```
p([ ]) :- s([ ], [ ]).
p(X) :- name(X, Y), s(Y, [ ]).
```

The Prolog predicate *name* is used to transform between a string X and a list *Y* of ASCII codes for the letters of *X*. Once we've added this definition to the program, we can find out whether the string *aabb* is derived by the grammar by writing the following simpler goal.

```
| ?- p(aabb).
```

NOTE: Prolog parses in a top-down left-to-right fashion. So if a grammar is left recursive, then make sure to remove the left recursion before you write the Prolog clauses.

Experiments to Perform

1. Try out the parser for the example grammar on five strings that are in the language of the grammar and five strings that are not in the language of the grammar.
2. Find out how Prolog expands the parsing macros for the sample grammar by typing the following query.

   |?- listing.

   a. Notice that the clauses in the listing use the predicate 'C', which is a system predicate defined by the single fact

   'C'([H|T], H, T).

   For example, the goal 'C'(B, c, D) will succeed if *B* is a list with head *c* and tail *D*. In other words, C'(B, c, D) is the same as the Prolog statement B = [c|D]. In section 7.1 we constructed our own parsers using clauses that included statements such as B = [c|D] to recognize a letter. Compare the clauses in the listing with the clauses for the example parser given in Section 7.1.

   b. Observe how computation takes place by tracing the goal

   |?- s("*aab*", [ ]).

   Compare this trace with the trace that you did in Experiment 1 of Section 7. 1.

3. Write a grammar and a parser for each of the following languages. Test your results on several strings.

   a. $\{a^n b^n \mid n \in N\}$.

   b. $\{a^m b^n c^{m+n} \mid m, n \in N\}$.

7.3 Programming Language Parsing

In this experiment we'll consider the problem of parsing a simple imperative programming language given by the following grammar:

$P \rightarrow S \mid ST$

$T \rightarrow ;ST \mid \varepsilon$

$S \rightarrow$ while V <> 0 do P od $\mid V := 0 \mid V :=$ succ(V) $\mid V :=$ pred(V)

$V \rightarrow$ identifier

We'll assume for this experiment that an identifier is a single uppercase letter. With this assumption, a parser for this language can be written in Prolog as follows:

```
p --> s | s , t.
t --> ";", s, t | [ ].
s --> "while", v, "<>", "0", "do", p, "od".
s --> v, ":=", "0" | v, ":=", "succ", "(", v, ")" | v, ":=", "pred", "(", v, ")".
v --> [D], {"A"=<D, D=<"Z"}.
```

For example, to parse the statement

```
while A <> 0 do A := pred(A) od
```

type the following goal:

```
|?- p("whileA<>0doA:=pred(A)od", [ ]).
```

Note that the Prolog grammar does not allow for spaces between syntactic objects. For example, the goal

```
|?- p("X:=succ(Y)", [ ]).
```

will return yes. But the following goal will return no:

```
|?- p("X  :=     succ(Y)", [ ]).
```

Experiments to Perform

1. Test the parser with ten example program statements. Make sure you test all parts of the grammar. E.g., "A:=0;B:=succ(A)", and so on.
2. Modify the Prolog implementation of the grammar so that an arbitrary number of spaces are allowed in the usual places in a program. Do ten tests to show that the experiment is a success.

7.4 Arithmetic Expression Evaluation

In this experiment we'll consider the problem of parsing and evaluating arithmetic expressions. For example, consider the following grammar.

$E \rightarrow N–E \mid N$

$N \rightarrow D \mid DN$
$D \rightarrow$ decimal digit.

This grammar defines subtraction to be right associative. For readability in Prolog we'll use *expr*, *nat*, and *dig* to represent the nonterminals *E*, *N*, and *D* and obtain the following implementation.

```
expr --> nat, "–", expr | nat.
nat --> dig | dig, nat.
dig --> "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

or alternatively,

```
dig --> [D], {"0"=<D, D=<"9"}.
```

To evaluate an arithmetic expression we need to introduce variables to hold values and we need to introduce semantic actions to compute values. The semantic actions are placed in parentheses to separate them from the macro code. Here is the modified grammar with value variables and semantic actions added.

```
expr(A) --> nat(B, T), "–", expr(C), {A is B – C}.
expr(A) --> nat(B, T), {A is B}.
nat(X, 1) --> dig(X).
nat(X, T) --> dig(A), nat(B, S), {T is 10*S, X is A*T + B}.
dig(X) --> [D], {"0" =< D, D =< "9", X is D – "0"}.
```

NOTE: The extra variable in nat is used to keep track of the tens place of the leading (i.e., leftmost) digit of each number. This is needed because parsing is from left to right.

For an example, suppose we want to evaluate the expression

12 – 3 – 9.

We type the following goal:

```
|?- expr(A, "12–3–9", [ ]).
```

The answer will be returned as

A = 18.

Experiments to Perform

1. Do five tests to evaluate correct expressions. Do five tests of incorrect

expressions.

2. Try to discover how evaluation takes place during parsing. First, list the contents of the program to see the code that is generated by the grammar. Then trace the execution of a simple example goal such as

```
|?- expr(A, "5–2", [ ]).
```

3. Modify the Prolog implementation of the grammar so that any number of spaces are allowed in the usual places. Do three tests such as the goal,

```
|?- expr(A, "5 –  2", [ ]).
```

4. The following grammar that makes subtraction left associative.

$E \rightarrow E–N \mid N$
$N \rightarrow D \mid DN$
$D \rightarrow$ decimal digit.

For example, the expression 4 – 5 – 6 means (4 – 5) – 6. Since the grammar is left recursive, it can't be transformed directly into Prolog. So we'll remove left recursion to obtain the following grammar.

$E \rightarrow N\,R$
$R \rightarrow -\,N\,R \mid \square$
$N \rightarrow D \mid DN$
$D \rightarrow$ decimal digit.

This grammar is not as "natural" as the left recursive grammar. What semantics can we add to the productions to accomplish the evaluation of an expression? One solution is to construct a postfix representation of the expression during the parse and then evaluate it after the parse is completed. For example, the parse of 4 – 5 – 6 would construct the list [4, 5, –, 6, –]. Then an evaluation predicate could evaluate it. Here is the Prolog code to build the postfix list for the parsed expression, where eval is the evaluation predicate.

```
expr(A) --> nat(B, T), r(C), {eval([B|C], A)}.
r(A) --> "–", nat(B, T), r(C), {A = [B, –|C]}.
r(A) --> [ ], {A = [ ]}.

eval([A], A).
eval([A, B, –|T], Ans) :- X is A – B, eval([X|T], Ans).
```

a. Test the eval predicate to make sure that it evaluates postfix expressions represented as lists. For example, test the goal

```
|?- eval([4, 5, –, 6, –], Ans).
```

b. Do five tests to evaluate arithmetic expressions. For example, test the goal

```
|?- expr(A, "4–5–6", [ ]).
```

5. The following grammar expands the description given in (4) of arithmetic expressions to those that use the operations +, –, and * together with parentheses.

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid N$
$N \rightarrow D \mid D\,N$
$D \rightarrow$ decimal digit.

Remove the left recursion and add semantics actions to the resulting grammar as was done in (4). You will also have to modify the eval predicate to handle the two operations of + and *. Test the resulting grammar with several expressions that use various combinations of the operations.

8

Computability

The experiments in this chapter use Prolog to explore some of the basic computational models. We'll be looking at various kinds of finite automata, pushdown automata, and Turing machines. We'll also see the string processing models of Markov and Post.

8.1 Deterministic Finite Automata

Let's see how to build an interpreter for executing deterministic finite automata. The input for the interpreter will be a DFA in the form of a transition table. If *t* is the transition function for a DFA, then we'll represent the state transitions in the Prolog program as a collection of facts having the following form:

t(state, letter, nextstate).

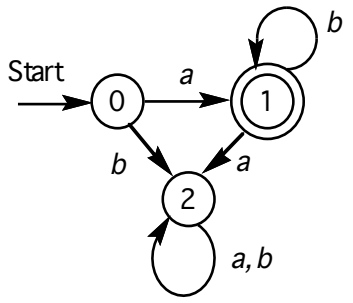
To indicate that state *i* is the start state we'll write the fact

start(*i*).

To indicate that state *s* is a final state, we'll write the fact

final(*s*).

For example, the following DFA recognizes the language of the regular expression *ab**.



This DFA in Prolog as the following facts.

```
start(0).
t(0, a, 1).
t(0, b, 2).
t(1, a, 2).
t(1, b, 1).
t(2, a, 2).
t(2, b, 2).
final(1).
```

The interpreter will process strings that are written as lists of letters. For example, to test whether the string *abb* is accepted by a DFA, we write the string as a list of letters and type the following goal:

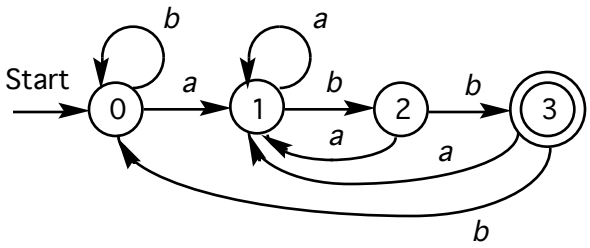
```
|?- accept([a, b, b]).
```

This action starts the execution of the following DFA interpreter, where the path predicate tries to find out whether a path exists from the start state to a final state that consumes all letters of the input string.

```
accept(S) :- start(I), path(I, S).
path(K, [ ]) :- final(K).
path(K, [H | T]) :- t(K, H, N), path(N, T).
```

Experiments to Perform

1. Calculate the transition function *t* for the following DFA and enter it as a collection of Prolog facts into an input file.



a. Use the DFA interpreter to test five strings that are accepted by the DFA and five strings that are rejected by the DFA.

b. To gain a better understanding of how the DFA interpreter works, trace the execution on a string that is accepted and on a string that is rejected by the DFA.

2. See whether you can generate some strings that are accepted by the DFA by using backtracking of goals that contain variables in place of letters. For example, try out goals like the following with backtracking.

```
|?- accept([A, B, C, D, E]).
|?- accept([H|T]).
```

Note: If an infinite loop occurs, do a trace to see what is happening.

3. Find a DFA for the regular expression

$$aa^*b + b(a + b).$$

Draw a picture of the DFA. Use the DFA interpreter to test the DFA on five strings that are accepted and five strings that are rejected. Try backtracking with goals that contain variables in place of letters.

8.2 Nondeterministic Finite Automata

Let's see how to build an interpreter for executing nondeterministic finite automata. The input for the interpreter will be an NFA in the form of a transition table. If t is the transition function for an NFA, then we'll represent the state transitions in the Prolog program as a collection of facts having the following form:

t(state, symbol, nextstate).

We'll use the empty list [ ] to denote the symbol Λ. To indicate that state i is

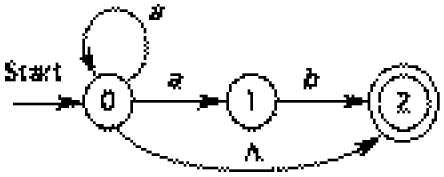
the start state we'll write the fact

start(*i*).

To indicate that state *s* is a final state, we'll write the fact

final(*s*).

For example, the following NFA recognizes the language of the regular expression $a^*ab + a^*$.



We can represent this NFA in Prolog as the following facts.

```
start(0).
t(0, a, 0).
t(0, a, 1).
t(0, [ ], 2).
t(1, b, 2).
final(2).
```

The interpreter will process strings that are written as lists of letters. For example, to test whether the string *aab* is accepted by the NFA, we write the string as a list of letters and type the following goal:

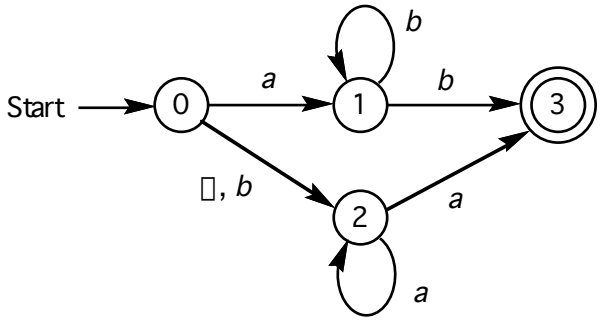
| ?- accept([*a, a, b*]).

This action starts the execution of the following NFA interpreter, where the path predicate tries to find out whether a path exists from the start state to a final state that consumes all letters of the input string.

