Prolog
A Tutorial Introduction

James Lu
Jerud J. Mead

Computer Science Department
Bucknell University
Lewisburg, PA 17387

Contents

1 Introduction

Logic programming is a programming paradigm based on mathematical logic. In this paradigm the programmer specifies relationships among data values (this constitutes a logic program) and then poses queries to the execution environment (usually an interactive interpreter) in order to see whether certain relationships hold. Putting this in another way, a logic program, through explicit facts and rules, defines a base of knowledge from which implicit knowledge can be extracted. This style of programming is popular for data base interfaces, expert systems, and mathematical theorem provers. In this tutorial you will be introduced to *Prolog*, the primary logic programming language, through the interactive *SWI-Prolog* system (interpreter).

You will notice that *Prolog* has some similarities to a functional programming language such as *Hugs*. A functional program consists of a sequence of function definitions — a logic program consists of a sequence of relation definitions. Both rely heavily on recursive definitions. The big difference is in the underlying execution "engine" — i.e., the imperative parts of the languages. The execution engine of a functional language evaluates an expression by converting it to an acyclic graph and then reducing the graph to a normal form which represents the computed value. The *Prolog* execution environment, on the other hand, doesn't so much "compute" an answer, it "deduces" an answer from the relation definitions at hand. Rather than being given an expression to evaluate, the *Prolog* environment is given an expression which it interprets as a question:

> *For what parameter values does the expression evaluate to* true*?*

You will see that *Prolog* is quite different from other programming languages you have studied. First, *Prolog* has no types. In fact, the basic logic programming environment has no literal values as such. Identifiers starting with lower-case letters denote data values (almost like values in an enumerated type) while all other identifiers denote variables. Though the basic elements of *Prolog* are typeless, most implementations have been enhanced to include character and integer values and operations. Also, *Prolog* has mechanisms built in for describing tuples and lists. You will find some similarity between these structures and those provided in *Hugs*.

2 Easing into *Prolog*

In this section you will be introduced to the basic features of the logic programming language *Prolog*. You will see how to define a knowledge base in terms of facts (unconditionally true) and rules (truth is conditional), how to load the knowledge base into the *SWI-Prolog* system, and how to query the system.

2.1 Logic Programming

Remember that all programming languages have both declarative (definitional) and imperative (computational) components. *Prolog* is referred to as a declarative language because all program statements are definitional. In particular, a *Prolog* program consists of *facts* and *rules* which serve to define relations (in the mathematical sense) on sets of values. The imperative component of *Prolog* is its execution engine based on *unification* and *resolution*, a mechanism for recursively extracting sets of data values implicit in the facts and rules of a program. In this section you will be briefly introduced to each of these terms.

Facts and Rules

Everything in *Prolog* is defined in terms of two constructs: the *fact* and the *rule*. A *fact* is a *Prolog* statement consisting simply of an identifier followed by an n-tuple of constants. The identifier is interpreted as the name of a (mathematical) relation and the fact states that the specified n-tuple is in the relation. In *Prolog* a relation identifier is referred to as a *predicate*; when a tuple of values is in a relation we say the tuple *satisfies* the predicate.

As a simple example, consider the following *Prolog* program which defines a directed graph on five nodes – this first example involves only facts.

Example 1 – **A Directed Graph I**

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

This program contains six facts: `edge(a,b)`, `edge(a,e)`, ..., `edge(e,b)`. Intuitively, these six facts describe the directed graph which has edges from node `a` to `b`, `a` to `e`, `b` to `d`, and so on. The identifier `edge` is a predicate and the six facts define the edge relation on the graph; the identifiers `a, b, c, d, e` are abstract constant values representing names for the graph nodes. The constant values are similar to what you might define in an enumerated type in Pascal or *C*).

∎

A fact states that a certain tuple of values satisfies a predicate *unconditionally*. A *rule*, on the other hand, is a *Prolog* statement which gives conditions under which tuples satisfy a predicate. In a sense, a fact is just a special case of a rule, where the condition is always satisfied (i.e., equivalent to "true").

A graph is very interesting, but not of much use if there is no way to specify more complex properties other than edge. The following example illustrates a rule which defines the property of "path of length two" between two edges.

Example 2 – **A Directed Graph II**

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

```
tedge(Node1,Node2) :-
      edge(Node1,SomeNode),
      edge(SomeNode,Node2).
```

The last three lines of this program constitute a rule which defines a new predicate `tedge`. The left side of the rule looks just like a fact, except that the parameters of the fact are capitalized indicating they are variables rather than constants. The right side of the rule consists of two terms separated by a comma which is read as "AND". The way to read this rule is as follows.

> The pair `(Node1,Node2)` satisfies the predicate `tedge` if there is a node `SomeNode` such that the pairs `(Node1,SomeNode)` and `(SomeNode,Node2)` both satisfy the predicate `edge`.

∎

But these examples are not very convincing since we don't know how a *Prolog* system would process the facts and rules. In the next section we will investigate a *Prolog* system and see how we make use of the program above to determine properties of the graph described in the facts.

2.2 The *SWI-Prolog* Environment

The version of *Prolog* that we will use is called *SWI-Prolog*, developed at the Swedish Institute of Computer Science. The *SWI-Prolog* environment is an interactive system, much like the *Hugs* functional programming environment. It can be executed in two different forms: from the command line or via an Emacs interface. In either case, before moving ahead you should add the following line to the alias section of your `.cshrc` file (or whatever your shell's startup file is called):

```
alias prolog '/usr/local/bin/prolog -f none'
```

Don't forget to activate this alias by entering the shell command

```
source ~/.cshrc
```

Emacs — skip if you don't use emacs

A GNU Emacs mode for SWIprolog is available. If you will not be using *Prolog* from the emacs editor you can skip to the next section.

First, you must add the following lines (as is) to your `.emacs` file in your home directory. Put these lines in the section labeled "`Hooks`".

```
(setq auto-mode-alist (append
                       (list
                        (cons "\\.pl$" 'prolog-mode))
                       auto-mode-alist))

(autoload 'prolog-mode "~/emacs/prolog.el" "" t)
```

When you have added these lines and load a file whose name ends in `'.pl'`, emacs will automatically go into Prolog mode.

To use the emacs mode you create your program, giving it a name with a `.pl` extension. Once you have created your program, you may invoke the interpreter by the emacs command

```
Control-c Shift-c
```

This will activate *Prolog* and at the same time load your program into the interpreter. When loading finishes, you may begin querying your program. Remember, every rule in *Prolog*, as well as each query, ends with a period.

The Emacs mode provides the following commands:

```
M-x run-prolog  Run an inferior Prolog process,
                input and output via buffer *prolog*.

C-c K           prolog-compile-buffer
C-c k           prolog-compile-region
C-c C-k         prolog-compile-predicate
C-c C           prolog-consult-buffer
C-c c           prolog-consult-region
C-c C-c         prolog-consult-predicate
```

Using the interpreter

To start the *Prolog* interpreter from the command line you enter the command `prolog`. After a bit of initialization you will see appear the *SWI-Prolog* prompt:

```
| ?-
```

Exiting the system is easy. Just enter '`halt.`' at the prompt (not the single quote marks!). Do this now and then restart *Prolog*.

Before you can test out the capabilities of the system you must have a *Prolog* program to work with. Use your text editor to enter the program in Example 2.1. Be sure to give it a name with the a `.pl` extension — let's assume you name it '`first.pl`'. To load a file into the interpreter you enter the following, being sure not to forget the final '.'.

```
| ?- ['first'].
```

Notice that the `.pl` extension is left off — and don't forget the single quotes, square brackets, and the period!

The *SWI-Prolog* interpreter is similar to the *Hugs* interpreter in that a program contains a series of definitions (read from a file). The *Hugs* interpreter allows the user to compute an *expression* based on the (function) definitions in the program. In the same way, the *SWI-Prolog* interpreter reads a program and then allows the user to compute a *query* based on the definitions (facts and rules). A query "asks" the system if there are any data values which satisfy a particular predicate (or combination).

Let's give this a try.

1. At the *SWI-Prolog* prompt enter the following:

   ```
   ?- edge(a,b).
   ```

   When you hit return the system should respond '`Yes`' and put up another prompt. Try another which should succeed. What you are entering are queries. The first asks the system if the tuple `(a,b)` satisfies the predicate `edge`, and, of course, it does. Try a couple which should fail. What happens if you use a node name which does not occur in one of the facts? You might expect an error, but it just says '`No`'!

2. You can enter compound queries. How does the system respond to these queries?

   ```
   ?- edge(a,b), edge(a,e).
   ?- edge(a,b), edge(a,c).
   ```

   What is your conclusion about the meaning of the '`,`' separating the two parts?

3. Now to try out the rule. The predicate `tedge` defines the notion of "path of length two" between two nodes. Come up with two queries based on `tedge` which should succeed and two which should fail.

4. Now for one final demonstration of the power of the *Prolog* system. When an identifier is used in a program and it starts with a lower-case letter, that identifier denotes a constant value. Identifiers which start with an upper-case letter denote variables. Variables are used in queries to get the system to find all values which can be substituted so to make the resulting predicate true.

   Try the following query.

```
?- edge(a,X).
```

When you press return the system should respond with a value for `X`. If you press the semi-colon (no return!) you are asking if there are anymore values which can satisfy the query. If there are more another will be written; if there are no more, then `'No'` will be written. If you press return rather than the semi-colon, then no more answers will be sought. This ability of *Prolog* to find all possible answers is a powerful computational tool.

Now to really put the idea to the test try the following query and see how many solutions you get.