```
accept(S) :- start(I), path(I, S).
path(K, [ ]) :- final(K).
path(K, [H | T]) :- t(K, H, N), path(N, T).
path(K, X) :- t(K, [ ], N), path(N, X).
```

Experiments to Perform

1. Calculate the transition function t for the following NFA and enter it as a collection of Prolog facts into an input file:



   a. Use the NFA interpreter to test five strings that are accepted by the NFA and five strings that are rejected by the NFA.

   b. To gain a better understanding of how the NFA interpreter works, trace the execution on a string that is accepted and on a string that is rejected by the NFA.

2. See whether you can generate some strings that are accepted by the NFA by using backtracking of goals that contain variables in place of letters. For example, try out goals like the following with backtracking.

```
|?- accept([A, B, C]).
|?- accept([a|T]).
|?- accept([b|T]).
|?- accept([H|T]).
```

   Note: If an infinite loop occurs, do a trace to see what is happening.

3. Find an NFA for the regular expression

$$ab^*c + b^*c + ac^*b.$$

   Draw a picture of the NFA. Use the NFA interpreter to test the NFA on five strings that are accepted and five strings that are rejected. Try backtracking with goals that contain variables in place of letters.

4. Since any DFA is an NFA by default, we should be able to execute any DFA on the NFA interpreter. Try it out with a DFA of your own choosing.

5. We want to include the capability of executing NFAs with instructions that include lists of next states. For example, instead of writing the two instructions t(0, a, 1) and t(0, a, 2), we write the single instruction

t(0, a, [1, 2]).

Modify the NFA interpreter so that, in addition to executing instructions with single next states as it does now, it also executes instructions that have lists of next states. So it should be able to handle instructions of the form

t(state, letter, [next1, next2, ...])
t(state, [ ], [next1, next2, ...])

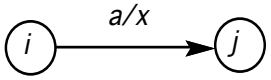
Hint: Add two more "path" clauses to execute lists of states.

Test the definition on the following NFA to accept strings for the language of the regular expression $abb^* + bb^* + baa^* + aa^*$.

```
start(0).
t(0,a,1).
t(0,b,2).
t(0,[ ], [1, 2]).
t(1,b,[1,3]).
t(2,a,[2,3]).
final(3).
```

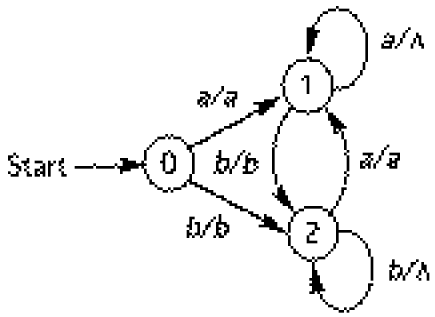
8.3 Mealy Machines

Recall that a Mealy machine is a finite automaton with output where each edge has a label of the form a/x, where a is a letter of the input alphabet and x is an output letter. For example, a typical edge from state i to state j with input a and output x looks like the following.



If the machine is in state i with input letter a, then the letter x is output and the machine moves to state j. There is a start state but no final state since we are not concerned with acceptance or rejection of an input string. A Mealy machine has one edge out of each state for each letter of the alphabet. For

example, let's construct a Mealy machine to collapse substrings of two or more identical letters over the alphabet {*a, b*} to a single letter. For example, the input *abbaaabbbaa* will yield the output *ababa*. Here is a picture of the machine.



We can represent state transitions as facts of the form

t(state, input, output, nextstate).

To indicate that state *i* is the start state we'll write the fact

start(*i*).

For example, the preceding Mealy machine can be represented by the following facts, where the output symbol Λ is represented by an empty list.

```
start(0).
t(0, a, a, 1).
t(0, b, b, 2).
t(1, a, [ ], 1).
t(1, b, b, 2).
t(2, a, a, 1).
t(2, b, [ ], 2).
```

It's quite easy to construct an interpreter for Mealy machines. We'll represent input and output strings as lists. Let the predicate *mealy*(*A*, *B*) mean that for a given Mealy machine with input list *A* the output after execution is the list *B*. For example, for the preceding machine, the goal

```
| ?- mealy([a, b, b, a, a, a, b], X).
```

will return X = [a, b, a, b]. With these assumptions the Mealy interpreter can be written as the following simple program, where the execute predicate tries

to find a path that consumes all letters of the input string and at the same time keeps track of the output string.

```
mealy(In, Out) :- start(I), execute(I, In, Out).
execute(S, [ ], [ ]).
execute(S, [H|T], Y) :- t(S, H, [ ], N), execute(N, T, Y).
execute(S, [H|T], [X|Y]) :- t(S, H, X, N), execute(N, T, Y).
```

Experiments to Perform

1. Use the Mealy machine interpreter and the example Mealy machine to perform each of the following tests.

   a. Test the Mealy machine interpreter and the Mealy machine on five input strings.

   b. To gain a better understanding of how the Mealy machine interpreter works, trace the execution on at least two strings.

2. See whether you can generate some input and/or output strings for the example Mealy machine by using backtracking of goals that contain variables as arguments. For example, try out goals like the following with backtracking to see if any patterns occur.

```
|?- mealy(X, Y).
|?- mealy(X, [a]).
|?- mealy(X, [b]).
|?- mealy(X, [c]).
```

3. Construct a Mealy machine to decode a string of binary digits of even length, where the code is defined as follows: 00 = *a*, 01 = *b*, 10 = *c*, 11 = *d*. For example, the string 010011 decodes to the string *bad*. Representing strings as lists, the goal

```
|?- mealy([0, 1, 0, 0, 1, 1], X).
```

   will return X = [b, a, d]. Use the Mealy interpreter to test the machine on five input strings. Do a trace on one of the tests.

   Note: If you wish, you may create a more sophisticated code together with a Mealy machine to decode strings.

4. Construct a Mealy machine to model a candy machine that dispenses two kinds of 15 cent candy bars and accepts only nickels and dimes. Let *n* and *d* represent nickel and dime, respectively. Let *a* and *b* represent

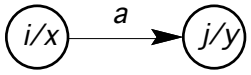
buttons to push for candy bars *alpha* and *beta*, respectively. Assume that the machine always returns correct change and if a button is pushed with less than 15 cents in the machine, then the coins are returned. For example, the goal

```
|?- mealy([d, d, a], X).
```

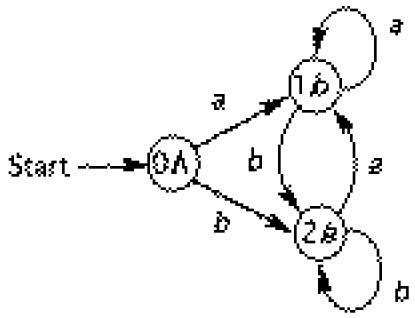
should return X = [n, alpha] or X = [alpha, n]. Use the Mealy interpreter to test your Mealy machine on five input strings. Do a trace on one of the tests.

8.4 Moore Machines

Recall that a Moore machine is a finite automaton with output that occurs at each state. For example, if the output associated with state i is x, we'll write i/x inside the state circle. A typical state transition for a Moore machine can be pictured as follows:



Each time a state is entered, an output takes place. So the first output always occurs as soon as the machine is started. There is a start state but no final state since we are not concerned with acceptance or rejection of an input string. A Moore machine has one edge out of each state for each letter of the alphabet. For example, let's construct a Moore machine to interchange a's and b's in strings over the alphabet $\{a, b\}$. For example, the input $abab$ will yield the output $baba$. Here is a picture of the machine.



We can represent a state transition as a fact of the form

t(state, output, input, nextstate).

To indicate that state i is the start state we'll write the fact

start(*i*).

For example, the example Moore machine can be represented by the following facts, where the output symbol □ is represented by an empty list.

```
start(0).
t(0, [ ], a, 1).
t(0, [ ], b, 2).
t(1, b, a, 1).
t(1, b, b, 2).
t(2, a, a, 1).
t(2, a, b, 2).
```

It's easy to construct an interpreter for Moore machines. For example, for this machine, the goal

```
|?- moore([a, b, a, b], X).
```

will return X = [b, a, b, a]. The Moore machine interpreter is a simple program, consisting of the following clauses, where the execute predicate tries to find a path that consumes all letters of the input string and at the same time keeps track of the output string.

```
moore(In, Out) :- start(I), execute(I, In, Out).
execute(S, [ ], [ ]) :- t(S, [ ], Y, Z).
execute(S, [ ], [X]) :- t(S, X, Y, Z).
execute(S, [H|T], Y) :- t(S, [ ], H, N), execute(N, T, Y).
execute(S, [H|T], [A|Y]) :- t(S, A, H, N), execute(N, T, Y).
```

Experiments to Perform

1. Use the Moore machine interpreter and the example Moore machine to perform each of the following tests.

   a. Test the Moore machine interpreter and the Moore machine on five input strings.

   b. To gain a better understanding of how the Moore machine interpreter works, trace the execution on at least two strings.

2. See whether you can generate some input and/or output strings for the example Moore machine by using backtracking of goals that contain variables as arguments. For example, try out goals like the following with backtracking to see if any patterns occur.

```
|?- moore(X, Y).
```

```
|?- moore(X, [a, b, a, b]).
|?- moore(X, [c]).
```

3. Construct a Moore machine to collapse substrings of two or more identical letters over the alphabet {*a, b*} to a single letter. For example, the input *abbaaabbbaa* will yield the output *ababa*. Use the Moore interpreter to test your Moore machine on five input strings. Do a trace on one of the tests.

8.5 Pushdown Automata

In this experiment we'll construct an interpreter for pushdown automata. Suppose we write a pushdown automaton as a set of Prolog facts of the following two forms.

```
t(state, letter, top, operation, nextstate).
start(state).
final(state).
```

In order to write a simple interpreter for PDAs, we'll need to make a few assumptions. The input string will be represented as a list. The stack is a list that is initialized with the value [*e*], which means *e* is always the starting stack symbol. We'll reserve the letters *p* and *n* for the operations pop and nop, and we'll agree to let the push instruction be represented by the symbol that is to be pushed. For example, in the instruction

$$t(0, a, e, b, 1)$$

the letter *b* means push *b*.

For example, a PDA to recognize the language $\{a^n b^n \mid n \in \mathrm{N}\}$ can be written as the following set of facts.

```
start(0).
t( 0, a, e, a, 0 ).
t( 0, a, a, a, 0 ).
t( 0, b, a, p, 1 ).
t( 0, [ ], e, n, 2 ).
t( 1, b, a, p, 1 ).
t( 1, [ ], e, n, 2 ).
final(2).
```

To check whether this PDA accepts the string *aabb*, we type the following goal:

```
| ?- accept([a, a, b, b]).
```

This action starts the execution of the PDA interpreter, which we will describe next.

The interpreter executes a computation sequence, where the "path" predicate represents an ID containing the current state, the current input string, and the current stack. If the input is empty and the current state is final, then the computation ends successfully. Otherwise, if the input is not empty, the computation continues by looking up an appropriate instruction. The predicate oper(Stack, *O*, NewStack) means "perform stack operation *O* on Stack, resulting in NewStack." The interpreter can be written as follows, where *S* is the list representing the input string:

```
accept(S) :- start(I), path(I, S, [e]).
path(K, [ ], Stack) :- final(K).
path(K, [A|B], [H|T]) :-  t(K, A, H, O, N),
                          oper([H|T], O, NewStack),
                          path(N, B, NewStack).
path(K, S, [H|T]) :-  t(K, [ ], H, O, N),
                      oper([H|T], O, NewStack),
                      path(N, S, NewStack).
oper([H|T], p, T).
oper(Stack, n, Stack).
oper(Stack, A, [A|Stack]).
```

Experiments to Perform

1. Use the PDA interpreter and the example PDA to perform each of the following tests.
   - **a.** Test five strings that are accepted and five strings that are rejected..
   - **b.** To gain a better understanding of how the PDA interpreter works, trace the execution on two acceptable strings and a string that is not accepted.
2. Find a PDA for the language of all strings over {*a, b*} that have the same number of *a*'s and *b*'s. Test your solution with the PDA interpreter.
3. Modify the PDA interpreter so that it executes PDAs that accept by empty stack rather than by final state. The following PDA accepts the language $\{a^n b^n \mid n \in \mathrm{N}\}$ by empty stack.

```
t( 0, a, e, e, 0 ).
t( 0, [ ], e, p, 1 ).
```

t(1, *b, e, p,* 1).
start(0).

Test this PDA on your modified interpreter by checking five strings that are accepted and five strings that are rejected.

4. Modify the PDA interpreter so that it executes PDA instructions that contain lists of stack operations. For example, the interpreter should be able to execute an instruction like

t(0, b, a, [p, b, a], 1).

This instruction performs three stack operations: pop, push(b), push(a).

8.6 Turing Machines

We'll describe a logic program interpreter for Turing machines. The interpreter will execute any Turing machine with a single two-way infinite tape. We'll make the following assumptions about any Turing machine that is to be executed by the interpreter.

The read/write head is at the left end of any nonempty input string.
The letters *l*, *s*, and r represent the moves of the read/write head.
The symbol # denotes a blank tape cell.
Instructions are represented as facts of the following form.

t(state, letterToRead, letterToWrite, move, nextState).
start(state).

For example, a Turing machine to add 1 to a binary number can be written as the following set of facts:

start(0).
t(0, 0, 0, *r*, 0).
t(0, 1, 1, *r*, 0).
t(0, #, #, *l*, 1).
t(1, 0, 1, *s*, halt).
t(1, 1, 0, *l*, 1).
t(1, #, 1, *s*, halt).

To find the result of adding 1 to the binary number 1011, we type the goal

|?- compute([1, 0, 1, 1], Out).

This action starts the execution of the Turing machine interpreter, which in turn executes a computation sequence of the given Turing machine. When a "halt" instruction is executed the variable Out will be returned as a list that represents the tape.

The interpreter starts out by placing the input list onto the tape. The tape is represented by the following three items.

"Cell" is a variable that holds the tape symbol in the current cell pointed at by the read/write head.

"Left" is a list that holds the information on the tape to the left of the current cell. The head of the list holds the symbol immediately to the left of the current cell.

"Right" is a list that holds the information on the tape to the right of the current cell. The head of the list holds the symbol immediately to the right of the current cell.

For example, the input list [1, 0, 1, 1] is represented on the tape as

left = [ ], Cell = 1, Right = [0, 1, 1].

The "find" predicate tries to find and execute an instruction. Its first argument is the state. The next three arguments are the representation of the tape, and the last argument holds the variable for the output tape. The "move" predicate makes a move and returns a new representation of the tape in variables *A*, *B*, and *C*. The "continue" predicate checks for the halt state. If it's found, the output tape is constructed and placed in the last variable. Otherwise the execution continues by calling the "find" predicate to execute another instruction. The clauses for the interpreter are listed as follows:

```
compute([ ], OutTape) :-
        start(I),
        find(I, [ ], #, [ ], OutTape).

compute([Head|Tail], OutTape) :-
        start(I),
        find(I, [ ], Head, Tail, OutTape).

find(State, Left, Cell, Right, OutTape) :-
        t(State, Cell, Write, Move, Next),
        move(Move, Left, Write, Right, A, B, C),
        continue(Next, A, B, C, OutTape).
```

```
continue(halt, Left, Cell, Right, OutTape) :-
        reverse(Left, R),
        append(R, [Cell|Right], OutTape).

continue(State, Left, Cell, Right, OutTape) :-
        find(State, Left, Cell, Right, OutTape).

move(l, [ ], Cell, Right, [ ], #, [Cell|Right]).
move(l, [Head|Tail], Cell, Right, Tail, Head, [Cell|Right]).
move(s, Left, Cell, Right, Left, Cell, Right).
move(r, Left, Cell, [ ], [Cell|Left], #, [ ]).
move(r, Left, Cell, [Head|Tail], [Cell|Left], Head, Tail).
```

When the halt state is reached, the tape is reconstructed as a single list with entries in the proper order. To do this we need to reverse the "Left" part of the tape before concatenating it to the list [Cell|Right]. The predicate "reverse(*X*, *Y*)" sets *Y* to the reverse of list *X*. The output tape consists of all cells that were used during the computation. The predicate "append(*X*, *Y*, *Z*)" sets *Z* to the concatenation of the two lists *X* and *Y*. These predicates can be accessed by including the following statement with the source.

```
:- use_module(library(lists)).
```

For the example Turing machine, to add 1 to the binary number 1011, we write the goal

```
|?- compute([1, 0, 1, 1], Out).
```

This causes the Turing machine interpreter to execute the instructions of the Turing machine and, upon halting, to return the list

```
Out = [1, 1, 0, 0, #],
```

which represents the binary number 1100. Notice in this example that the symbol # is included in the output tape. This is because the computation used that extra cell when it scanned to the right looking for a blank.

Experiments to Perform

1. Use the Turing machine interpreter and the sample Turing machine to perform each of the following tests.

   a. Test five input strings that represent binary numbers as lists like

   ```
   [0],    [1, 1, 1, 1],    [0, 1, 1, 0, 0, 1, 1],    and so on.
   ```

b. What happens when the sample Turing machine is started on a blank tape [ ]? Is there any significance to this?

c. To gain a better understanding of how the Turing machine interpreter works, trace the execution on at least two strings.

2. Write a Turing machine to accept strings of the form $\{a^n b^n c^n \mid n \in N\}$. Test your solution with the Turing machine interpreter.

3. Notice that the Turing machine interpreter stops with a "no" answer whenever it tries to execute an instruction that is not in the program. For example, suppose that we have the example Turing machine in the program and we type the following goal.

```
|?- compute([a, 1, 0], X).
```

Since there is no instruction with "a" in the "letterTo Read" position, the find predicate fails and thus the goal fails. Suppose that instead of a "no" answer we want the tape returned along with a message like

"Error: no instruction to execute of the form t(State, Cell, _, _, _)."

Modify the interpreter to accomplish this action and test it with a sample Turing machine. *Hint:* Add a second clause to the find predicate.

4. Write and test a multi-tape Turing machine interpreter that executes any Turing machine having one or more tapes. Use a goal of the following type to execute such a Turing machine.

```
|?- compute(Input, Output).
```

where Input is a list of input lists that will initialize each of the tapes, and Output is a variable that returns a list of lists representing the content of the tapes. For example, suppose that a typical instruction for a 2-tape Turing machine is written as the Prolog fact

```
t(0, [a, b], [b, a], [r, s], 1).
```

Suppose further that we wanted to execute this Turing machine with initial tape configurations such as [a, b, a, b] and [a, a, a, a, a]. Then we would write the following goal.

```
|?- compute([[a, b, a, b], [a, a, a, a, a]], Output).
```

8.7 Markov Algorithms

Markov algorithms, which are string processing functions, form a computational model that has the same power as Turing machines. We'll describe a logic program interpreter for Markov algorithms.

A *Markov algorithm* over an alphabet A is a finite ordered sequence of productions $x \rightarrow y$, where $x, y \in A^*$. Some productions may be labeled with the word "halt," although this is not a requirement. A Markov algorithm transforms an input string into an output string. In other words, a Markov algorithm computes a function from A^* to A^*. Here's how the execution proceeds:

> Given an input string w, the productions are scanned, starting at the beginning of the ordered sequence. If there is a production $x \rightarrow y$ such that x occurs as a substring of w, then the leftmost occurrence of x in w is replaced by y to obtain a transformed string. If $x \rightarrow y$ is a halt production, then the process halts with the transformed string as output. Otherwise, the process starts all over again with the transformed string, where again the scan starts at the beginning of the ordered sequence of productions. If a scan of the instructions occurs without any new replacements of the current string, then the process halts with the current string as output.

If a production has the form $\Lambda \rightarrow y$, then it transforms any string w into the string yw. For example, suppose we wish to transform any string of the form a^i into the string a^{i+1}. The following single production Markov algorithm will do the job:

$$\Lambda \rightarrow a \text{ (halt)}.$$

This production causes the letter a to be appended to the left of any input string, after which the process halts.

Here is a simple example of a Markov algorithm to interchange a's and b's in a string over {a, b}. For example, the string *abba* will be transformed into the string *baab*. The algorithm consists of the following productions.

```
#a → b#
#b → a#
# → Λ (halt)
Λ → #
```

The interpreter is quite simple to construct once we decide on the representation for the productions. Since strings are represented internally

as lists, we'll assume that all strings are lists of characters represented in ASCII. Thus we need to put double quotes around each string. The empty string will be represented by adjacent double quotes, "". Since an instruction may have halt attached, we'll assume that each production is a Prolog fact of the following form, where the third argument indicates whether the production has halt attached.

m(Left, Right, no/halt).

For example, the productions of the example Markov algorithm are represented by the following Prolog facts.

```
m("#a", "b#", no).
m("#b", "a#", no).
m("#", "", halt).
m("", "#", no).
```

With the preceding assumptions we can write a simple interpreter to execute any Markov algorithm. The predicate "markov" will initiate the computation. For example, the goal

|?- markov("abba", Out).

will return the answer Out = baab. Here are the definitions, including an access call to the lists library.

```
:-use_module(library(lists)).

markov(In, Out) :- mark(In, ListOut), name(Out, ListOut).

mark(In, Out) :-
      m(L, R, Status),
      replace(L, R, In, X),
      check(Status, X, Out).
mark(In, In).

check(halt, X, X).
check(no, X, Out) :- mark(X, Out).
```

The "replace" predicate checks to see whether the sublist L occurs in the list In and if so, it replaces the leftmost occurrence of L by R. For example, the goal replace([a, b], [x, y, z], [b, a, b, a, b], X) returns X = [b, x, y, z, a, b]. The "prefix" predicate tests for a prefix and if found, it returns the rest of the list.

For example, the goal prefix([a, b], [a, b, e, f, g], R) returns R=[e, f, g]. Here are the definitions.

```
replace(L, R, In, Out):- prefix(L, In, T), append(R, T, Out).
replace(L, R, [X|Xs], [X|Out]) :- replace(L, R, Xs, Out).

prefix([ ], X, X).
prefix([H|T], [H|S], Rest):- prefix(T, S, Rest).
```

Experiments to Perform

1. Use the Markov interpreter and the example Markov algorithm to perform each of the following tests.
   - **a.** Test five input strings.
   - **b.** To gain a better understanding of how the Markov interpreter works, trace the execution on at least two strings.
2. Construct and test a Markov algorithm to collapse strings over {a, b} into strings with no repeated substrings of a's or b's. For example, the string *aaabbbabbb* will be transformed into the string *abab*.
3. Construct and test a Markov algorithm to reverse a string over the alphabet {a, b}. For example, if the input string is *abbaa*, then the output string is *aabba*.

8.8 Post Algorithms

Post algorithms, which are string processing functions, form a computational model that has the same power as Turing machines. We'll describe a logic program interpreter for Post algorithms.

A *Post algorithm* over an alphabet A is a finite set of productions that are used to transform strings. So a Post algorithm computes a function from A^* to A^*. The productions have the form $s \rightarrow t$, where s and t are strings made up of symbols from A and possibly some variables. A variable X occurs in s if and only if it occurs in t. There is no particular ordering of the productions in a Post algorithm, unlike the ordering of productions in Markov algorithms. Some productions may be labeled with the word "halt," although this is not required.

The computation of a Post algorithm proceeds by string pattern matching. If the input string matches the left side of some production, then we construct a new string to match the right side of the same production. If the pro-

duction is a halt production, then the computation halts, and the new string is output. Otherwise, the process continues by trying to match the new string with the left side of some production. If no matches can be found, then the process halts, and the output is the current string.

A Post algorithm can be either deterministic or nondeterministic. Nondeterminism occurs if some computation has a string that matches the left side of more than one production or matches the left side of a production in more than one way. Any nondeterministic Post algorithm can be rewritten as a deterministic Post algorithm. So no additional power is obtained by nondeterminism.

The interpreter is quite simple to construct once we decide on the represention for the productions. Since strings are represented internally as lists, we'll assume that all strings are lists of characters represented in ASCII. Thus we need to put double quotes around each string. The empty string will be represented by adjacent double quotes, "". Since an instruction may have halt attached, we'll assume that each production is a Prolog fact of the following form, where the third argument indicates whether the production has halt attached.

p(Left, Right, no/halt).

For example, a deterministic Post algorithm to remove the rightmost occurrence of the letter b from any string over {a, b} can be written as follows.

Xb → X (halt)
Xa → X#a@
Xa#Y → X#aY
Xb#Y@ → XY (halt)
#X@ → X (halt)

These instructions are represented as the following Prolog facts.

```
p([X, "b"], [X], halt).
p([X, "a"], [X, "#a@"], no).
p([X, "a#", Y], [X, "#a", Y], no).
p([X, "b#", Y, "@"], [X, Y], halt).
p(["#", X, "@"], [X], halt).
```

With the preceding assumptions we can write an interpreter to execute any Post algorithm. The predicate "post" will initiate the computation. For example, the goal

|?- post("abbba", Out).

will return the answer Out = abba. Here are the definitions, including an access call to the lists library.