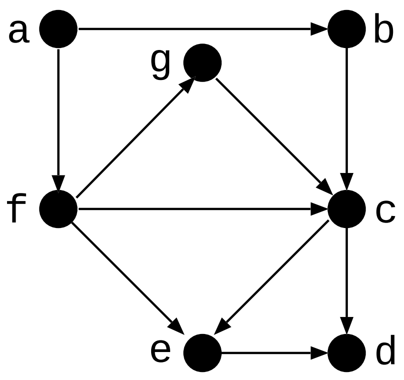
```
?- edge(X,Y).
```

5. *Prolog* contains some built in relations. For example, the relations `'<,<=,==,>=,>` may be used for testing the usual arithmetic relationships on numbers. The query `'1 < 3'` succeeds while `'3 < 3'` fails. Try a few of these.

<u>Problem 1</u>

Following the example above, write a *Prolog* program which describes a directed graph with the following structure. Also include in the program a predicate which defines the 'path of length 3' relation. Define it in two ways: first only referencing the `edge` predicate, and then making use of the `tedge` predicate.

Implement and test.



∎

3 A Closer Look at *Prolog*

In the previous section you worked with *Prolog* in the context of a simple example and learned about the basic components of *Prolog*: facts, rules, and the interpreter. In this section you will take a closer look at these same components.

3.1 Facts

There's not a lot more to be said about facts. It is important to realize, however, that if a predicate appears in a sequence of facts, that doesn't mean that it cannot be the head of a (non-fact) rule as well. For example, we can take the predicates `edge` and `tedge` from Example 2.1 and define a new, more complex graph.

```
edge(a,b).
edge(a,e).
edge(b,d).
edge(b,c).
edge(c,a).
edge(e,b).
edge(X,Y) :- tedge(X,Y).

tedge(Node1,Node2) :-
      edge(Node1,SomeNode),
      edge(SomeNode,Node2).
```

Add this new rule to your program and see what effect it has on the results of queries such as

```
?- edge(a,c).
?- edge(a,b).
?- edge(e,c).
```

Notice that each of these queries will respond `'yes'`. We will avoid queries which we know should respond `'no'` for a bit because they can behave in odd ways.

3.2 Rules

The real power of any programming language is in its abstraction mechanism. In *Prolog* the *rule* provides the mechanism for abstraction. Where a fact can only specify that a tuple of values satisfies a predicate, a rule can specify under what conditions a tuple of values satisfies a predicate.

The basic building block of a rule is called an *atom*: a predicate followed by a tuple of terms, where both constants and variables are terms (we will see that a term can actually have more structure). We see immediately that every fact is an atom. When the tuple satisfies the predicate of the atom we say the atom is true. Every rule is divided into two parts by the symbol '`:-`': the left-hand side is called the *head* of the rule while the right-hand side is the *body* of the rule. In general, we read the rule

```
Atom :- Atom1, ..., Atomn
```

as

if each of `Atom1,...,Atomn` is true, then `Atom` is also true.

We see this structure in the rule defining `tedge`.

```
tedge(Node1, Node2) :- edge(Node1, SomeNode), edge(SomeNode, Node2).
```

There are three atoms in this rule, with `tedge(Node1, Node2)` being the head of the rule and the body of the rule consisting of the atoms `edge(Node1, SomeNode)` and `edge(SomeNode, Node2)`.

recursive rules

Not surprisingly it is recursion which gives rules their real power. Let's look at a more interesting extension to the directed graph program. Suppose we aren't satisfied with a predicate which distinguishes pairs of nodes linked by paths of length two. Suppose we want a general predicate which is satisfied by a pair of nodes just in case they are linked by a path in the graph – a path of any (positive) length. Thinking recursively, we can see that there is a path from one node to another if there is an edge between them (a base case), or if there is an edge to an intermediate node from which there is a path to the final node. The following two rules define the path relation for our program.

```
path(Node1,Node2) :-
      edge(Node1,Node2).
path(Node1,Node2) :-
      edge(Node1,SomeNode),
      path(SomeNode,Node2).
```

There are two important characteristics in this example. First, the use of two rules (with the same head) to define a predicate reflects the use of the logical "OR" in a *Prolog* program. We would read these two rules as:

> `path(Node1,Node2)` is true if `edge(Node1,Node2)` is true **or** if there is a node `SomeNode` for which both `edge(Node1,SomeNode)` and `path(SomeNode,Node2)` are true.

Second, the predicate used in the head of the second rule also appears in the body of that rule. The two rules together show how a recursive definition can be implemented in *Prolog*.

Problem 2

1. Add the predicate `path` to the program you wrote for Problem 1 in Section 2.2. Test it out to see that it works as expected. Test on nodes which are connected by paths! [You can also test it on non-connected pairs, but beware of odd behavior.]

2. Modify the definition of `path` so that paths of length zero are allowed. [`FACT`!]

■

3.3 Computation

The previous sections have illustrated how *Prolog*'s declarative statements (facts and rules) can be used to write a program which defines a base of knowledge, in this case, knowledge about a particular directed graph. But a program is of little use if we cannot extract or deduce useful implicit knowledge from it. In the above example, for instance, we might want to deduce all the nodes which can be reached by a path of length less than four starting from node `d`. The imperative or computational component of *Prolog* makes such a deduction possible.

For the sections which follow we will make use of the program discussed above (complete with `path` predicate). For convenience it is presented here with line numbers added.

Example 3 – **A Directed Graph III**

```
1   edge(a,b).
2   edge(a,e).
3   edge(b,d).
4   edge(b,c).
5   edge(c,a).
6   edge(e,b).

7   tedge(Node1,Node2) :-
          edge(Node1,SomeNode),
              edge(SomeNode,Node2).

     8   path(Node1,Node2) :-
              edge(Node1,Node2).
     9   path(Node1,Node2) :-
              edge(Node1,SomeNode),
              path(SomeNode,Node2).
```

∎

Ground Queries

Computation in *Prolog* is facilitated by the *query*, simply a conjunction of atoms. A query represents a question to a *Prolog* program: is the query true in the context of the program? As an example, consider the following query.

```
edge(a,b)
```

This query is called a *ground* query because it consists only of value identifiers as parameters to the predicate. When a ground query is posed we expect a `yes/no` answer. How is the answer determined?

In response to the query the *Prolog* system scans the program looking for a rule or fact which "matches" the query. if we look at the program above we see that the query matches identically the fact on line 1, so the query must be true in the context of the program.

That was easy, but how about this query?

```
path(a,b)
```

It is also a ground query, but when we scan the program we don't find a rule which exactly matches it. The crucial issue here is what we should mean by "match". When we scan the program we only have two rules, 8 and 9, with the predicate `path` in the head. Of course, we know that a rule is supposed to mean that the head is true if the body is true. So, if `path(a,b)` is to be true it will have to be true based on those rules defining `path` — i.e., those rules with `path` as the head. But the heads of each of these rules contain variables, not data constants. Not surprisingly, *Prolog* finds a match when constant and variable identifiers coincide. In this case we see that if variables `Node1` and `Node2` are replaced by `a` and `b`, respectively, then the body of rule 8 is true (`edge(a,b)` is a fact!). Then the head of the rule with the same substitution must be true. *Prolog* should conclude that the query is true.

To summarize: when *Prolog* tries to deduce a query it begins by searching for rules with heads which match the predicate of the query. This search is always done from top to bottom, so order of definition does matter. A match is found if the parameters to the query are identical to those of the head of a rule or if the corresponding parameter in the rule head is a variable.

But a query needn't be a single atom; the definition clearly states a query is a conjunction of atoms. But simple logical properties indicate that a conjunction of atoms is true just in case every atom in the conjunction is true. So to determine whether a query is true, the *Prolog* system must determine whether each atom in the query is true.

non-Ground Queries

The ground queries mentioned in the previous section are very easy to verify from the program. But queries can also have variables as parameters; such queries are called *non-ground* queries. Consider the following non-ground query.

```
tedge(a,X)
```

The atom in this query contains as arguments one variable and one constant. Is the query true? Well, its not clear that the question even makes sense. Certainly, if we replace `X` by the constant `d` then the query is true. This is the notion of *satisfiability.* We say that the query is satisfiable relative to the program because there is a substitution for its variable which makes the query true.

Reasoning out this query takes a bit of work; follow closely (write the sequence down yourself!). When we scan the program we find one rule, rule 7, which defines `tedge`: we focus on this rule. The query almost looks like the head of the rule. We write down the substitution

```
Node1 = a, X = Node2
```

and ask whether `tedge(a,Node2)` is true. Of course, the meaning of the rule says that this atom should be true if the body (with the same substitutions made) is also true. So the truth of our original query can be replaced by the truth of a new query.

```
edge(a,SomeNode), edge(SomeNode,Node2)
```

We need to assess the truth of the two atoms in our new query. Let's first consider `edge(a,SomeNode)`. As we scan the program we see that there are two possible facts which fit; we'll take the first one, `edge(a,b)`. If we take the substitution

```
SomeNode = b
```

and make the replacement in our query, then the first atom of the query is satisfied. Now for the second atom. After the last substitution that atom has been changed to `edge(b,Node2)`. By a similar process we see that if we take the substitution

```
Node2 = d
```

and apply it to the query, then the query is satisfied.

The whole query has now been resolved. If we look at our original query, we want to know what value for X will satisfy `tedge(a,X)`? By following the series of substitutions, `X = Node2, Node2 = d`, we find that the substitution `X = d` satisfies the original query.

Unification/Resolution

We can identify two distinct activities in this process. *Unification* refers to the process of taking two atoms (one from the query and the other being a fact or the head of a rule) and determining if there is a substitution which makes them the same. We will look at the underlying algorithm for unification later. The second activity is *resolution*. When an atom from the query has been unified with the head of a rule (or a fact), resolution replaces the atom with the body of the rule (or nothing, if a fact) and then applies the substitution to the new query. We can make unification and resolution more explicit by processing the original query again.

1. Unify

    `tedge(a,X)` and `tedge(Node1,Node2)`,

    giving the substitution

    `Node1 = a, X = Node2`.

    Resolution: replace `tedge(a,X)` with `edge(Node1,SomeNode),edge(SomeNode,Node2)` and apply the substitution above to get the new query:

    `edge(a,SomeNode),edge(SomeNode,Node2)`

2. Select the atom `edge(a,SomeNode)`.
    Unify

    `edge(a,SomeNode)` with `edge(a,b)`,

    giving the substitution

    `SomeNode = b`.