```
:-use_module(library(lists)).

post(In, Out):-
        po(In, ListOut),
        flatten(ListOut, List),
        name(Out, List).

po(In, Out) :-
        p(L, R, Status),
        match(L, In),
        check(Status, R, Out).
po(In, In).

check(halt, R, R).
check(no, R, Out) :-
        flatten(R, F),
        po(F, Out).

match([ ], [ ]).
match([X], X).
match([H|T], Input):-
        var(H),
        getVariables(T, Vars, String, Rest),    % get variables
        partition(Input, String, Left, Right),
        assign([H|Vars], Left),
        match(Rest,Right).
match([H|T], Input):-
        is_list(H),
        prefix(H, Input, Rest),
        match(T, Rest).

getVariables([H|T],[H|K], String, Rest):-
        var(H),
        getVariables(T, K, String, Rest).
getVariables([H|T], [ ], H, T) :- is_list(H).
getVariables([ ], [ ], [ ], [ ]).

partition(In, Pattern, [ ], Right) :- prefix(Pattern, In, Right).
partition([H|T], Pattern, [H|L], Right) :- partition(T, Pattern, L, Right).

assign([X|T], R) :- var(X), X=R, assign(T,[ ]).
assign([ ], _).
```

The "prefix" predicate tests for a prefix and, if found, returns the rest of the list. For example, the goal prefix([a, b], [a, b, e, f, g], R) returns R=[e, f, g]. The "flatten" predicate flattens a list of lists into a list of elements. Here are the definitions.

```
prefix([ ], X, X).
prefix([H|T], [H|S], Rest):- prefix(T, S, Rest).

flatten([H|T], Out):-
      is_list(H),
      flatten(H, F),
      flatten(T, S),
      append(F, S, Out).
flatten([H|T],[H|S]):- flatten(T, S).
flatten([ ], [ ]).
```

Experiments to Perform

1. Implement the interpreter for Post algorithms and test it in the following ways.

   a. Test the sample Post algorithm to remove the rightmost occurrence of the letter b from any string over {a, b}. Be sure to test the algorithm on a variety of strings such as: the empty string; the single letter a; the single letter b; all a's; all b's; b's followed by a's; a's followed by b's; and intermixed a's and b's.

   b. To gain a better understanding of how the Post interpreter works, trace the execution on at least two strings.

   c. The Post algorithm consisting of the single production XaY □ XY is a nondeterministic algorithm to remove all occurrences of the letter a from any string over any alphabet. Use the interpreter to test this Post algorithm on at least five input strings.

2. Construct and test a Post algorithm to collapse strings over {a, b} into strings with no repeated substrings of a's or b's. For example, the string *aaabbbabbb* will be transformed into the string *abab*.

3. Construct and test a Post algorithm to reverse a string over the alphabet {a, b}. For example, if the input string is *baabba*, then the output string is *abbaab*.

9

Problems and Projects

The experiments in this chapter use Prolog to explore a variety of problems and projects that have lengthier programming needs.

9.1 Lambda Closure

When constructing a DFA from an NFA, we need to calculate the lambda closure of various sets of states. In this experiment, the predicate "closure" will do the job. For example, suppose we are given an NFA with states 0, 1, 2, 3, 4. To calculate the closure of the set {0, 1} we can type the following goal:

```
|?- closure([0, 1], [0, 1, 2, 3, 4], C).
```

In the following program for closure, as in the NFA experiment, an NFA table is represented as a collection of facts of the form

t(state, symbol, nextstate).

The lamba transitions for the NFA are represented in the form

t(state, [ ], nextstate).

We can write a program for the closure predicate as follows.

```
%  The closure predicate.
closure(S, [ ], [ ]).
closure(S, [H|T], Ans)
                :- inClosure(H, S), closure(S, T, B), append([H], B, Ans).
closure(S, [H|T], Ans) :- closure(S, T, Ans).
```

```
% Test to see if X is in the closure of S.
inClosure(X, S) :- member(X, S).
inClosure(X, [H|T]) :- trans(H, X).
inClosure(X, [H|T]) :- inClosure(X, T).

% Test to see if (I, J) is in the transitive closure of t via lambda.
trans(I, J) :- t(I, [ ], J).
trans(I, J) :- t(I, [ ], K), trans(K, J).
```

Experiments to Perform

1. Suppose we have the following NFA table:

		a	*b*	□
Start	0	∅	{1, 2}	{1}
	1	{2}	∅	∅
Final	2	∅	{2}	{1}

For this NFA, compute the lambda closures of the following sets by hand, and then use the closure predicate to verify your answers.

a. {0}. **b.** {1}. **c.** {2}. **d.** {0, 1}. **e.** {1, 2}.

2. It can get tedious to type goals that always include the states of an NFA. For example, if there are 12 states in some NFA {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11} and we want the closure of the set {0, 1}, then we have to type the goal

```
|?- closure([0, 1], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], C).
```

Modify the program so that you only need to type the input set whose closure is to be found. Do this by putting the states in the data base along with the transition table. For example, we can create a states predicate and enter a fact like the following.

```
states([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]).
```

Then the modified program can "pick up" the states that it needs to use

in the computation from the states predicate. So the closure goal can be modified to something like

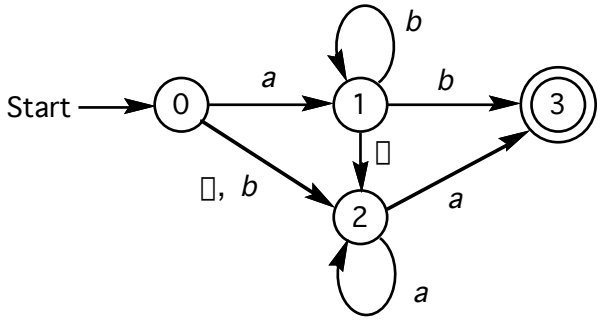
```
|?- newClosure([0, 1], C).
```

3. Use "Thompson's Construction" to find an NFA for the regular expression $(aa + b)^*(ab)^*$. Then use the closure predicate to compute the lambda closure of each entry of the transition table.

4. In Exercise (2) the closure predicate was modified by putting the states of an NFA in the program. Modify the closure predicate by computing the list of states without having them in the program. In other words, try to find an algorithm that constructs the list of states from the transition table of the NFA.

9.2 Transforming an NFA into a DFA

In this experiment, we'll develop and test an algorithm to transform any NFA to an equivalent DFA such that both machines recognize the same regular language. We'll assume that an NFA is represented in the program in the usual way. For example, suppose we have the following NFA.



This NFA will be represented by the following set of facts.

```
start(0).
t(0, a, 1).
t(0, b, 2).
t(0, [ ], 2)
t(1, b, 1).
t(1, b, 3).
t(1, [ ], 2).
t(2, a, 2).
t(2, a, 3).
final (3).
```

Given an NFA in the program, we'll translate the NFA into an equivalent DFA by typing the goal

```
|?- dfa(S, T, F).
```

The program will return the start state S, the table T of DFA instructions as a list, and the list F of final states.

We'll need the list processing library, which can be accessed by including the following statement with the source program.

```
:- use_module(library(lists)).
```

Here is the defintion of the dfa predicate.

```
dfa(S, T, F) :-
        getAlphabet(Alphabet),          % get DFA alphabet
        getStartState(S),               % get DFA start state
        getTable([S], Alphabet, T),     % get DFA table
        getFinals(T, F).                % get DFA final states
```

The getAlphabet and getStartState predicates can be written as follows.

```
getAlphabet(Alphabet):-
        setof(X, Y^Z^t(Y, X, Z), A),
        delete(A, [ ], Alphabet).

getStartState(S):- start(I), closure([I], S).
```

The getTable predicate constructs the DFA table by rows, beginning with the row indexed by the start state of the DFA. As long as new entries appear in the table, continue to construct rows indexed by them. The initial call to compute the table T is the goal

```
|?- getTable([DFAStartState], Alphabet, T).
```

For example, if the DFA start state is [0, 2] and the alphabet is [a, b], then the initial goal would be

```
|?- getTable([[0,2]], [a, b], T).
```

Here is the definition of the getTable predicate.

```
getTable([ ], _, [ ]).
getTable([H|T], A, Table) :-
        row(H, A, Row, Y),              % get row H of table
        delete(Y, H, Z),                % compute remaining
        append(T, Z, W),                % row indices
        getTable(W, A, Rest),           % get rest of the table
        union(Row, Rest, Table).
```

The row predicate constructs a row of DFA table instructions. For example, to find the row indexed by the state [0, 2] with column alphabet [a, b], we would write the goal

```
|?- row([0, 2], [a, b], R, N).
```

The goal returns the row R of DFA instructions and the list N of next states, some of which might be new row indices. For example, the goal might return R = [t([0, 2], a, [1, 2, 3]), t([0, 2], b, [2])], and N = [[1, 2, 3], [2]]. Here is the definition of the row predicate.

```
row(_, [ ], [ ], [ ]).
row(S, [H|T], [t(S, H, X)|Y], [X|Z]) :- entry(S, H, X), row(S, T, Y, Z).
```

The entry predicate constructs next state for an entry of the DFA table. For example, to find the entry in row [0, 2] and column "a" of the DFA table, we would write the goal

```
|?- entry([0, 2], a, X).
```

If the goal returns X = [1, 2, 3], then this defines the DFA instruction instruction t([0, 2], a, [1, 2, 3]). The algorithm computes X as the □-closure of the union of the entries in the NFA table that occur in column “a” at rows 0 and 2. Here is the definition of the entry predicate.

```
entry([ ], _, [ ]).
entry([H|T], A, U) :-
        setof(X, t(H, A, X), R),
        entry(T, A, S),
        union(R, S, V),
        closure(V, U).
        entry([H|T], A, U) :-           % no t(H, A, _) entry
        entry(T, A, S),
        closure(S, U).
```

Here is a definition for the getFinals predicate to compute the set F of final states from the table T of DFA instructions.

```
getFinals(T, F) :-
        setof(S, A^B^member(t(S, A, B), T), SS),
        setof(S, (member(S, SS), X^(member(X, S), final(X))), F).
```

Here is a definition for the closure predicate to compute the lambda closure C of a list of states S in an NFA.

```
closure(S, C) :- getStates(States), getClosure(S, States, C).

getClosure(S, [ ], [ ]).
getClosure(S, [H | T], Ans) :-
        inClosure(H, S),
        getClosure(S, T, B),
        append([H], B, Ans).
getClosure(S, [H | T], Ans) :- getClosure(S, T, Ans).
```

Here is a definition for the getStates predicate to compute the list of DFA states from the NFA instructions.

```
getStates(States) :-
        setof(S, Y^Z^t(S, Y, Z), C), % get current states
        setof(S, X^Y^t(X, Y, S), N),        % get next states
        union(C, N, States).
```

Here is a definition for the inClosure predicate to test to see if X is in the λ-closure of S.

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

Here is a definition for the trans predicate to test to see whether there is a path from state I to state J by λ-edges.

```
trans(I, J) :- t(I, [ ], J).
trans(I, J) :- t(I, [ ], K), trans(K, J).
```

Experiments to Perform

1. Implement the translator and test it with the example NFA. You will

have to find a definiton for the union predicate. Notice that the output is not ready to be executed by the DFA interpreter. For example, if S is the start state, then we need to construct a Prolog fact of the following form.

start(S).

Similarly, for each final state S in the list of final states, we need to construct a Prolog fact of the following form.

final(S).

For each instruction in the table of the form t(S, A, N) we need to construct a Prolog fact of the following form

t(S, A, N).

a. Write a predicate "outPut" to place the DFA into a file as Prolog facts. Then test it with the following goal, where filename is the name of the file that will hold the DFA instructions.

|?- dfa(S, T, F), outPut(S, T, F, filename).

b. Test the translator and the outPut predicate from part (a) on the following NFA

		a	*b*	□
Start	0	∅	{1, 2}	{1}
	1	{2}	∅	∅
Final	2	∅	{2}	{1}

c. Test the translator and the outPut predicate from part (a) on a DFA of your own choosing. Is there any change between the given DFA and the output DFA?

2. Do the following tests to observe whether the translator constructs equivalent DFAs.

a. Use the NFA interpreter to test the sample NFA on five strings that it accepts and five strings that it rejects.

b. Then use the DFA interpreter to test the DFA obtained from the sample NFA on the same ten strings.

3. Repeat the two tests asked for in (2) on the following machines.

a. Use the given NFA and translated DFA from (1b).

b. Use the given DFA and translated DFA from (1c).

4. Suppose that we don't like the form of the output of the translator where the states are lists. Instead, we want to modify the output so that each state is a natural number. For example, if the states output by the translator are [0, 1, 2], [0, 2], [1, 2], and [ ], then we might associate these states with the numbers 0, 1, 2, and 3, respectively. Thus an instruction like t([0, 1, 2], a, [1, 2]) would be modified to t(0, a, 2). A predicate, modifyDFA, to do the job can be written as follows.

```
modifyDFA(Start, Table, Finals, NewS, NewT, NewFs) :-
    setof(X, Y^Z^member(t(X, Y, Z), Table), S),    % get old states
    associate(0, S, AssocPairs),                   % associate with 0, 1,...
    member([Start, NewS], AssocPairs),             % get NewS
    makeFinals(Finals, AssocPairs, NewFs),         % get NewFs
    replaceStates(Table, AssocPairs, NewT).        % get NewT
```

The predicate modifyDFA transforms the start state, table, and final states of the DFA into new forms where states are numbers. Write the Prolog definitions for the three predicates, associate, makeFinals, and replaceStates. A short description of each predicate follows.

The associate predicate creates a list of association pairs. For example, the list of pairs might look like

```
[ [ [0, 1, 2], 0], [ [0, 2], 1], [ [1, 2], 2], [ [ ], 3] ].
```

The makeFinals predicate constructs the list of final state numbers.

The replaceStates predicate constructs the list of modified instructions like, [t(0, a, 2), t(0, b, 3), ...].

After completing the definitions for these predicates, test modifyDFA on the sample NFA with the following goal.

```
|?- dfa(S, T, F), modifyDFA(S, T, F, NewS, NewT, NewF).
```

9.3 Minimum-State DFA

This experiment implements and tests a Prolog program to transform a DFA into an equivalent minimum-state DFA. As a concrete example, we'll consider the following DFA from Example 4, page 623 of DS,L,&C.

```
start(0).
```

```
t(0,a,1).
t(0,b,2).
t(1,a,4).
t(1,b,1).
t(2,a,4).
t(2,b,3).
t(3,a,4).
t(3,b,3).
t(4,a,4).
t(4,b,5).
t(5,a,5).
t(5,b,5).
final(4).
final(5).
```

The algorithm must construct a partition of the set of DFA states into equivalence classes, where each class contains a subset of DFA states that are equivalent. We start the process by forming the set

$$E_0$$

of distinct pairs of the form $\{s, t\}$, where s and t are either both final or both nonfinal. E_0 contains the possible equivalent pairs.

Next we construct a new collection E_1 from E_0 by throwing away any pair $\{s, t\}$ if there is some letter a such that $\{T(s, a), T(t, a)\}$ is a distinct pair that does not occur in E_0. This means that the pair $\{T(s, a), T(t, a)\}$ contains two states of different types. So we must throw $\{s, t\}$ away.

The process continues by constructing a new collection E_2 from E_1 by throwing away $\{s, t\}$ if there is some letter a such that $\{T(s, a), T(t, a)\}$ is a distinct pair that does not occur in E_1. This means that there is a string of length 2 such that the DFA, if started from either s or t, consumes the string and enters two different types of states. So we must throw $\{s, t\}$ out of E_1.

We continue the process by constructing a descending sequence

$$E_0 \quad E_1 \quad E_2 \quad \cdots \quad E_n \quad \cdots .$$

Each set E_n in the sequence has been constructed to have the property that for each pair $\{s, t\}$ in E_n and for any string of length less than or equal to n, the DFA, if started from either s or t, will consume the string and enter the same type of states—either both reject or both accept. Since E_0 is a finite set, the sequence of sets must eventually stop with some set E_k such that

$$E_{k+1} = E_k.$$

This means E_k is the desired set of equivalent pairs of states, because for any pair {*s*, *t*} in E_k and any length string, the DFA, if started from either *s* or *t*, will consume the string and enter the same type of states—either both reject or both accept.

Now we'll write the program. The "minDFA" predicate will execute the program with a goal such as the following where "outfile" represents the name of a file to put the minimum-state DFA as a set of Prolog facts.

```
|?- minDFA(S, T, F, outfile).
```

The goal returns a minimum state DFA where

S = start state,
T = table of instructions,
F = set of final states.

The instructions for the minimum-state DFA are placed in outfile for subsequent execution on the DFA Interpreter. Here is the Prolog program.

```
:- use_module(library(lists)).

minDFA(S, T, F, File):-
    getStates(States, Finals, NonFinals),      % find the finals and nonfinals
    getEzero(NonFinals, Finals, Ezero),        % compute E0
    getEquivStates(Ezero, Elast),              % compute Elast
    getEquivClasses(Elast, States, Classes),   % construct the classes
    getStartState(Classes, S),                 % get the min-state start state
    makeTable(Classes, T),                     % get the min-state table
    getFinalStates(Classes, F),                % get the min-state final states
    outPut(S, T, F, File).
```

The predicate definitions use several set operations (minus, equal, subset), all of which have been defined earlier.

```
getStates(States,Finals, NonFinals) :-
    setof(X, Y^Z^t(X,Y,Z), States),
    setof(X, final(X), Finals),
    minus(States, Finals, NonFinals).

getEzero(NonFs, Fs, Ezero):-
    getFinalPairs(Fs, FinalPairs),
```

```
        getNonFinalPairs(NonFs, NonFinalPairs),
        append(FinalPairs, NonFinalPairs, Ezero).

getFinalPairs(Fs, FinalPairs) :-
        setof([I,J],(member(I, Fs), member(J, Fs)),FinalPairs).

getNonFinalPairs(NonFs, NonFinalPairs) :-
        setof([I,J],(member(I, NonFs), member(J, NonFs)),NonFinalPairs).

getEquivStates(Ei, Elast) :-
        getNextESet(Ei, En),
        checkEsets(Ei, En, Elast).

checkEsets(_, [ ], [ ]).
checkEsets(Ei, En, En) :-
        equal(Ei, En).
checkEsets(Ei, En, Elast) :-
        getEquivStates(En, Elast).

getNextESet(Ei, En) :-
        getAlphabet(Alpha),
        setof([I, J], (member([I, J], Ei), check(I, J, Alpha, Ei)), En).
getNextEset(Ei,[ ]).

getAlphabet(Alpha) :- setof(Y, X^Z^t(X, Y, Z), Alpha).

% Check if [t(I, a), t(J, a)] □ Ei for all a in alphabet.
% The goal is check(I, J, alphabet, Ei).

check(_, _, [ ], _).
check(I, J, [H|T], Ei) :-
        t(I, H, S1), t(J, H, S2), !, member([S1, S2], Ei),
        check(I, J, T, Ei).

%  Compute the set of equivalence classes from the relation Elast and the
%  set of States. The goal is getEquivClasses(Elast, States, Classes).

getEquivClasses(Elast, States, Classes):-
        getClasses(Elast, States, Cs),
        getDistinctClasses(Cs, Classes).

getClasses(_, [ ], [ ]).
getClasses(Elast, [H|T], [HClass|Classes]) :-
```

```
        setof(X, member([H, X], Elast), HClass),
        getClasses(Elast, T, Classes).

getDistinctClasses([ ], [ ]).
getDistinctClasses([H|T], [H|Rest]):-
        remove(H, T, NewT),
        getDistinctClasses(NewT, Rest).

remove(H, [ ], [ ]).
remove(H, [X|T], New):-
        subset(H, X),
        remove(H, T, New).
remove(H, [X|T], [X|New]):- remove(H, T, New).

getStartState(Classes, S):-
        start(I),
        getStart(I, Classes, S).
getStart(I, [S|_], S):- member(I, S).
getStart(I, [_|T], S):- getStart(I, T, S).

makeTable([ ], [ ]).
makeTable([C|Cs], Table):-
        setof(t(C, A, K),
           I^J^(member(I, C), t(I, A, J),member(K, [C|Cs]), member(J, K)), Row),
        makeTable(Cs, Rows),
        append(Row, Rows, Table).

getFinalStates(Classes, F):-
        setof(C, I^(final(I), member(C, Classes), member(I, C)), F).

% Output DFA to a file as Prolog facts.

outPut(S, T, F, File):- tell(File),
        write('start('), write(S), write(').'), nl,
        outDFA(T),
        outFinals(F),
        told.

outDFA([H|T]):-write(H), write('.'), nl, outDFA(T).
outDFA([ ]).

outFinals([H|T]):- write('final('), write(H), write(').'), nl, outFinals(T).
outFinals([ ]).
```

Experiments to Perform

1. Implement the minimum-state transformer. Be sure to include definitions for the set operations minus, equal, and subset. Test the transformer on the example DFA. Then perform the following two tests.
   a. Use the DFA interpreter to test the sample DFA on five strings that it accepts and five strings that it rejects.
   b. Then use the DFA interpreter to test the minimum-state DFA obtained from the sample DFA on the same ten strings.
2. Take the minimum-state DFA obtained from the example DFA and use it as input for the the minimum-state transformer. Is the output the same?

9.4 Defining Operations

Prolog has a predicate that allows one to define atoms as unary and binary operators. For example, suppose that we want to use the atoms "or" and "and" to denote two binary operations. We can do this by typing the goals

```
|?- op(500, xfy, or).
|?- op(400, xfy, and).
```

We can test the operations with a few goals as follows.

```
|?- a or b or c = A or B.
|?- a and b and c = A and B.
|?- a or b and c = A and B.
|?- a or b and c = A or B.
```

Experiments to Perform

1. Try out the tests and then make some conjectures about the precedence and associativity properties of the two defined operators.
2. Redefine the two operators by using "yfx" rather than "xfy" in both cases. Perform the same tests. What conjectures can you make about the redefined operators?
3. Now redefine the operators using "xfy" again, but interchange the 400 and 500 in the first arguments. Perform the same tests. What conjectures can you make about the redefined operators?
4. Redefine the operators by using "yfx" in both cases. Perform the same

tests. What conjectures can you make about the redefined operators?

5. Now redefine both operators using "xfx" in both cases. Perform the same tests. What conjectures can you make about the redefined operators?

6. We can define prefix unary operators using "fx" or "fy." Similarly, we can define postfix unary operators using "xf" or "yf." Try out various definitions for unary operators. To start things off you might try the definition

```
|?- op(100, fx, ~).
```

Then test the definition with something like

```
|?- ~ ~ p = ~ A.
```

Try out all combinations and make some conjectures about the various different definitions of your operators.

7. If we have a binary predicate, then we can make it into an infix operator. For example, suppose that we have written a definition for the "subset" predicate to test whether a set is a subset of another set, where sets are represented as lists. Then a typical goal might look like

```
|?- subset([a, b], [c, b, d, a]).
```

It we want to also use subset as an infix operator, we can make the following definition.

```
|?- op(400, xfy, subset).
```

Then we can use either form for the same goal. For example, the following goals should return the same result.

```
|?- subset([a, b], [c, b, d, a]).
|?- [a, b] subset [c, b, d, a].
```

Define a binary predicate of your choice in the usual way. Then redefine it as an infix operator and test the two different ways to represent the same goal.

9.5 Tautology Tester

Recall that a propositional wff is a tautology if it is true for all assignments of truth values to its letters. The goal of this experiment is to write a logic program to test whether a wff is a tautology. For this experiment we'll assume the wffs use lowercase letters, together with the four operators in the set {¬, □, , □ }. We'll use the method of Quine together with the following

fact.

> If A is a wff containing a letter p, then A is a tautology iff $A(p/\text{true})$ and $A(p/\text{false})$ are both tautologies.

To implement this idea, let "replace" be the predicate defined as follows:

replace(p, true, A, B) means $B = A(p$ / true).

Then we can say that A is a tautology iff B and C are tautologies where B and C are calculated by

replace(p, true, A, B) and replace(p, false, A, C).

At this point, B and C might contain another letter, say q. In this case we would call

replace(q, true, $B \square C$, D) and replace(q, false, $B \square C$, E).

Then we can say that A is a tautology iff D and E are tautologies. We continue in this manner until there are no propositional letters left. If D and E consist only of expressions involving true and false, then we find the value of $D \square E$ by calling the predicate

val($D \square E$, Answer)

which returns the truth value of $D \square E$ and puts it in Answer.

This is a very sketchy introduction to the problem. Now we'll give a partially completed Prolog program to solve the problem. Note that the symbols ->, #, &, and ~ on the keyboard will stand for the propositional operators $\square$, , $\square$, and ¬, where the operators are listed in order of precedence from lowest to highest.

```
% This is a partial Prolog program to process a file containing propositional
% wffs. As each wff is processed, a statement will be written saying whether
% it is a tautology. Whenever a definition needs to be completed, a comment
% marks the place where the clauses should go. Your task is to complete the
% definitions and then test the program.

    main(InFile) :- see(InFile), loop.

    loop :- read(X), process(X).
```

```
process(X) :- X == end_of_file, write('session terminated.'), seen.
process(X) :- write(X), nl,
        vars(X, L),         % L is the list of propositional variables in X.
        setof(Y, member(Y, L), S),     % S is L without repetitions.
        evaluate(X, S, Y),
        output(Y), nl,
        loop.

output(true) :- write('is a tautology.'), nl.
output(_) :- write('is not a tautology.'), nl.

evaluate(X, [ ], Y) :- val(X, Y).
evaluate(X, [H|T], Y) :-  replace(H, true, X, A),
                          replace(H, false, X, B),
                          evaluate(A&B, T, Y).

vars(X, [X]) :- atom(X).
vars(~X, Y):- vars(X, Y).
vars(X&Y, C):- vars(X, A), vars(Y, B), append(A, B, C).