Resolution: replace `edge(a,SomeNode)` by nothing (since we unified with a fact) and apply the substitution above to get the new query:

```
edge(b,Node2)
```

3. There is only one atom in the query.

   Unify

   `edge(b,Node2)` and `edge(b,d)`,

   giving the substitution

   ```
   Node2 = d.
   ```

   Resolution: replace `edge(b,Node2)` by nothing (since we unified with a fact). Since the resulting query is empty, we are done.

At this point the substitution can be determined. At this point it is a good idea to pause to observe what has happened with the parameters of the atoms involved in the computation. In our original query we had one constant and one variable. If you consider carefully, you may see that the constant parameter has been passed to each successive atom as a value, while the variable `X` has been passed a reference. Notice that at no point has there been anything like a traditional computation – just the unification to match up arguments (pass parameters?) and the resolution to reform the query.

Backtracking

We are actually not finished with the example. You may have noticed that there is another solution: we could redo the computation above and get the substitution `X = b` or `X = c` or `X = d`. The way *Prolog* does this is, when a query is finally reduced to the empty query, *Prolog* backtracks to the most recent unification to determine whether there is another fact or rule with which unification can succeed. If there is, an additional solution may be found. Backtracking continues until all possible answers are determined.

<u>**Problem 3**</u>

1. Consider the problem of backtracking in response to the query '`?- tedge(a,X)`'. Assume the graph of Example 3.3. Write out the steps of the unification/resolution above and clearly mark any place where more than one rule or fact must be checked for unification (remember that each one *may* lead to another solution). Then continue the unification/resolution process via backtracking to determine all possible answers to the query.

2. (Extra Credit!) **Your family tree** — Probably the most common first example of a logic program is that involving a family tree. Create a set of facts that define for your immediate family the relations:

   ```
   male      = { The males in your family. }
   female    = { The females in your family. }
   parent_of = { The pairs (x,y) where x is the parent of y
                 in your family. }
   ```

   [Don't go overboard if you have a large family — you don't want to be typing data till next week!]

   Note that the predicates `male` and `female` are *unary*, involving only one argument. You need one entry for each person in your family. Next define the following predicates based on the previous three predicates.

   ```
   father, mother, son, daughter, grandfather,
   aunt, uncle, cousin, ancestor
   ```

Notice that the first of these predicates must be defined in terms of facts, but that the later ones can be defined in terms of the earlier predicates.

So if I did this problem I might start with the following and work from there (in the obvious ways).

```
male(jerry).
male(stuart).
male(warren).
female(kathe).
female(maryalice).
brother(jerry,stuart).
brother(jerry,kathe).
sister(kather,jerry).
parent_of(warren,jerry).
parent_of(maryalice,jerry).
```

∎

4 Lists in *Prolog*

Lists in *Prolog* are represented by square brackets. Similar to *Hugs*, *Prolog* also allows for aggregate values of type lists. For example, when calling a predicate, we may pass a list containing elements a,b,c by typing

```
[a,b,c].
```

The empty list is represented by `[]`. To define rules for manipulating lists, we need list constructors, as well as selectors for pulling apart lists. We saw these operations in *Hugs* in the form of the '`:`' operator and the functions `head` and `tail`. Prolog combines all of these operations into one, with the help of unification (in a similar way to *Hugs* using pattern matching). To illustrate, we give below the Prolog program for appending lists. It contains two rules.

```
append([],List,List).
append([H|Tail],X,[H|NewTail]) :-  append(Tail,X,NewTail).
```

[Remember that we read these rules "left-hand-side is true **if** the right-hand-side is true."] A call to `append` requires three arguments. The natural way to view the intended use of the arguments is that the first two arguments represent the two lists that we want to append, and the third argument returns the result of the append. For example, if we pose the following query to append,

```
?- append([a,b,c],[d,e],X).
```

we would get back the answer `X = [a,b,c,d,e]`. To see how append actually works, an explanation of the vertical bar symbol `|` is needed. It acts both as the constructor and selector for lists.

When we write `[X|Y]`, this represents a list whose head is `X`, and whose tail (i.e. the rest of the list) is `Y`. Based on this interpretation, unification works exactly as you would expect:

```
[X|Y] unifies with [a,b,c] with the unifier {X = a, Y = [b,c]}.
[X|Y] unifies with [a,b,c,d] with the unifier {X = a, Y = [b,c,d]}.
[X|Y] unifies with [a] with the unifier {X = a, Y = []}.
[X|Y] does not unify with [].
```

Now recall from the previous section that a variable occurring in the head of a rule is treated *universally*, while a variable that occurs only in the body is handled *existentially*. Thus the first rule in the append definition

```
append([],List,List).
```

reads intuitively as: "The result of appending the empty list `[]` to any list `List` is just `List`." Note this is the first example of a "bodiless" rule that we have seen. A call to this rule will succeed as long as the first argument unifies with `[]`, and the last two arguments unify with one another.

```
Query                               Unifier          Prolog answer
-----------------------------------------------------------------
?- append([],[b,c,d],[b,c,d]).      {List = [b,c,d]}      yes
?- append([],[b,c,d],X).            {List = [b,c,d],
                                     X = [b,c,d]}       X = [b,c,d]
?- append(X,Y,Z).                   {X = [],
                                     Y = List,
                                     Z = List}          X = [], Y = Z
?- append([],[b,c],[b,c,d]).         None                 no
```

The second call in the above table is normally the way we would use append, with the first two arguments instantiated, and the third a variable that will provide an answer.

Now consider the second rule, which is the general case of appending a non-empty list to another list.

```
append([H|Tail],List,[H|NewTail]) :-  append(Tail,List,NewTail).
```

All the variables that occur in this rule are universal variables since each occurs in the head of the rule. If we call append with the query:

```
?- append([a,b,c],[d,e],Result).
```

The atom in the query does not unify with the first rule of append since the list `[a,b,c]` and `[]` do not match. The query however, unifies with the head of the second rule of append. According to unification, the resulting unifier is:

```
{ H = a, Tail = [b,c], List = [d,e], Result = [a|NewTail] }
```

This substitution is applied to the body of the rule, generating the new query:

```
?- append([b,c],[d,e],NewTail).
```

Thus by unifying the first argument of the query with the first parameter `[H|Tail]` in the head of the rule, we "pull apart" the list into its head and tail. The tail of the list then becomes the first argument in the recursive call. This is an example of using `|` in a way similar to `head` and `tail` in *Hugs*. Let's examine what happens if we carry the trace all the way through. First, at the *Prolog* prompt you should enter '`trace.`', to start the trace facility. After starting tracing every query you enter (including the entries you think of as commands!) will be traced. Just enter `<return>` when the interpreter hesitates. Enter the specified query (line 1 below) and enter return. The following is an annotated version of (approximately) what should appear on the screen.

```
1 Call    append([a,b,c],[d,e],Result).      %% Initial call
                                             %% Result = [a|NewTail]
2 Call    append([b,c],[d,e],NewTail).       %% First recursive call
                                             %% NewTail = [b|NewTail1]
3 Call    append([c],[d,e],NewTail1).        %% Second recursive call
                                             %% NewTail1 = [c|NewTail2]
4 Call    append([],[d,e],NewTail2).         %% Third recursive call
5 exit    NewTail2 = [d,e]                   %% Unifying with base case
                                             %% of append.
6 exit    NewTail1 = [c,d,e]                 %% Back substitution of
                                             %% NewTail2
7 exit    NewTail = [b,c,d,e]                %% Back substitution of
                                             %% NewTail1
8 exit    Result = [a,b,c,d,e]               %% Back substitution of
                                             %% NewTail
```

In the comments appearing on the right, we indicate the important substitutions to remember. In the first step, as the result of unifying the original query with the head of the second rule of append, the variable `Result` is unified with `[a|NewTail]` where `NewTail` is a variable that has yet to have a value (i.e. uninstantiated). In the second step where the first recursive call takes place, `NewTail` unifies with `[b|NewTail1]`. Remember that each time a rule is used, the variables are considered to be new instances, unrelated to the variables in the previous call. This is the reason for the suffix 1 in the variable `NewTail1`, indicating that this is a different variable than `NewTail`.

When computation reaches step 4, the first argument becomes the empty list `[]`. This unifies with the first rule of append, and in the process the variable `NewTail2` is unified with the list `[d,e]`. At this point, computation terminates and we "unwind" from the recursion. However, in the process of unwinding, we back substitute any variable that is instantiated in the recursion. That is how the final result of the append is constructed. Specifically, in unwinding from the recursive call in step 3, we back substitute the value of the variable `NewTail2` inside `[c|NewTail2]`. The result is the list `[c,d,e]`. This list becomes the value stored in the variable `NewTail1`. Next to unwind from the recursive call in step 2, we back substitute the value of `NewTail1` into `[b|NewTail1]`. Since from the previous step we just calculated `NewTail1` to be the list `[c,d,e]`, the current back substitution yields the list `[b,c,d,e]`, which becomes the value stored for the variable `NewTail`. Finally, in the last unwind, the value of `NewTail` is back substituted inside `[a|NewTail]`, producing the list [a,b,c,d,e]. This is now the final answer for the variable Result. The back substitution process just described illustrates how | acts as a list constructor that puts together an element with a list.

Example 4 – **Generating a Chain**

If we return to Example 3.3, the finite state machine, we can put the list idea to work. We want to define a predicate chain which, when given two nodes will list the nodes in a path from the first node to the second. This list of nodes is referred to as a *chain*. First, by the nature of *Prolog*, what we want to do is define a predicate with three parameters

```
chain(X,Y,Z)
```

where the first will match the initial node, the second will match the final node, and the third will match a list consisting of the nodes in a chain. We can think of the first two parameters as input parameters and the third as an output parameter (though it need not be used in this way).