% Finish the definition of vars for the operators # and ->.

val(true, true).
val(false, false).

val(~true, false).
val(~false, true).
val(~P, R) :- val(P, Q), val(~Q, R).

val(false&X, false).
val(X&false, false).
val(X&true, Y) :- val(X, Y).
val(true&X, Y) :- val(X, Y).
val(X&Y, R) :- val(X, A), val(Y, B), val(A&B, R).

% Finish the definition of val for the operators # and ->.

replace(P, true, P, true).

replace(P, true, ~P, false).
replace(P, true, ~R, ~Q):- replace(P, true, R, Q).

replace(P, true, P&Q, T):- replace(P, true, Q, T).
replace(P, true, Q&P, T):- replace(P, true, Q, T).
```

```
replace(P, true, R&Q, T&S):- replace(P, true, R, T), replace(P, true, Q,
S).

% Finish the definition of replace(P, true, ...) for wffs that use the
% operators # and ->.

replace(P, true, X, X).

% Complete the definition of replace(P, false, ...).
```

Experiments to Perform

1. What is accomplished by putting the clause replace(P, true, X, X) as the last clause for replacing P by true in a wff?

2. Could we remove the clause replace(P, true, ~P, false) from the program? Why or why not?

3. Could we remove the following two clauses from the program? Why or why not?

```
replace(P, true, P&Q, T):- replace(P, true, Q, T).
replace(P, true, Q&P, T):- replace(P, true, Q, T).
```

4. Finish the definitions asked for in the comments of the program and put the completed program in a file named "program." Then create a file for each of the following files. Test the program by following the instructions in the "Readme" file.

```
This is the "Readme" file for a program to evaluate whether
a propositional wff is a tautology. To run the program, enter Prolog
and then type the following commands:

    |?- [taut].
    |?- main(input).

The wffs use the operators ~, &, #, and -> to stand for not, and, or,
and *  implication, respectively. The input file is a set of wffs, one per
line.
```

```
%  This is a "operators" file that contains definitions for the four
```

```
%  propositional operators  ->, #, &, and ~.

declare :-   op(200, yfx, ->),
             op(150, yfx, #),
             op(100, yfx, &),
             op(50, fy , ~).
```

```
% This is the "taut" file. To test whether the wffs in the file "input"
%  are tautologies, load this file and then type the goal
%
%         main(input).

use_module(library(lists)).
:- [operators], declare, [program].
```

```
% This is the "input" file that contains sample propositional wffs. Note
% that the negation operator needs a preceding space when it follows
% another operator. For example, write p& ~p instead of p&~p.

pp&q&pp->pp# ~q.
p&q&p->p# ~q.
(a&b).
a# ~a.
p& ~p.
p->(q->p).
(p&q)->p.
(p&(p->q))->q.
(~q&(p->q))-> ~p.
(p&(q#r))->((p&q)#(p&r)).
(p->q)&(q->p).
(p# ~q)->p.
```

5. Modify the program so that for any propositional wff the program will write out whether it is a tautology, contingency, or contradiction.

9.6 CNF Generator

Recall that a conjunctive normal form (CNF) is a conjunction of fundamental disjunctions, each of which is a disjunction of literals, where a literal is either a propositional letter or its negation. For example, the following wff is a CNF.

$$p \Box (\neg q \quad p) \Box (q \quad r \quad \neg p).$$

Any propositional wff is equivalent to a CNF. For example,

$$\begin{aligned} p \Box (q \quad r) \Box \ s \ &\equiv \ \neg p \quad \neg (q \quad r) \quad s \\ &\equiv \ \neg p \quad (\neg q \Box \neg r) \quad s \\ &\equiv \ ((\neg p \quad \neg q) \Box (\neg p \quad \neg r)) \quad s \\ &\equiv \ (\neg p \quad \neg q \quad s) \Box (\neg p \quad \neg r \quad s) \\ &\equiv \ (\neg p \quad \neg q \quad s) \Box (\neg p \quad \neg r \quad s) \end{aligned}$$

How can we write a Prolog program to find a CNF for any wff? First of all, Prolog has a predicate "atom" that we can use to define a literal, and thus to define a fundamental disjunction. Let fundis(X) mean that X is a fundamental disjunction and let literal(X) mean that X is a literal. We can define these predicates as follows.

```
fundis(A) :- literal(A).
fundis(A#B) :- fundis(A), fundis(B).

literal(A) :- atom(A).
literal(~A) :- atom(A).
```

Now let cnf(X, Y) mean that X has a CNF Y. We can start the definition of cnf by noticing that a wff is already in CNF if it is a fundamental disjunction.

```
cnf(X, X) :- fundis(X).
```

We'll continue by noticing that the CNF of a conjunction of two wffs is the conjunction of the CNFs of the two wffs.

```
cnf(A&B, A&B) :- fundis(A), fundis(B).
cnf(A&B, C&D) :- cnf(A, C), cnf(B, D).
```

What about negation? We must move negations as far to the right as possible.

```
cnf(~(~A), B) :- cnf(A, B).
cnf(~(A&B), C) :- cnf(~A# ~B, C).
cnf(~(A#B), X&Y) :- cnf(~A, X), cnf(~B, Y).
cnf(~(A->B), C) :- cnf(A& ~B, C).
```

We need to distribute or (#) over and (&).

```
cnf(X#(Y&Z), C) :- cnf((X#Y)&(X#Z), C).
cnf((Y&Z)#X, C) :- cnf((Y#X)&(Z#X), C).
```

Of course, we also need to handle the conditional and disjunction operations.

```
cnf(A->B, C) :- cnf(~A#B, C).
cnf(A#B, C) :- cnf(A, X), cnf(B, Y), cnf(X#Y, C).
```

Experiments to Perform

1. Finish the program by writing a loop to read a file of wffs and convert each wff to CNF. Test the program on a variety of wffs.

2. For each cnf clause in the program, find a wff that uses the clause to calculate its CNF.

3. Argue or give a counterexample to show that the cnf clauses will do the job of finding a CNF for any propositional wff.

9.7 Resolution Theorem Prover for Propositions

The resolution inference rule is nice from a computational point of view because it is a single rule that can be applied repeatedly to prove propositions. The resolution inference rule works something like a cancellation process. It takes two clauses and constructs a new clause from them by deleting all occurrences of a positive literal p from one clause and all occurrences of $\neg p$ from the other clause. For example, suppose we are given the following two propositional clauses:

$$p \quad q,$$
$$\neg p \quad r \quad \neg p.$$

We obtain a new clause by first eliminating p from the first clause and eliminating the two occurrences of $\neg p$ from the second clause. Then we take the disjunction of the leftover clauses to form the new clause:

$$q \quad r.$$

Let's write down the resolution rule in a more general way. Suppose we have two propositional clauses of the following forms:

$$p \quad A,$$
$$\neg p \quad B.$$

Let $A - p$ denote the disjunction obtained from A by deleting all occurrences of p. Similarly, let $B - \neg p$ denote the disjunction obtained from B by deleting all occurrences of $\neg p$. The resolution rule allows us to infer the propositional clause

$$(A - p) \quad (B - \neg p).$$

We'll write the rule as follows.

Resolution Rule for Propositions

$$\frac{p \quad A, \square p \quad B}{\square \ (A \square p) \quad (B \square \square p)}.$$

Although the rule may look strange, it's a good rule. That is, it maps tautologies to a tautology. To see this, we can suppose that

$$(p \quad A) \square (\neg p \quad B) = \text{true}.$$

If p is true, then the equation reduces to B = true. Since $\neg p$ is false, we can remove all occurrences of $\neg p$ from B and still have $B - \neg p$ = true. Therefore

$$A - p \quad B - \neg p = \text{true}.$$

We obtain the same result if p is false. So the inference rule does its job.

A proof by resolution is a refutation that uses only the resolution rule. So we can define a *resolution proof* as a sequence of clauses, ending with the empty clause, in which each clause in the sequence either is a premise or is inferred by the resolution rule from two preceding clauses in the sequence. Notice that the empty clause is obtained when A either is empty or contains only copies of p and when B either is empty or contains only copies of $\neg p$. For example, the simplest version of the resolution rule can be stated as follows.

$$\frac{p, \square p}{\square \ []}.$$

In other words, we obtain the well known tautology $p \square \neg p \square$ false.

For example, let's prove that the following clausal form is unsatisfiable:

$$(\neg p \quad q) \square (p \quad q) \square (\neg q \quad p) \square (\neg p \quad \neg q).$$

In other words, we'll prove that the following set of clauses is unsatisfiable:

$$\{\neg p \quad q, p \quad q, \neg q \quad p, \neg p \quad \neg q\}.$$

The following resolution proof does the job:

Proof:

1.	$\neg p \quad q$	*P*
2.	$p \quad q$	*P*
3.	$\neg q \quad p$	*P*
4.	$\neg p \quad \neg q$	*P*
5.	$q \quad q$	1, 2, Resolution
6.	p	3, 5, Resolution
7.	$\neg p$	4, 5, Resolution
8.	$\square$	6, 7, Resolution

QED.

To program the proof process, we need to access the individual literals in each clause. To do this, we'll represent each clause as a list of it's literals. Thus each wff will be represented as a list of clauses, where each clause is represented as a list of literals. So the process will start out as follows.

1. Read a propositional wff.
2. Find the cnf of the negation of the wff.
3. Represent the cnf as a list of its clauses, where each clause is represented as a list of literals.

Here is the formal proof with the clauses represented as lists of literals.

Formal Proof			*Clause as list of literals*
1.	$\neg p \quad q$	*P*	[¬ p, q]
2.	$p \quad q$	*P*	[p, q]
3.	$\neg q \quad p$	*P*	[¬ q, p]
4.	$\neg p \quad \neg q$	*P*	[¬ p,¬ q]
5.	$q \quad q$	1, 2, Resolution	[q, q]
6.	p	3, 5, Resolution	[p]
7.	$\neg p$	4, 5, Resolution	[¬ p]
8.	$\square$	6, 7, Resolution	[ ]

Let L be the list of clauses representing the cnf of the negation of the original wff. For example, in this proof we have

L = [[¬ p, q], [p, q], [¬ q, p], [¬ p, ¬ q]].

We'll add a new resolvant to L at each proof step. A refutation occurs once the empty clause is found in the list. Let proof(L) return yes if a refutation of L exists. A Prolog definition for proof can be written as follows, where

find_resolvant(L, R) and tries to find a resolvant R for two clauses in L.

```
proof(L) :- member([ ], L).
proof(L) :- find_resolvant(L, R), distinct(R, L), proof([R | L]).
```

The predicate distinct(R, L) checks to see whether the clause R found by find_resolvant is a new clause that does not occur in L. If R is not new, then backtracking will cause find_resolvant to try again. For example, if R = [p, q] and L = [[q, p]], then distinct(R, L) should return no because [p, q] and [q, p] contain the same elements. For another example, the clauses [p, p] and [p] are distinct.

Let's try to define the find_resolvant predicate. We can use the backtracking feature of Prolog to use the member predicate to search for a clause to resolve with the clause at the head of the list of clauses.

```
find_resolvant([C | T], R) :-  member(D, T),
                               resolvable(C, D, Literal),
                               resolve(C, D, Literal, R).
find_resolvant([C | T], R) :- find_resolvant(T, R).
```

The predicate "resolvable" tests whether two clauses can be resolved, and if so returns a selected literal from the first clause. The predicate "resolve" constructs the resolution of two clauses. Here are the definitions, where "negation" is a predicate to return the negation of a literal.