The way to arrive at a definition of chain is to think of the conditions under which a chain will exist (taking advantage of recursion when necessary). Since a chain is just a list of nodes in a path, we should be able to make use of the definition of path, taking each element in a path and tacking it onto a list. The first thing that occurs is that if there is an edge from X to Y then there is a chain [X,Y]. This is easy to state in *Prolog*.

```
chain(X,Y,[X,Y]) :- edge(X,Y).
```

Remember, that we read this definition as "If there is an edge from X to Y then [X,Y] is a chain from X to Y." Of course, it may be the case that there is a less direct route to Y. But if so, there is always a first step from X to, say, I and then a path from I to Y. Here's the recursion. If there is a path from I to Y, then there is a chain for the path and we can tack X onto that path. Here is a representation of this in *Prolog*.

```
chain(X,Y,[X|Z]) :- edge(X,I), path(I,Y), chain(I,Y,Z).
```

If no path can be found to Y then we should have the empty list as the chain. Here is the complete definition of chain. Why is the last line in the form of a fact rather than a rule?

```
chain(X,Y,[X,Y]) :- edge(X,Y).
chain(X,Y,[X|Z]) :- edge(X,I), path(I,Y), chain(I,Y,Z).
chain(X,Y,[]).
```

There is a subtle point about this example. If you now try to generate chains from, say, a to d, you will find that no solution is generated - in fact the system is in an infinite loop trying to find a path to d. If you recall, *Prolog* always looks to unify a predicate by starting with its first definition and then pro ceding in order until one is found which unifies. What this means is, if you put the edge definitions which introduce loops into the graph at the end of the definition list for edge, then these definitions will be used last. To see the affect of this try some queries before changing the order and then try the same queries after changing the order.

∎

Example 5 – **Reversing a List**

As another example, the following rules define a Prolog program for reversing lists. It works by recursion and makes use of append. A call to reverse requires two arguments, the first is the input list, and the second is the result of reversing.

```
reverse([],[]).             %%% Reversing an empty list is the empty
                            %%% list.
reverse([H|Tail],Result) :-          %%% To reverse a non-empty list,
      reverse(Tail,Tailreversed),    %%% first reverse the tail,
      append(Tailreversed,[H],Result). %%% append the result to the
                                     %%% list [H].
```

∎

Problem 4

1. Write a *Prolog* definition for a predicate `mem` which tests for membership in a list. [How many parameters should it have? How can recursion be used? There is already a predefined function called `member` which behaves the same way.]

∎

Before moving on, here is another application of lists. The use of finite state machines is very common in many sorts of computer applications. What we will look at here is how to simulate a finite state machine, regardless of the language it is to recognize.

The simulation actually involves two components. First there must be a mechanism for describing the machine, i.e., its states and transitions. Second, we must define a predicate which simulates the action of the specified machine. Specifying a machine is a matter of defining three components: the state transitions, the start state, the final states. Defining the final states and the start state is easy (as will be illustrated). To define the transition we must indicate for each state and input character what is the next state. Here is an example description.

```
start(s).

final(f1).
final(f2).

delta(s,a,q1).
delta(s,b,q2).
delta(q1,b,s).
delta(q1,a,f1).
delta(q2,a,s).
delta(q2,b,f2).
```

Can you draw a diagram representing this finite state machine?

Now for the simulator. This simulator, by the way, should work for any finite state machine defined in the way just described. What we want is to know what input strings this machine will accept (i.e., process the characters one-at-a-time starting in the start state and ending in **a** final state with no input characters remaining). The predicate `accept` is our goal. We want to give a list of characters to the predicate and have it tell us whether the string is accepted by the machine. The following definition is the easy solution. Try it out. Are the actions of `accept` affected by the order of the components of `delta`? Rearrange them to see.

```
accept(S) :- start(S), accept(Q,S).     %%% This looks strange!!!?
accept(Q,[X|XS]) :- delta(Q,X,Q1),      %%% Does this make sense that
                    accept(Q1,XS).      %%% the number of parameters to
accept(F,[]) :- final(F).               %%% accept changes?
```

This definition is pretty strange. But if we consider how *Prolog* works, by unifying a query with the head of a rule (or a fact), we see that if we enter the query

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

that it can only match the first rule. The reference to `accept` in the *body* of that clause, however, will unify only with one of the second two rules. Interesting! This is a kind of overloading, it would appear.

Problem 5

1. An interesting problem to consider is how one might define a *Prolog* program to check whether a given list is a palindrome. To be more exact, we would like to define `palindrome` so that it behaves in the following way.

```
Query:                     Result:
----------------------------------------------
palindrome([a,b,c])         no
palindrome([])              yes
palindrome([a,b,a])         yes
palindrome([a,b,b,a])       yes
```

   There are various ways to implement this predicate, with one solution being exceptionally simple and intuitive.

   Hint: Think of a finite state machine which has two states – `q0` and `q1`.

   - Define two `palindrome` predicates – one taking 1 parameter and the other taking three parameters

     `palindrome(q,X,S)`

     where the first is a state, and the other two are lists. The first is defined in terms of the second as follows.

     {palindrome(Xs) :- palindrome(q0,Xs,[]).
   - In state `q0` remove things from the list `X` and add them to `S`.
   - In state `q1` succeed when the heads of the two lists are the same – i.e., when `palindrome(q1,[X|Xs],[X|S])` is true.

   Problem – how does the transition from `q0` to `q1` occur?

∎

5 Data Input in *Prolog*

As in any language, one of the most complex and detail-oriented aspects is the input and output of data. IO processing in *Prolog* is no less complex than what we saw in *Hugs*. We will look at just enough of the input facilities to allow us to read characters (unfiltered) from a file. If you are interested you can check out the other IO facilities in the *SWI-Prolog* manual.

5.1 Input

In order to see how file input is accomplished in the context of logic programming, lets start with the high level and see if we can figure out what is needed. First, we will find it convenient to access the characters in a file from a *Prolog* list. This means that we need a predicate which will take a file name (as an input parameter) and then unify a second parameter with the list of characters from the file.

```
file(X,Y) :- .....  %% X is the file name
                    %% Y is the list of file characters
```

If we have a file called `'test.data'` we would expect the query

```
?- file('test.data',D).
```

to cause `D` to be unified with the list of characters from the file and the characters in `D` to be printed.

Of course, before we can pose the query we must complete the definition for the predicate `file`. How should we do this? If we just think intuitively, `file(X,Y)` should be true if we can open the file `X` and `Y` results from reading the data in `X`. Well, let's write that out:

```
file(X,Y) :- open(X,In), %% bind the internal name 'In' to X
             readAll(In, Y).
```

That looks OK — `open` will have to be some built-in predicate which will open the file named by `X` and unify `In` with an internal file descriptor. The idea is that having "opened" the file we then use the file descriptor when actually reading the data.

What about `readAll`? We want this predicate to unify `Y` with the list of characters from the file referenced by `In`. The definition of `open` will follow momentarily, but first a quick look at `readAll`. The idea is to take characters from one list (i.e., the file) and put them on another list (to unify with `Y` above) only if the characters are OK. This is a recursive activity and can be seen to fall generally into the following pattern.

```
readAll([C|Rest],[C|Answer]) :-  %% if C is OK
    ...., % conditions go here
    readAll(Rest,Answer).

readAll([C|Rest],Answer) :-      %% if C is not OK
    ...., % conditions go here
    readAll(Rest,Answer).
```

But in this problem the first argument to `readAll` is the file descriptor – that means it isn't really a list and we can't use this pattern-matching. Instead, we use another special "built-in" predicate `'get0'` which (internally) matches the next character and moves a file pointer along to the next character (we never see that in our code). This reading of a character will have to be represented as part of the "condition".

Before tackling the definition of `readAll`, here are definitions you will need from the standard predicate collection which accompanies *SWI-Prolog* – these are automatically loaded when the system is started. [See the online *SWI-Prolog* manual – Section 3.13.2.]

open/4: The predicate `open` takes four parameters (or at least that's how we will use it!) and can be called as follows:

```
open(X, read, In,[eof_action(eof_code)]
```

where `X` is the file name, `read` is a constant which indicates how the file will be used, `In` is the internal designator for the file, and `eof_action(eof_code)` is a list of options (it could be longer!) which indicates that when the end-of-file is reached a value of -1 should be returned.

get0/2: (That's a zero, not an Oh!) This predicate takes two parameters:

```
get0(In, C)
```

where `In` is the internal file designator and `C` unifies with the next character in the file...next is important since there is some internal mechanism to keep things moving along the input stream. Each time `get0` unifies the hidden file pointer is moved along.

`get0(In,C)` doesn't actually cause `C` to unify with the next character in the file, but the ASCII code of the next character. This means that our list will actually contain ASCII codes (i.e., integers) rather than characters (i.e., they will display as integer values rather than as characters).

close/1: The `close` predicate takes a file designator and unifies by closing the referenced file.

<u>**Problem 6**</u>

In this problem you will implement the predicate definition for `readAll` by making use of the built-in predicates just defined.

Above we discussed the structure of the predicate `file` and indicated that it would make use of the `open` built-in predicate. That structure still works, but now you use the `open/4` predicate just described.

To get you started, you might try the following mini-problem. Rather than trying to get at all the characters in the file, just write `readAll` so that it gets the first character of the file. The following definitions should work.

```
file(X,Y) :-
        open(X,read,In,[eof_action(eof_code)]),
        readAll(In, Y).

readAll(In,C) :- get0(In,C).
```

There are things this doesn't do, for example, check the validity of `C` or close the file. But if you put it into a file (name it `'filein.pl'`) you can try a query such as

```
file('filein.pl',C).
```

When this works, remember, you will get an ASCII code printed rather than a character.

Before starting on the general problem there are two things you should consider. First, we want to make sure that the only character codes which end up in our list are those with legal (i.e., printable) values. A reasonable way to handle this is to introduce a predicate `legal/1` and facts and rules which define which characters are "legal" (ascii value `>= 32` and ascii value = 10, for example) . Second, when the end-of-file is reached remember that `get0(C)` will unify with the value -1. In this case no character will be put on the list. This should require two different clauses. (Be sure to remember to close the file when you reach the end-of-file!).

Finally, rememeber that the way to think when defining the rules for a predicate is "the left side will be true is the right side is true" – i.e., what condition (on the right) will lead to the truth of the condition on the left?

Now you can set about implementing the predicate `readAll`. When done you should be able to test by entering a query based on the `file` predicate, as indicated at the beginning of the section.

[Hint: define two predicates – `legal/1` as described above, and `eof_char/1` which is true for just one value - `'-1'`.] ∎

Remember that this file input program (the predicates `file`, `readAll`, `eof_char`, `legal`) should be in a prolog file called `filein.pl`. Now when you want to use these predicates (most likely just `file`) you can include the following line as the first line of a program file.

```
:- consult(filein).
```

This line is treated as a query which must be resolved before the rest of the program is read in.

5.2 Building a Tokenizer

One useful application of the `file` predicate you just developed is supplying characters from a file to a tokenizer. Remember that a tokenizer recognizes the lexical elements of a language and that these lexical elements can be described by a regular expression or regular grammar. In this section you will write a tokenizer for the FP language.

We have two goals in mind. First, we need to implement a finite state machine which will accept the tokens of the FP language. This will be implemented (similar to the FP project tokenizer) by a predicate which will unify with the "next" token in the input stream – call it `getToken`. Second, we need a higher level predicate which will repeatedly (i.e. recursively) reference the finite state machine (i.e., `getToken`) to unify with the list of tokens – we will call this predicate `getTokens`.

We will attack the tokenizer in two parts. First we will look at the case where the tokens have a fixed length, in this case having either one or two characters. Then we will look at how to get at the identifier and numeric literal tokens.

One and Two-character Tokens

Suppose we want to recognize a `'+'` sign and return the token `plus`. First, since we are just handling one character it is not necessary to follow a transition in the state machine, just to determine and return the token and the part of the input list which remains.

```
getToken(start,[43|Rest],Rest,plus).
```

Remember that the input list consists of ASCII values, so the 43 above must be the ASCII code for `'+'`. [*Prolog* doesn't see the ASCII integer code and the printable character as the same.] Notice that this clause will only unify for the start state and the ASCII code 43 at the head of the input list. If some other ASCII value is at the head of the list then another clause will have to unify.

Another thing about the fact given above for `getToken`. If we look at the four parameters, the first two really behave as "input" parameters and the second as "output" parameters. Now, there is actually no input or output activity as we are used to in C^{++}, but that is intuitively what is happening. So when a query with `getToken` is done, the arguments associated with the second two parameters will supply values which feed into the next predicate in the query.

[N.B.: You can find the list of ascii codes (in decimal, octal, and hex!) by executing the UNIX command `'man ascii'`.]

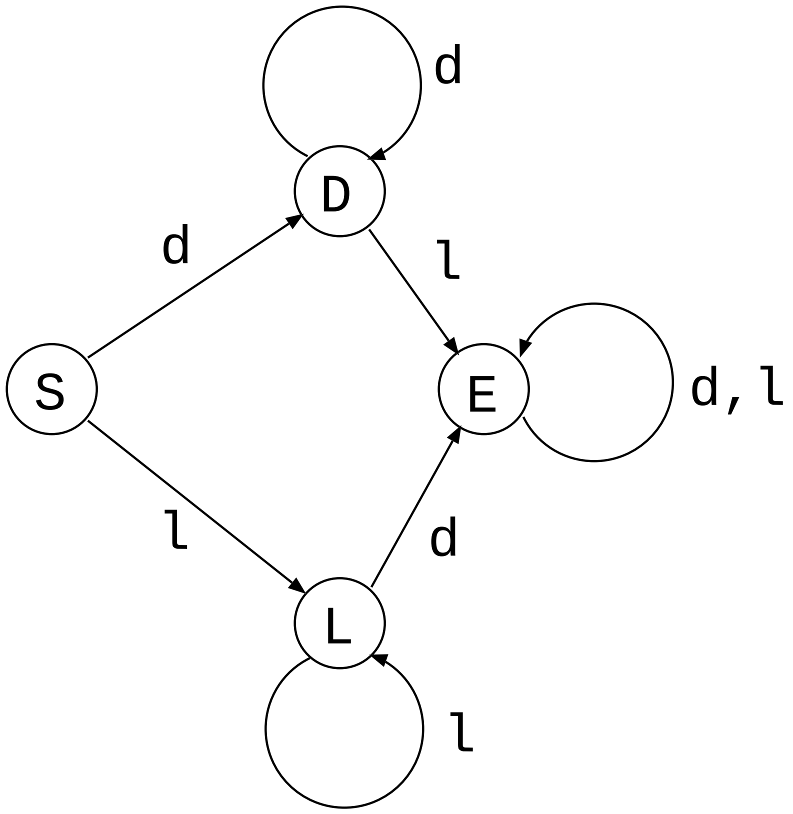
A fixed multi-character token (e.g. '==') can be handled as follows:

```
getToken(start,[63,63|Rest],Rest,plus).
```

Revisiting the last two parameters. The final one, `plus`, is just the enumerated value for the token. The third is there so that we can use recursion on the rest of the list. In other words, when a `getToken` query is handled, the value bound to `Rest` will be the part of the input list which hasn't been consumed yet. That will go in for the second parameter in a subsequent (i.e., recursive) reference to `getToken`.

The Number Token

Let's assume the following finite state machine for the FP tokens. Notice that since the fixed tokens are handled without transitions, only those transitions related to number, identifier, and error tokens are represented.



To recognize numeric (or identifier or error) tokens takes a bit more work than required for the fixed tokens. There are a couple of (related) points to consider. First, the number of characters involved in a numeric token is unknown. Second, we must retain the characters of the token as the value associated with the token returned. This means two things. We must accumulate the characters of the token as they are recognized. We must define an encapsulation "function" for the number token.

This segment of code defines the number-token aspect of the `getToken` predicate.

```
getToken(digits,[C|R],Rest,S,T) :-
    digit(C),
    getToken(digits,R,Rest,[C|S],T).

getToken(digits,[C|R],Rest,S,T) :-
    letter(C),
    getToken(error,R,Rest,[C|S],T).

getToken(digits,R,R,S,number(N)) :-
    reverse(S,S1),
    atom_chars(N,S1).
```

There are a few things which you should notice.

1. The number of parameters for `getToken` has expanded to five! This is because we need another to unify with the characters in the token.
2. When digits are encountered they are passed *down* to the next level, not back up as we have seen before.
3. When a letter is seen we move from the `digits` state to the `error` state. Any other character terminates the number token.

4. You should try this definition for `getToken` on a file containing only number tokens (and any other tokens for which you have defined `getToken`). But use the following clause

   ```
   getToken(digits,R,R,S,number(S)).
   ```

   in place of the third clause above. You will notice that the result is displayed in terms of ASCII codes. The predicate `atom_chars/2` converts a list of ASCII codes to a character format. If you add that reference to `atom_chars/2` back in, then you will notice that the characters in the token are backwards! The reverse takes care of that.

Problem 7

Now that you have some examples to go on, implement the *Prolog* version of the FP tokenizer, `getToken` and test it on individual tokens.

∎

Now that you have an implementation for `getToken` it is necessary to implement a higher level predicate which uses `getToken` to help generate a return list consisting of the tokens from the file.

Problem 8

Remember that `getToken` is a predicate which takes three arguments as follows.

```
getToken(InList,Token,Rest)
   %% InList -- the list of input characters
   %% Token  -- the token at the beginning of InList
   %% Rest   -- the part of InList which follows Token
```

The idea is to implement a predicate, say `getTokens`, which takes two parameters, the input list and a second parameter which will unify with the list of tokens found in the input list.

Implement `getTokens`, remembering that though the top-level definition of the predicate may take only two parameters, a lower-level form of the same predicate could take three. [This hint is important. If you think about it, recursive references to `getTokens` will actually need three parameters, with the extra one representing the part of the input list remaining after the token.]

You will have to wait till later to test this predicate.

The following rules will be useful.

```
whiteSpace(32).
whiteSpace(10).

digit(D) :- 46 < D, D < 59.

letter(L) :-  64 < L, > < 91.
letter(L) :- 96 < L, L < 123.
```

∎

6 Parsing

In this section you will investigate how *Prolog* is used to parse an input stream. You will make use of the tokenizer written for the last section to write a parser for the grammar of FP given in the *Hugs* tutorial. *Prolog* is particularly well suited to parsing since grammar rules have an uncanny resemblance to *Prolog* clauses.

Remember that when parsing for a particular grammar rule there are really two kinds of activities:

- checking for a particular token and
- parsing another complex grammatical structure.

We will deal with each of these in turn.

6.1 Low-level Parsing with *Prolog*

By low-level parsing we mean the recognition of individual tokens. The idea, of course, is that we have an input stream consisting of tokens and to continue a parsing activity we must check to make sure the next token in the stream is the one we expect to see (according to some grammar rule). This seems pretty straight forward.

We need a predicate which will succeed exactly when the input stream is headed by a specified token. Remember that when we use the tokenizer from the previous section the input stream will appear as a list of tokens.

Problem 9

1. Define a *Prolog* predicate `match/2` using a single rule or fact (a fact would be nice!) which fits the description just given. Assume the first parameter unifies with a token and the second with a token list. The predicate would be used in one of these ways:

   `match(if, Tin).` or
   `match(ident(a), Tin).`

   [Here `a` is the identifier's name gleaned from the input stream.]

2. The `match/2` predicate is good for checking what the next token is. But there are times when you also want to check the token and then eliminate it from the stream. Define a new predicate `eat/3` which does this. You should easily be able to adapt the definition for `match/2` to get a definition for `eat/3`.