```
resolvable([H | T], C, H) :- negation(H, V), member(V, C).
resolvable([H | T], C, K) :- resolvable(T, C, K).

resolve(C, D, Literal, R) :-    negation(Literal, V),
                                removeAll(Literal, C, A),
                                removeAll(V, D, B),
                                append(A, B, S),
                                makeSet(S, R), !.

negation(~A, A).
negation(A, ~A).
```

Note: The definition for resolve ends with a cut symbol so that the predicate will not be called again on backtracking, which might occur if the resolvant is not distinct from the given proof list. In this case, the backtracking goes to the resolvable predicate which then attempts to find another

Experiments to Perform

1. Write a program to test whether a propositional wff is a tautology by checking whether there is a resolution proof for the negation of the wff. You can use the predicates given in the discussion as a starting point. Then supply the missing definitions for the three predicates: distinct, removeAll, and makeSet.

2. Modify your program so that the clauses of a resolution proof are output for each input wff that is a tautology.

10

Logic Programming Theory

The experiments in this chapter use Prolog to explore some of the theoretical ideas that underlie logic programming.

10.1 The Immediate Consequence Operator

For a definite logic program *P*, the *Herbrand base* of *P* is the set of ground atoms that can be formed by the predicate names, constants, and function names that occur in *P*. The set of atoms in the Herbrand base of *P* that are logical consequences of *P* can be calculated using the immediate consequence operator *Tp*. *Tp* has the power set of the Herbrand base of *P* for its domain and codomain, and it is defined as follows, where I is any set of atoms in the Herbrand base of *P*.

$$Tp(\mathrm{I}) = \{\mathrm{A} \mid \mathrm{A} \leftarrow \mathrm{B}_1, \ldots, \mathrm{B}_n \text{ is a ground instance of a clause in P and } \mathrm{B}_i \in \mathrm{I}\}.$$

If we start with the empty set and keep applying *Tp* to the previous result, we obtain the following sequence of sets that are ordered by inclusion.

$$\varnothing \subseteq Tp(\varnothing) \subseteq Tp(Tp(\varnothing)) \subseteq \ldots \subseteq Tp^k(\varnothing) \subseteq \ldots$$

The symbol $Tp\uparrow k$ is used to denote $Tp^k(\varnothing)$. So the sequence can be written as

$$\varnothing \subseteq Tp\uparrow 1 \subseteq Tp\uparrow 2 \subseteq \ldots \subseteq Tp\uparrow k\ 1 \ldots$$

The union of this sequence is denoted by $Tp\uparrow\omega$ and it equals the set of atoms in the Herbrand base of *P* that are logical consequences of *P*. For example, let

P be the following logic program.

$$p(a) \leftarrow p(x), q(x)$$
$$p(f(x)) \leftarrow p(x)$$
$$q(b) \leftarrow$$
$$q(f(x)) \leftarrow q(x)$$

For this program we can calculate $T_P \uparrow \omega = \{q(f^n(b)) \mid n \in \mathrm{N}\}$.

Experiments to Perform

1. Translate the given program into Prolog. Then make some tests to convince yourself that the calculation of $T_P \uparrow \omega$ is correct.

2. Add the following clause to the given program:

$$p(b) \leftarrow$$

   Let P denote the new larger program. Calculate $T_P \uparrow \omega$ for this new program P. Translate the new program into Prolog. Then make some tests to convince yourself that your calculation of $T_P \uparrow \omega$ is correct.

10.2 Negation as Failure

We'll examine theorems about the soundness and completeness of the negation as failure rule.

Soundness

A normal logic program is a logic program in which the clauses may contain negative atoms in their bodies. A normal goal is a goal that may contain negative atoms. If P is a normal program, then the completion of P is denoted by comp(P). A soundness theorem regarding the negation as failure rule in logic programming can be stated in terms of the program's completion.

> Let P be a normal program and let G be a normal goal. If P $\vdash$ {G} has a finitely failed SLDNF-tree, then G is a logical consequence of comp(P).

This result assumes that a negative literal is picked during resolution only if its atom is grounded. This is called the *safeness* condition. If we drop the safeness condition and allow the computation to proceed without regard to whether a picked negative literal is grounded, then we might not obtain the

results of the soundness theorem. For example, consider the following program:

$$p \leftarrow \neg q(x)$$
$$q(a) \leftarrow$$

Without the safeness condition, the program and goal

$$\leftarrow p$$

have a finitely failed SLDNF-tree. But the theorem does not hold because $\neg p$ is not a logical consequence of comp(P).

Completeness

A completeness theorem regarding the negation as failure rule in logic programming can be stated as follows:

> Let P be a definite program and G be a definite goal. If comp(P) logically implies G, then every fair SLD-tree for $P \cup \{G\}$ is finitely failed.

This result does not extend to normal programs. We'll examine the reasons in the experiments.

Experiments to Perform

1. (*Without the Safeness Condition*). Let P be the following program.

$$p \leftarrow \neg q(x)$$
$$q(a) \leftarrow$$

Consider the goal

$$\leftarrow p.$$

a. Compute comp(P) and show that $\neg p$ is not a logical consequence of comp(P).

b. Translate P and the goal $\leftarrow p$ into Prolog and test whether the safeness condition holds. Use the trace facility to construct the SLDNF-tree.

2. (*With the Safeness Condition*). Let P be the following program, which is similar to the program in Experiment (1).

$$p \leftarrow \neg r$$
$$r \leftarrow q(x)$$
$$q(a) \leftarrow$$

Consider the goal

$$\leftarrow p.$$

In this case, ¬ p is a logical consequence of comp(P).

a. Compute comp(P) and show that ¬ p is a logical consequence of comp(P).

b. Translate the program and goal into Prolog and test whether the safeness condition comes into play. Use the trace facility to construct the SLDNF-tree.

3. Let P be the following normal program:

$$q(a) \leftarrow \neg r(a)$$
$$r(a) \leftarrow p(a)$$
$$r(a) \leftarrow \neg p(a)$$
$$p(x) \leftarrow p(f(x))$$

We will consider the goal $\leftarrow q(a)$.

a. Compute comp(P) and verify that ¬ q(a) is a logical consequence of comp(P).

b. Show that $P \cup \{\leftarrow q(a)\}$ does not have any finitely failed SLDNF-tree.

c. Translate program P into Prolog and test the result of part (b).

10.3 SLDNF-Resolution

We'll examine theorems about the soundness and non-completeness of the SLDNF-resolution rule.

Soundness

The term SLDNF means SLD-resolution with the Negation as Failure rule. A soundness theorem regarding computed answers via SLDNF-resolution in logic programming can be stated as follows:

> Let P be a normal program and let G be a normal goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $comp(P) \cup \{G\}$.

For example, let P be the program

$$p(a) \leftarrow$$
$$q(b) \leftarrow$$

We will consider the goal G:

$$\leftarrow \neg p(x), q(x).$$

Non-Completeness

SLDNF-resolution is not complete. For example, consider the following program P.

$$p(x) \leftarrow$$
$$q(a) \leftarrow$$
$$r(b) \leftarrow$$

We will consider the goal G:

$$\leftarrow p(x), \neg q(x).$$

Experiments to Perform

1. Show that there is an SLDNF-refutation of $P \cup \{G\}$ with computed answer {x/b}. Then show that {x/b} is a correct answer for $comp(P) \cup \{G\}$.
2. Translate the program into Prolog and find out whether Prolog preserves the soundness of SLDNF-resolution in this case.
3. Show that {x/b} is a correct answer for $comp(P) \cup \{G\}$. In other words, show that $p(b) \leftarrow \neg q(b)$ is a logical consequence of comp(P).
4. Translate the program into Prolog and try to verify that {x/b} cannot be an instance of any computed answer for $P \cup \{G\}$.
5. For another example of non-completeness of SLDNF-resolution, consider the following program P.

   $$r(a) \leftarrow p(a)$$
   $$r(a) \leftarrow \neg p(a)$$
   $$p(x) \leftarrow p(f(x))$$

   We will consider the goal G:

   $$\leftarrow r(a).$$

   a. Show that the empty substitution is a correct answer for

   $$comp(P) \cup \{G\}.$$

   In other words, show that r(a) is a logical consequence of comp(P).

   b. Translate the program into Prolog and try to verify that the empty substitution cannot be an instance of any computed answer for

   $$P \cup \{G\}.$$

Answers to Selected Experiments

Chapter 2

2.5 Family Trees

1. |?- p(a, e). means "Is *a* a parent of *e*?"
|?- p(X, e). means "Who is a parent of *e*?"
|?- p(a, X). means "Who is a child of *a*?"
|?- g(a, T). means "Who is a grandchild of?"
|?- g(M, e). means "Who is a grandparent of *e* ?"
|?- g(U, V). means "Is there a grandparent-grandchild pair?"

2a. ch(X, Y) :- p(Y, X).

2b. gch(X, Y) :- g(Y, X).

3a. co(X, Y) :- p(A, X), p(B, Y), sib(A, B).

3b. sco(X, Y):- p(A, X), p(B, Y), co(A, B).

Chapter 3

3.1 The Ancestor Problem

4. ancestor(X, Y, 1) :- p(X, Y).
ancestor(X, Y, N) :- p(X, W), ancestor(W, Y, M), N is M + 1.

3.3 Switching Pays

4.
```
rands(_, _, 0):- write(' Done with trials.'), nl.
rands(A, B, N) :-
        N >= 1,
        random(A, B, Out),
        write(Out), nl,
        K is N - 1,
        rands(A, B, K).
```

3.4 Inductively Defined Sets

1. S = {2, 5, 8, ... } = {2 + 3k | k ∈ N}.
3. a. A = {3, 7, 15, 31, 63, ... } = {3 + 2^n | n > 0 and n ∈ N}.
b. A = {4x + 9y | x, y ∈ N}.

```
inA(0).
inA(X) :- Y is X - 4, Y >= 0, a(Y).
inA(X) :- Y is X - 9, Y >= 0, a(Y).
```

6. All elements of U can be recognized by the program

```
inU(a).
inU(b).
inU(h(X, Y)) :- inU(X), inU(Y).
```

But the goal inU(X) will not generate all elements of U by backtracking because only the left side of the computation tree, which is infinite, will be traversed on backtracking.

Chapter 4

4.1 Negation and Inferences in Prolog

5. The experiment is a proper test for hypothetical syllogism because we have the following equivalence

$$[((p \to q) \to (q \to r) \to p) \to r] \equiv [(p \to q) \to (q \to r) \to (p \to r)].$$

4.2 The Blocks World

1. The answers we might expect are: yes, no, X = a, no, yes, X = b.
The third and sixth goals might have problems because negation is used without the varible being instantiated. In these cases the goals and answers are as follows:

```
|?- onTop(X).
no
|?- \+ onTop(X).
yes
```

2b.

```
bottom(A, A):- on(A, [ ]).
bottom(A, Y):- on(A, X), bottom(X, Y).
```

2c.

```
move_ordered(A, B) :-
      onTop(A), onTop(B),
      bottom(A, X),
      retract(on(X, [ ])), assertz(on(X, B)).
```

2d.

```
move_reversed(A, B) :-
      onTop(A), onTop(B),
      retract(on(A, X)), assertz(on(A, B)),
      move_reversed(X, A).
move_reversed([ ], _).
```

4.3 Verifying Arguments in First-Order Logic

1. (Committees) Let $c(x)$ mean that x is a committee member, $r(x)$ mean that x is rich, $o(x)$ mean that x is old, and $f(x)$ mean that x is famous. Then the argument can be formalized as follows:

$$\exists x\,(c(x) \wedge r(x) \wedge f(x)) \wedge \forall x\,(c(x) \rightarrow o(x)) \rightarrow \exists x\,(c(x) \wedge o(x) \wedge f(x)).$$

To see whether prolog will verify such an argument, we need to do some rewriting of the first premise:

$$\begin{aligned}
\exists x\,(c(x) \wedge r(x) \wedge f(x)) &\equiv \exists x\,(\neg c(x) \vee (r(x) \wedge f(x)) \\
&\equiv \exists x\,((\neg c(x) \vee r(x)) \wedge (\neg c(x) \vee f(x)) \\
&\equiv \exists x\,((\neg c(x) \vee r(x)) \wedge \exists x\,((\neg c(x) \vee f(x)) \\
&\equiv \exists x\,((c(x) \rightarrow r(x)) \wedge \exists x\,((c(x) \rightarrow f(x)).
\end{aligned}$$

The latter wff can be written in prolog as the following two facts:

```
r(X) :- c(X).
f(X) :- c(X).
```

The second premise can be written as the following prolog facts:

```
c(a).
o(a).
```

The goal corresponding to the conclusion can be written in prolog as follows:

```
|?- c(X), o(X), f(X).
```

2. (quadrupeds). Let $h(x)$ mean that x is a human, $m(x)$ mean that x is a man, and $nq(x)$ mean that x is not a quadruped. Then the argument can be formalized as follows:

$$\exists x\,(h(x) \wedge nq(x)) \wedge \forall x\,(m(x) \rightarrow h(x)) \rightarrow \exists x\,(m(x) \wedge nq(x)).$$

The two premises can be written in prolog as the following two facts:

```
nq(X) :- h(X).
h(X) :- m(X).
```

To verify the conclusion, we can use two cases. If a is a being such that m(a) is false, then $m(a) \rightarrow nq(a)$ is true. On the other hand, if a is a being such that

m(a) is true, then the two premises of the experiment tell us that $m(a) \rightarrow nq(a)$ is true. This can be verified by writing the fact

m(a).

and then executing the goal

|?- nq(a).

3. (freshman). Let $f(x)$ mean that x is a freshman, $s(x)$ mean that x is a sophomore, and $j(x)$ mean that x is a junior, $nj(x)$ mean that x is not a junior, and $l(x, y)$ mean that x likes y. Then the argument can be formalized as follows.

$$\forall x\,(f(x) \rightarrow \exists y(s(y) \wedge l(x, y))) \rightarrow \exists x \forall y(f(x) \rightarrow l(x, y) \rightarrow nj(y)) \rightarrow \forall x\,(s(x) \rightarrow nj(x)).$$

The premises can be written in prolog as the following facts:

```
f(a).
l(a, Y) :- s(Y).
nj(Y) :- f(X), l(X, Y).
```

We also need the following fact to test for the existence of juniors:

```
j(X) :- \+ nj(X).
```

To verify the conclusion, we can use two tests. If there are no sophomores, then certainly no sophomore is a junior. This can be verified with the following goal.