∎

6.2 High-level Parsing with *Prolog*

Consider the following rule from the grammar for the language FP.

```
<FnDef> ::= Ident <ParamList> Assign <FnExpDef>
```

One way to read this rule is as follows.

> If the input string contains an `Ident` token *and* a `<ParamList>` *and* an `Assign` token *and* a `<FnExpDef>` *then* it contains a `<FnDef>`.

The use of '*and*' emphasizes the conjunctive nature of the right-hand-side of the grammar rule. It seems that we could immediately translate this into *Prolog*.

```
parseFnDef(..) :-
     parseIdent(..),
     parseParamList(..),
     parseAssign(..),
     parseFnExpDef(..).
```

In fact, every grammar rule can be converted to a *Prolog* clause in this way (more or less!).

Example 6 – **Parsing `if..then..else`**

When parsing more than just a single token we cannot count on a *Prolog* fact to do the job. But the principle should be the same. If we want to recognize an `if..then..else` statement we need to match the initial part of an input list and pass on the part of the list which was not part of the match. We can consider the following Pascal-style statements and the corresponding list of tokens.

```
if a then b else c;
x := 5;

[if,ident(a),then,ident(b),else,ident(c),;,ident(x),=,number('5'),;]
```

[Notice that some tokenizer converted the `:=` to a simple `=`.]

Now what we want is some predicate `parseIfThenElse`, which will match the first 6 tokens and 'return' the rest of the list as unmatched. How should this predicate be defined? Well, a fact will not do it, so we must define a *Prolog* rule. The left-hand side could be something like

```
parseIfThenElse(Sin,Sout) :- ...
```

where `Sin` unifies with the original list and `Sout` unifies with the list remaining after matching the `if..then..else` structure. We might issue the query with the specified results.

```
| ?- parseIfThenElse([if,ident(a),then,ident(b),else,ident(c),scolon,
     ident(x),assign,number('5'),scolon],X).

X = [;,x,=,5,;] ?

yes
```

How about the right-hand side for the rule for `parseIfThenElse`? That part of the rule must indicate that the token `if` is matched and the rest of the list, lets call it `X`, is returned; then the token `a` will be matched from the head of `X`, leaving a list `Y`; and so on. In other words we can form the rule as follows.

```
parseIfThenElse(Sin,Sout) :-
              parseIf(Sin,R),
              parseName(R,S),
              parseThen(S,T),
              parseName(T,U),
              parseElse(U,V),
              parseName(V,W),
              parseEndIf(W,Sout).
```

Notice that when the last predicate on the right is resolved the variable `Sout` should have the part of the list not recognized.

One final point on this example. You may be wondering where the predicates `match/2` and `eat/3` are supposed to come in. In fact, the above rule could be rewritten making use of `eat/3` as follows, since each predicate is matching a single token.

```
parseIfThenElse(Sin,Sout) :-
              eat(if,Sin,R),
              eat(ident(_),R,S),
              eat(then,S,T),
              eat(ident(_),T,U),
              eat(else,U,V),
              eat(ident(_),V,W),
              eat(endif,W,Sout).
```

Of course, neither of these rules will be of much use in the general case since the `else` and `then` parts are likely to be more complex.

∎

Problem 10

1. Implement a single parse predicate, as illustrated above, for the following grammar rule.

   ```
   parseFnDef(Tin,Tout) :-
        IDENT <ParamList> ASSIGN <FnExpDef>
   ```

   Be sure to make appropriate use of the low-level parsing predicates defined earlier.

2. Implement a single general rule for `parseIfThenElse`.

∎

Problem 11

A fun but somewhat unrelated problem.

Write a predicate `separate` which takes a list of tokens and returns two token lists, one containing all the number tokens of the input list and the other containing all the identifier tokens from the input list. Any other tokens from the list are discarded. If the file contains the line

```
‘12 + 13 equals twenty five (25)’
```

then we should see the following result.

```
?- tokens('test',Y), separate(Y,Ids,Nums).

Ids = [id(equals),id(twenty),id(five)]
Nums = [num('12'),num('13'),num('25')]
Y = [num('12'),plus,num('13'),id(equals),id(twenty),id(five),
     lp,num('25'),rp,eof]

Yes
```

Your names for tokens may be slightly different, but should be recognizable. ∎

6.3 Parsing Expressions

A common problem in parsing is dealing with alternatives. We can see this problem and also deal with a more complete grammar by implementing a parser for basic arithmetic expressions. Recall the following grammar rules for expressions (a good review!).

```
Exp    ::= Term { '+' Term } | Term { '-' Term }
Term   ::= Factor { '*' Factor } | Factor { '/' Factor }
Factor ::= number | ident | '(' Exp ')'
```

This grammar can be written in a bit more convenient way as follows.

```
Exp      ::= Term ExpTail
ExpTail  ::= empty | '-' Term | '+' Term
Term     ::= Factor TermTail
TermTail ::= empty | '*' Factor | '/' Factor
Factor   ::= number | ident | '(' Exp ')'
```

The grammar rules can be implemented in the same manner as in the previous section, except that we have alternatives to deal with; each non-terminal has at least two alternative right-hand sides. We recall that the alternative operator '|' is just a shorthand notation for multiple grammar rules with the same left-hand side. The appropriate parse predicates can at this point be defined by having one clause for each grammar rule, where we expand each alternative rule into multiple rules. I.e., the following list of clauses defines a parser for the expression grammar above.

```
exp(Tin, Tout)       :- Term(Tin,X), expTail(X,Tout).

expTail(Tin, Tin)   :- .
expTail(Tin, Tout)   :- eat(plus,Tin,X), term(X,Tout).
expTail(Tin, Tout)   :- eat(minus,Tin,X), term(X,Tout).

term(Tin, Tout)      :- factor(Tin,X), termTail(X,Tout).

termTail(Tin, Tin)  :- .
termTail(Tin, Tout)  :- eat(mult,Tin,X), factorX,Tout).
termTail(Tin, Tout)  :- eat(div,Tin,X), factorX,Tout).

factor(Tin, Tout)    :- eat(number(_),Tin,Tout).
factor(Tin, Tout)    :- eat(ident(_),Tin,Tout).
factor(Tin, Tout)    :- eat(lp,Tin,X),exp(X,Y),eat(rp,Y,Tout).
```

Problem 12
Implement the parser just described and run it on a file containing an expression — do this a few times to convince yourself that the parser works. Notice that in this problem you have used the FP tokenizer to implement a parser for a "language" unrelated to FP. ∎

6.4 Parsing Optional Structures

One of the features of a selection statement is that the `else` part is optional. How do we deal with optional parts in *Prolog*-based parsing? In fact, we can pretty much just write down the form of the grammar rules as is. For example, if we write the selection grammar rule as follows:

```
<IfElse> ::= IF <Bexp> THEN <FnExp> <ElsePart> ENDIF
```

Then the corresponding *Prolog* parse predicates can be written as follows.

```
parseIfElse(Sin,Sout) :-
    eat(if,Sin,R),
    parseBexp(R,S),
    eat(then,S,T),
    parseFnExp(T,U),
    parseElsePart(U,V),
    eat(endif,V,Sout).

parseElsePart(Sin,Sout) :-
    eat(else,Sin,R),
    parseFnExp(R,Sout).
parseElsePart(Sin,Sin). %% makes it optional
```

Another form of optional parsing occurs when a particular token can start more than one phrase. This occurs in a function expression where an identifier may be alone, signifying the use of a parameter, or followed by a sequence of terms, in which case the identifier is a function name. Once again we can just take the grammar rules describing this ambiguous situation and translate them directly to *Prolog*. The following rule is an abbreviated version of the FP function expression grammar rule.

```
<FnExp> ::= NUMBER |
            IDENT  |
            IDENT  <TermList>

parseFnExp(Sin,Sout) :-
    eat(number,Sin,Sout).
parseFnExp(Sin,Sout) :-
    eat(ident(_),Sin,Sout).
parseFnExp(Sin,Sout) :-
    eat(ident(_),Sin,R),
    parseTermList(R,Sout).
```

Problem 13

Now, using the techniques described above, implement a set of parse predicates for the FP grammar which follows. Be sure to test it out when you are done.

```
<Program>   ::= <FnDefList> PERIOD

<FnDefList> ::= <FnDef> |
                <FnDef> SCOLON <FnDefList>

<FnDef>     ::= IDENT <ParamList> ASSIGN <FnExpDef>

<FnExpDef>  ::= <FnExp> |
                let <FnDefList> in <FnExp>

<ParamList> ::= IDENT |
                IDENT <ParamList>

<FnExp>     ::= NUMBER  |
                IDENT   |
                <FnApp> |
                <Selection>

<Selection> ::= if <BExp> then <FnExp> [<ElsePart>] endif

<ElsePart>  ::= else <FnExp>

<BExp>      ::= <FnExp> <CompOp> <FnExp>

<CompOp>    ::= LT | LE | GT | GE | EQ | NE

<FnApp>     ::= <FnName> <TermList>

<FnName>    ::= IDENT | ADD | SUBT | MULT | DIV | MOD

<TermList>  ::= <Term> |
                <Term> <TermList>

<Term>      ::= NUMBER |
                IDENT  |
                LP <FnExp> RP
```

∎

7 Building a Symbol Table

The parser described in the last section allows us to check a program for syntactic correctness, but doesn't do anything about insuring proper use of program elements. For example, parsing the function definition

```
f a b = + x b;
```

will succeed, but, in fact, should fail because the identifier `x` is not a formal parameter of the function being defined. In order to be to deal with this type of error we must build a symbol table as we parse and judge the fitness of the code based on its syntax and the content of the symbol table.

In the development of the parser we relied on representing parsing rules in terms of two lists: the input list and the list comprising the data not parsed by the rule. A symbol table has a similar functionality in the parsing process, except that, rather than being torn down as the parsing proceeds, the symbol table is constructed as the parsing proceeds. Where have we seen the building of a list before? How about the predicate `separate`? The idea there was to take a list `A` and produce two lists, one containing the integers from `A` and the other containing the identifiers from `A`. The technique used in implementing `separate` is to have an input list and (in this case) two result lists. The `separate` predicate was the subject of Problem 6.2 – look up your solution and notice the technique used to build the result lists.

So, to add a symbol table to our parsing function we can do (in reverse) what we did with the program list in the original parser: for each predicate have an input symbol table and a resulting output symbol table. For example, `parseFnDefList` can be defined as

```
parseFnDefList(Sin,STin,STout,Sout) :-
      parseFnDef(Sin,STin,STout,Sout).

parseFnDefList(Sin,STin,STout,Sout) :-
      parseFnDef(Sin,STin,ST,X),
      eat(scolon,X,Y),
      parseFnDefList(Y,ST,STout,Sout).
```

Notice that consuming a semicolon doesn't involve the symbol table, so that parse predicate makes no reference to it. On the other hand, the definition reflects the fact that if `parseFnDefList` is matched, the `STout` should reflect the addition of symbols to `STin` as a result of repeated successes of the predicate `parseFnDef`.

One question which must be addressed, of course, is how to integrate the above definition into the original `parse` predicate. Originally, `parse` was defined as

```
parse(S) :- parseFnDefList(S, []).
%% parse(S) :- parseFnDefList(S, [eof] -- is another possibility
```

The parameters to `parseFnDefList` indicate that the predicate `parse` will succeed only if when the parsing is done all the input will have been consumed – i.e., the result list is empty. We need to do a similar thing if we add a symbol table. First, we must recognize the necessity to have the parser return the symbol table so it can be used by a subsequent interpreter predicate. Thus, we should define `parse` as follows

```
parse(S, ST) :- parseFnDefList(S, [], ST, []).
```

In this definition the second parameter to `parseFnDefList` represents the initial symbol table while the second represents the resulting symbol table.

Problem 14

Make a copy of the file containing your parser predicates. In this new file modify your parser predicates to 'pass' around a symbol table, as illustrated by the discussion above. What you should do at this point is simply add to the result symbol table any identifier you happen upon, either on the left or right of the assign token. As an example, you might try something like

```
parseFnDef(Sin,STin,STout,Sout) :-
    eat(id(N),Sin,X),
    parseParamList(X,[id(N)|STin],STa,Y),
    eat(assign,Y,Z),
    parseFnExpDef(Z,STa,STout,Sout).
```

When you execute the query

```
    -? tokens('prog.FP',S), parse(S, ST).
```

You should get a list containing the identifiers in your program, but there are bound to be duplicates. ∎

There are other aspects of the symbol table which we must consider. First, the entries in the symbol table must reflect the values. Since we want to represent scope, it would seem appropriate to make each entry a pair in which the first component is the identifier and the second component is a list of attributes; if an identifier is currently not in scope the attribute list can be empty. One advantage that *Prolog* presents is that the elements of a list needn't be of the same type. This means that an attribute list can have a triple to describe a function and a pair to describe an argument.

Example 7 – **Searching the Symbol Table**

To see if a particular identifier is in scope requires a simple search predicate. If we have a list of identifiers the following would work.

```
inScope(H, [H|Xs).
inScope(H, [Y|Xs]) :- H \== Y, inScope(H,Xs).
```

[The atom "`H \== Y`" indicate `H` is different from `Y`.] But if we have a symbol table as described above, then we must match more than just the identifier – we must match a pair whose first component is the identifier. The following will work.

```
inScope(H, [(H,A)|Xs).
inScope(H, [(Y,A)|Xs]) :- H \== Y, inScope(H,Xs).
```

This does what we wanted, but does it really solve the problem? No! Remember that we don't remove entries from the symbol table when there are no associated attributes – rather, an identifier is out of scope if it has no entry in the symbol table or its entry has an empty attribute list. We didn't check for that! Here is a correct implementation of `inScope`.

```
inScope(H, [(H,A)|Xs]) :- A \== [].
inScope(H, [(Y,A)|Xs]) :- H \== Y, inScope(H,Xs).
```

To avoid the inevitable "singleton variable" warning, we can rewrite the clauses as follows.

```
inScope(H, [(H,A)|_]) :- A \== [].
inScope(H, [(Y,_)|Xs]) :- H \== Y, inScope(H,Xs).
```

We can make another simplification. We can turn the first clause into a fact by using pattern matching to specify that the attribute list must be non-empty.

```
inScope(H, [(H,[_|_])|_]).
inScope(H, [(Y,_)|Xs]) :- H \== Y, inScope(H,Xs).
```

Now, what if we want to see if an identifier is of a particular type (function or parameter)? Here we will have to find the identifier in the list, but only succeed if the first item in the attribute list matches the specified type.

```
isFunction(H, [(H,[(function,_,_)|As])|Xs]).
isFunction(H, [(Y,_)|Xs]) :- H \== Y, isFunction(H,Xs).
```

Notice that the attribute entry for a function has three components: one for the type and two others for future use (the level number and the number of parameters, presumably).

∎

Searching a symbol table for an identifier of a certain type is one problem to be solved in implementing a symbol table. Another is updating the symbol table. This can be done in three ways: adding a new entry to the symbol table, adding a new attribute entry to the attribute list for an existing entry, or deleting attribute entries no longer needed (i.e., attributes of attributes which are going out of scope). Here is a sample of code from a parser which illustrates what sorts of things are possible.

```
   parse(S,ST) :- parseFnDefList(S,[],ST,[]).


   parseFnDefList(Sin,STin,STout,Sout) :-
        parseFnDef(Sin,STin,ST,X),
        eat(scolon,X,Y),
        parseFnDefList(Y,ST,STout,Sout).
   parseFnDefList(Sin,STin,STout,Sout) :-
        parseFnDef(Sin,STin,ST,Sout).


   parseFnDef(Sin,STin,STout,Sout) :-
        parseFnName(Sin,STin,STa,X),
        enterScope(STa, STb)
        parseParamList(X,STb,STc,Y),
        eat(assign,Y,Z),
        parseFnExpDef(Z,STc,STd,Sout),
        leaveScope(STd, STout).

   parseParamList(Sin,STin,STout,Sout) :-
        parseParamName(Sin,STin,ST,X),
        parseParamList(X,ST,STout,Sout).
   parseParamList(Sin,STin,STin,Sin) :-


   parseParamName([N|Sin],STin,STout,Sin) :-
        identifier(N),
        paramNotInTable(N,STin),
        addParamToTable(N,STin,STout).


   parseFnExpDef(Sin,STin,STout,Sout) :-
parseLet(Sin,STin,STout,X),
parseFnExp(X,STout,Sout).
   parseLet(Sin,STin,STout,Sout) :-
eat(let,Sin,X),
parseFnDefList(X,STin,STout,Y),
        eat(in,Y,Sout).


   parseFnExp(Sin,ST,Sout) :- checkName(Sin,arg,ST,Sout).
   parseFnExp(Sin,ST,Sout) :- eat(num(_),Sin,Sout).
   parseFnExp(Sin,ST,Sout) :- parseIfThenElse(Sin,ST,Sout).
   parseFnExp(Sin,ST,Sout) :- parseFunctionApp(Sin,ST,Sout).
```

```
parseFunctionApp(Sin,ST,Sout) :-
     checkFName(Sin,ST,X),
     parseTermList(X,ST,Sout).


parseIfThenElse(Sin,ST,Sout) :-
     eat(if,Sin,X),
     parseBExp(X,ST,Y),
     eat(then,Y,Z),
     parseFnExp(Z,ST,W),
     parseELSEpart(W,ST,Sout).

parseELSEpart(Sin,ST,Sout) :-
     eat(else,Sin,X),
     parseFnExp(X,ST,Sout).
parseELSEpart(Sin,ST,Sin).


parseTermList(Sin,ST,Sout) :-
     parseTerm(Sin,ST,X),
     parseTermList(X,ST,Sout).
parseTermList(Sin,ST,Sin).


parseTerm(Sin,ST,Sout) :-
     eat(num(_),Sin,Sout).
parseTerm(Sin,ST,Sout) :-
     checkName(Sin,arg,ST,Sout).
parseTerm(Sin,ST,Sout) :-
     eat(lp,Sin,X),
     parseFnExp(X,ST,Y),
     eat(rp,Y,Sout).



parseFnName([N|Sin],STin,STout,Sin) :-
     identifier(N),
     fnNotInTable(N,STin),
     addFnToTable(N,STin,STout).

checkName([N|Sin],Type,ST,Sin) :-
     identifier(N), defined((N,Type),ST).

checkFName(Sin,ST,Sout) :-
     checkName(Sin,fn,ST,Sout).
checkFName([+|Sin],ST,Sin).
checkFName([-|Sin],ST,Sin).
checkFName([*|Sin],ST,Sin).
checkFName([/|Sin],ST,Sin).
```

Problem 15

These predicates form a partially completed parser with symbol table, but there are some things needing completion. What needs completing are the implementations for the new predicates used here, such as `fnNotInTable`, `addFnToTable`, `defined`, etc.

1. Complete the definitions for the predicates which have not yet been defined. These should use some of the techniques discussed earlier in this section.
2. Complete the predicates `enterScope` and `leaveScope`. You will have to introduce into the symbol table a value for the current level and the level for each attribute set.

∎

When you complete this problem you will have a complete parser except for code translation. How would you propose implementing that code translation part of the parser? We have discussed in class the code generation needed for a function definition, and all it takes is to build a list of machine instructions and store them in a code table. Sounds easy enough! Give it a try...

8 *Prolog* Semantics - The Computational Model

In the earlier sections we have tried to give an intuitive idea of the syntax and semantics of *Prolog*. In this section we will look more carefully at the computational model which defines the semantics of *Prolog*. Remember that computation in *Prolog* centers on the extraction from a program of the sets of values which satisfy a query, where a query is just a conjunction of atoms. The computation cycle has two steps.