```
|?- s(X), j(X).
```

For the other case, we'll assume the existence of some sophomore by writing the fact

```
s(b).
```

To see that b is not a junior, we should get a yes-answer to the goal

```
|?- nj(b).
```

4.4 Equality Axioms

2. $e(t, u)$.
$e(u, v)$.
$e(A, B)$:- $e(A, C)$, $e(C, B)$.

Chapter 5

5.1 List and String Notation

3. q([b]).
q([b, b|X]) :- q(X).

5.2 Sets and Bags of Solutions to a Query

2. When Y does not occur in the first argument and it occurs free in the second argument of setof, the goal returns the set S for each particular value of Y. This is similar to the situation in formal logic where an interpretation for a wff that has a free variable must assign that variable to a constant.

3b. newsetof(X, P, S):- setof(X, P, S).
newsetof(_, _, [ ]).

4e. setof(p(X, Y, Z, N), (p(X, Y, Z, N), N < 100), S).

5.3 List Membership and Set Operations

6. minus([ ],_,[ ]).
minus(A, B, S):- setof(X, (member(X, A), \+ member(X, B)), S).
minus(X, X, []).

5.4 List Operations

3. first([X], [ ]).
first([H | T], [H | X]) :- first(T, X).

4. pairs([ ], [ ], [ ]).
pairs([H | T], [J | S], [[H, J] | X]) :- pairs(T, S, X).

5. dist(A, [ ], [ ]).
dist(A, [H | T], [[A, H] | X]) :- dist(A, T, X).

6. prod([ ], Y, [ ]).
prod([H | T], Y, W) :- dist(H, Y, A), prod(T, Y, B), cat(A, B, W).

7. replace(A, B, [ ], [ ]).
replace(A, B, [A | T], [B | S]) :- replace(A, B, T, S).
replace(A, B, [H | T], [H | S]) :- replace(A, B, T, S).

8. This solution uses the append predicate that is in the "lists" library. It also uses a predicate "dist" that distributes an element into a set of sets. For example, the goal

```
|?- dist(a, [[b, c], [d, e, f]], X).
```

will return X = [[a, b, c], [a, d, e, f]]. Here is the program.

```
:- use_module(library(lists)).

power([ ], [[ ]]).
power([H | T], C) :- power(T, A), dist(H, A, B), append(A, B, C).
dist(X,[Y], [[X | Y]]).
dist(X,[H | T], [[X | H] | Z]) :- dist(X, T, Z).
```

9. The goal |?- outLine(L, N, K) means print out N elements from list L and return K as the list of the remaining elements from L.

```
outLine(L, 0, L).
outLine([H|T], N, K) :- print(H), tab(2), M is N-1, outLine(T, M, K).
```

Chapter 6

6.1 Binary Trees

2b.
```
in(t, [ ]).
in(t(A, B, C), X) :- in(B, S), in(C, T), append(S, [A|T], X).
```

2c.
```
post(t, [ ]).
post(t(A, B, C), X) :- post(B, S), post(C, T),
                       append(S, T, Y), apppend(Y, [A], X).
```

4.
```
isIn(N, t(N, _, _)).
isIn(N, t(A, B, C)) :- N < A, isIn(N, B).
isIn(N, t(A, B, C)) :- N > A, isIn(N, C).
```

6.2 Arranging Objects

6. One solution replaces the third clause of insert with

```
insert(A, [H|T], [H|S]) :- A > H, insert(A, T, S).
```

Another solution places a cut at the end of the second clause of insert:

```
insert(A, [H|T], [A|[H|T]]) :- A =< H, !.
```

6.3 Simple Ciphers

2b. Here is a simple loop to do the job. It can be executed by the call

```
|?- loop(1).

loop(27).
loop(K) :-  cipher(abcdefghijklmnopqrstuvwxyz, K, S),
            distinct(S, L),
            print(S), tab(3), print(L), nl,
            M is K + 1,
            loop(M).
```

3. An affine cipher can be defined as follows, where "additive" and "multiplicative" are additive and a multiplicative cipher, respectively.

```
affine(In, AddKey,MultKey, Out) :-
                                  additive(In, AddKey, X),
                                  multiplicative(X, MultKey,Out).
```

6.4 The Birthday Problem

1. The goal dup(L, D) returns D as a list of duplicates, if any, that occur in the list L. The "member" predicate tests membership in a list and the the "remove" predicate removes all occurrences of an element from a list. These predicates must be written too.

```
dup([ ], [ ]).
dup([H|T], [H|S]) :- member(H, T), remove(H, T, U), dup(U, S).
dup([H|T], S) :- dup(T, S).

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

Chapter 7

7.4 Arithmetic Expressions

3. We can define a grammar for strings of spaces as follows:

```
space --> [ ] | " ", space.
```

Now we can insert space where ever we wish to allow zero or more spaces. For example, to allow spaces on either side of the – symbol, we would change the production

```
expr --> nat, "–", expr.
```

to

```
expr --> nat, space, "–", space, expr.
```

5.

```
expr(A) --> term(B), r(C), {eval([B|C],A)}.
r(A) --> "+",term(B), r(C), {A = [B,+|C]}.
r(A) --> "–",term(B), r(C), {A = [B,–|C]}.
r(A) --> [ ], {A = [ ]}.

term(A) --> factor(B), t(C), {eval([B|C],A)}.
t(A) --> "*", factor(B), t(C), {A = [B, *|C]}.
t(A) --> [ ], {A = [ ]}.

factor(A) --> "(", expr(B), ")", {A is B}.
factor(A) --> nat(B, T), {A is B}.

nat(X, 1) --> dig(X).
nat(X, T) --> dig(A), nat(B, S), {T is 10*S, X is A*T + B}.
```

```
dig(X) --> [D],{"0" =< D, D =< "9", X is D – "0"}.

eval([A], A).
eval([A, B, +|T],Ans) :- X is A + B, eval([X|T], Ans).
eval([A, B, –|T],Ans) :- X is A – B, eval([X|T], Ans).
eval([A, B, *|T],Ans) :- X is A * B, eval([X|T], Ans).
```

Chapter 8

8.2 Nondeterministic Finite Automata

5. To add the capability of executing instructions that include a list of next states (e.g., t(0, a, [2, 3])), add two more path clauses at the end of the definition to obtain the following program.

```
accept(S) :- start(I), path(I, S).
path(K, [ ]) :- final(K).
path(K, [H|T]) :- t(K, H, N), path(N, T).
path(K, X) :- t(K, [ ], N), path(N, X).
path([K|M], X) :- path(K, X).
path([K|M], X) :- path(M, X).
```

8.3 Mealy Machines

2. The goal mealy(X, Y) yields, upon backtracking, pairs of the form

$$X = Y = [\,] \text{ and } X = [a, a, \ldots, a],\ Y = [a].$$

The goal mealy(X, [a]) yields inputs of the form X = [a, a, ..., a]. The goal mealy(X, [b]) yields inputs of the form X = [b, b, ..., b]. The goal mealy(X, [c]) returns no.

4.

```
t(0, a, [ ], 0)      t(1, a, n, 0)      t(2, a, n, rn)
t(0, b, [ ], 0)      t(1, b, n, 0)      t(2, b, n, rn)
t(0, n, [ ], 1)      t(1, n, [ ], 2)    t(2, d, n, 3)
t(0, d, [ ], 4)      t(1, d, [ ], 3)    t(2, n, [ ], 3)
                                        t(rn, [ ], n, 0)

t(3, a, alpha, 0)    t(4, a, d, 0)
t(3, b, beta, 0)     t(4, b, d, 0)
t(3, n, n, 3)        t(4, n, [ ], 3)
t(3, d, d, 3)        t(4, d, n, 3)
```

8.4 Moore Machines

2. The goal moore(X, Y) yield s, upon backtracking, yields double pairs of the form X = Y = [ ], X = [ ], Y = [[ ]], and X = [a, a, ..., a], Y = [b, b, ..., b]. The goal mealy(X, [a]) yields inputs of the form X = [a, a, ..., a]. The goal moore(X, [a, b, a, b]) yields X = [b, a, b, a] twice, then returns no. The goal moore(X, [c]) returns no.

8.5 Pushdown Automata

3. Replace the clause

path(*K*, [ ], Stack) :- final(*K*).

with the clause

path(*K*, [ ], [ ]).

8.6 Turing Machines

3. Add the following "find" clause that executes upon failure of the first one.

```
find(State, Left, Cell, Right, OutTape)
        :-  reverse(Left, R), append(R, [Cell|Right], OutTape),
            write('Error: cannot execute instruction of the form'), tab(2),
            write(t(State, Cell, _, _, _)).
```

Chapter 9

9.1 Lambda Closure

2. newClosure(In, Out) :- states(S), closure(In, X, Out).

4. newClosure(In, Out) :- states(S), closure(In, X, Out).
states(S):- s([ ], S).
s(X, Y) :- t(A, _, _), \+ member(A, X), s([A|X], Y).
s(X, X).

9.2 Transforming an NFA into an equivalent DFA

1a.
```
outPut(S,T,F, File) :-  tell(File),
                        write('start('), write(S), write(').'), nl,
                        outDFA(T),
                        outFinals(F),
                        told.

outDFA([H|T]):-write(H), write('.'), nl, outDFA(T).
outDFA([ ]).

outFinals([H|T]):- write('final('), write(H), write(').'), nl, outFinals(T).
outFinals([ ]).
```

4.
```
replaceStates([ ],_,[ ]).
replaceStates([t(S, A, T) | Tail], Assoc, [t(N, A, M) | Rest]) :-
        member([S, N], Assoc), member([T, M], Assoc),
replaceStates(Tail, Assoc, Rest).

associate(N, [ ], [ ]).
associate(N, [H | T], [[H, N] | X]) :- M is N+1, associate(M, T, X).

makeFinals(F, Assoc, Out) :-
    setof(X, H^(member(H, F), member([H, X], Assoc)), Out).
```

9.4 Defining Operations

The symbols xfy, yfx, and xfx mean that the operator is right associative, left associative, and not associative, respectively. A higher number in the first argument of "op" indicates a lower precedence of evaluation. The symbol fx means that the prefix unary operator cannot be applied to itself. The symbol fy means that the prefix unary operator can be applied to itself.

9.5 Tautology Tester

1. The clause replace(P, true, X, X) is needed to return X if P does not occur in X. For example, replace(p, true, p&q, X) calls replace(p, true, q, X), which returns X = q.

2. The clause replace(P, true, ~P, false) could be removed from the program. Then a call like replace(p, true, ~p, X) would return X = ~true, which would eventually be evaluated to false by the val predicate.

3. Both clauses can be removed from the program. For example, a call like replace(p, true, p&q, X) would be computed by the clause

```
replace(P, true, R&Q, T&S):-  replace(P, true, R, T),
                              replace(P, true, Q, S).
```

and return X = true&q instead of X = q. Eventually the val predicate will evaluate either version of X to the same value.

5. Modify the output predicate as follows:

```
output(X, S, true):- write('is a tautology.'), nl.
output(X, S, false):- evaluate(~X, S, Y), secondOutput(Y).
secondOutput(true):- write('is a contradiction.'), nl.
secondOutput(_):-  write('is a contingency.'), nl.
```

9.6 CNF Generator

1. The loop can be written as follows:

```
main(InFile) :- see(InFile), loop.
loop :- read(X), process(X).
process(X):- X = end_of_file, write('session terminated.'), nl, seen.
process(X) :- write(X), nl, cnf(X, Y), write(Y), nl, nl, loop.
```

9.7 Resolution Theorem Prover for Propositions

2. To output the proof list modify the definition for "proof" as follows:

```
proof(L, L) :- member([ ], L).
proof(L, K) :- find_resolvant(L, R), distinct(R, L), proof([R|L], K).
proof(L, L).
```

Chapter 10

10.1 The Immediate Consequence Operator

2. $T_P \uparrow \omega = \{p(f^n(a)) \mid n \in \mathbb{N}\} \cup \{p(f^n(b)) \mid n \in \mathbb{N}\} \cup \{q(f^n(b)) \mid n \in \mathbb{N}\}$.

Index