1. Select an atom from the query and determine if there is a fact or the head of a rule with which the atom can be unified. If there is none, then the query fails.

2. Resolve the query with the fact or rule by replacing the atom by the body of the rule (in the case of a fact this just means the elimination of the atom).

This cycle is repeated until the query fails or the query is eliminated. In the sub-sections which follow we will look in detail at the two steps of the cycle.

8.1 Unification

Unification is the process of finding a unifier for two atoms. In order to unify, two atoms must have the same structure, if we ignore constants and variables. Since atoms can contain function applications nested within their arguments, we require a recursive (i.e., stack based) algorithm.

Let's begin with a couple of definitions which will make the statement of the algorithm clearer.

Definition 1

A *term* is defined inductively as follows:

1. A variable or a constant is a *term*.
2. If f is an n-ary function name and $t_1, \ldots, t_n$ are *terms*, then $f(t_1, \ldots, t_n)$ is also a term.
3. Any expression is a *term* only if it can be constructed following steps 1. and 2. above.

∎

Definition 2

Let p be an n-ary predicate and $t_1, ..., t_n$ be terms. Then $p(t_1, ..., t_n)$ is an atom.

∎

Intuitively, the unification algorithm first checks that the two specified atoms start with the same predicate symbol – if not the atoms don't unify. If the predicate symbols are the same then the algorithm checks that the terms comprising the parameter list are of the same structure. A stack is created containing the pairs of corresponding terms (from the two atoms). At this point the algorithm repeatedly removes a pair of terms from the top of the stack and compares their structures. As long as they match the algorithm continues; if a critical difference is encountered then the atoms can't be unified and the algorithm terminates. The algorithm will continue to process the pairs from the stack until the stack is empty, at which time a unifier (in fact an mgu) will have been determined.

Before continuing you should review the definitions in Section 8.3 for *substitution*, *instance*, *unifier*, and *mgu*.

Definition 3

A Unification Algorithm
Let `A_1(t_1,...,t_n) and A_2(s_1,...,s_m)` be two atoms.

```
Algorithm:
          If A_1 and A_2 are different predicates or n <= m

          Then output Failed.

          Else
               Initialize the substitution to empty,
                    the stack to contain the pairs (t_i,s_i),
                    and Failure to false.

               While the stack is not empty & not Failure do
                     pop (X,Y) from the stack

                     case
                          X is a variable that does not occur in Y:
                            substitute Y for X in the stack
                            and in the substitution
                            add X=Y to the substitution

                          Y is a variable that does not occur in X:
                            substitute X for Y in the stack
                            and in the substitution
                            add Y=X to the substitution

                          X and Y are identical constants or variables:
                            continue

                          X is f(x_1,...,x_n) and Y is f(y_1,...,y_n)
                            for some function f and n>0:
                            push (x_i,y_i), i=1,...,n, on the stack

                          otherwise:  Failure = true

                If Failure, then output Failed
                Else output the substitution
```

∎

There are a couple of things to point out about the algorithm:

1. An important thing to point out here is that if two atoms unify, then this algorithm always returns the most general unifier for the two atoms.

2. In the first two components of the case-statement the phrase "does not occur in" is referred to as the *occurs check*. This condition guards against a substitution such as $X = f(X)$, which cannot be unified finitely.

   An interesting point is, the occurs check is very expensive computationally, so most implementations of *Prolog* omit it. This makes the implementation faster, but also makes the implementation unsafe.

Example 8 – **A Unification Example I** exnum

Consider the two atoms $p(a, f(Y))$ and $p(X, f(g(X)))$ – do they unify? Running the algorithm on these atoms leaves the following trace.

1. Initially: Since the predicates are the same we set the stack to be $[(a, X), (f(Y), f(g(X)))]$ and $\theta = \phi$.

2. Pop (a, X) from the stack, leaving $[(f(Y), f(g(X)))]$. By the second case we set $\theta = \{X = a\}$ and apply θ to the stack, giving $[(f(Y), f(g(a)))]$.
3. Pop $(f(Y), f(g(a)))$ from the stack, leaving `[ ]`. By the fourth case, we adjust the stack to be $[(Y, g(a))]$, with no adjustment to θ.
4. Pop $(Y, g(a))$ from the stack, leaving `[ ]`. By the first case we modify $\theta = \{X = a, Y = g(a)\}$ – since the stack is empty we don't have to apply θ.
5. The stack is empty to $\theta = \{X = a, Y = g(a)\}$ is the mgu for the two atoms.

▮

Example 9 – **A Unification Example II** exnum

Consider the two atoms $p(a, f(X))$ and $p(X, f(b))$ – do they unify? A quick examination leads us to believe the answer is *no*, because X would have to have both a and b as its substituted value. How does this come out of the algorithm? The following list reflects what happens during each pass through the algorithm.

1. Initially: Since the predicates are the same we set the stack to be $[(a, X), (f(X), f(b))]$ and $\theta = \phi$.
2. Pop (a, X) from the stack, leaving $[(f(X), f(b))]$. By the second case we set $\theta = \{X = a\}$ and apply θ to the stack, giving $[(f(a), f(b))]$.
3. Pop $(f(a), f(b))$ from the stack, leaving `[ ]`. By the fourth case, we adjust the stack to be $[(a, b)]$, with no adjustment to θ.
4. Pop (a, b) from the stack, leaving `[ ]`. By the third case, since a and b are different constants, the algorithm must return **failed**.

▮

8.2 Resolution

After the *Prolog* interpreter has accepted a query from the user the interpreter goes through a cycle which involves two activities: unification and resolution. We have just seen the unification algorithm. Before focusing on the resolution process, we will look at the actions of the interpreter in a bit more detail. When the interpreter inputs a query it has a program, consisting of facts and rules, and a query which is a conjunction of atoms. The interpreter executes the following algorithm.

```
While the query is not empty do
      select an atom from the query
      select a fact or rule with which the atom unifies

      If there is none, then return {\bf failed}
      Else
           If a fact was selected
           Then
                remove the atom from the query
           Else If a rule was selected
                replace the atom in the query by the
                body of the selected rule

return succeeded
```

This algorithm ignores many details, but carries the essence of the interpreter's activities. The adjustment to the query which occurs if unification succeeds is the resolution process. We will now detail what happens during resolution.

You will have noticed that in the algorithm above there is no mention of the mgu which results from the unification process. If all *Prolog* was meant to do was to say "success" or "failure", then the description above would be adequate. But the system should also return information about what variable substitutions satisfy a query. So we have to look more closely at the resolution process to see how the substitution values are determined.

Actually, there are two issues here. First, how do we use the mgu's returned by the unification algorithm to determine substitution values? Second, how do we make sure that we can return *all possible* substitution values? This second question involves a process called backtracking and is what happens when the user responds with a semi-colon to a returned substitution.

determining substitutions

backtracking

8.3 Queries

In the previous sections the notion of query was introduced. In this section we will take a closer look at queries and how they are processed.

Queries allow questions to be posed regarding information contained in a program. In the case that a query contains no variables, then the question posed is one with a yes/no answer. This type of query is called *ground query.* For example, the query

```
?- edge(a,b).
```

asks whether there is an edge from `a` to `b`. If the query is implied by the program, **Prolog** will return the answer `yes`. Otherwise the answer `no` is returned. Note that a period is also necessary following a query.

A slightly more complex query is one that contains more than a single atom. For example, the query

```
?- edge(a,b),edge(e,b).
```

poses the question: "Is there an edge from node `a` to node `b` AND from node `e` to node `b`?". Hence similar to commas occurring in bodies of rules, a comma that appears inside a query is treated as a conjunction. With respect to the program shown in Example 2.1, it is easy to see **Prolog** will answer `yes` to both of the queries

```
?- edge(a,b).          %% and
?- edge(a,b),edge(e,b).'
```

On the other hand, **Prolog** answers `no` to the query

```
?- edge(b,a).
```

since there is no edge from node `b` to node `a`.

Atoms in a query may contain variables. These are called *non-ground* queries. The query

```
?- edge(a,X).
```

is an example of a non-ground query. Variables that occur inside a query are treated *existentially.* This implies that their meaning is similar to variables that occur in the body of a rule but do not appear in the head of the same rule. The query

```
?- edge(a,X).
```

therefore, is posing the question: "Does there exist a node X such that there is an edge from node `a` to `X`?". Referring again to the program in Example 2.1, we would clearly expect **Prolog** to return a `yes` since node `a` is connect to nodes `b` and `e`. Indeed, this is the response of **Prolog**. However, in addition, **Prolog** will supply a *witness* (in this case a node) for the variable `X` in the query that explains why the query is true. Therefore, a witness for `X` in this query must either be `b` or `e`. To further clarify this, if we pose the query

```
?- edge(a,X).
```

Prolog responds by the answer

```
X = b
```

indicating the response "There is an `X` such that `edge(a,X)` is true, and `b` is one such `X`."

The **Prolog** mechanism for finding witnesses for variables is called *unification.* One may think of it as a generalized pattern-matching operation. The next few definitions formalizes this.

Definition 4

A **substitution** is a finite set of equalities

$$\{X_1 = t_1, X_2 = t_2, \ldots, X_n = t_n\}$$

where each X_i is a variable and t_i is a term, $X_1, \ldots, X_n$ are distinct variables, and X_i is distinct from t_i. Each $X_i = t_i$ is called a **binding** for X_i.

Typically, we denote substitutions by the Greek letters θ, γ, σ, possibly subscripted.

∎

Example 10 – **A Substitution**

The set $\theta = \{X = a, Y = b\}$ is a substitution, but $\gamma = \{X = a, X = b\}$ and $\sigma = \{X = a, Y = Y\}$ are not since in γ, the same variable X occurs twice on the left hand side of a binding. In σ, the binding $Y = Y$ violates the condition that each X_i must be distinct from t_i in the definition of substitution.

∎

Definition 5

Suppose θ is a substitution and E is an expression. Then $E\theta$, the **instance** of E by θ, is the expression obtained from E by simultaneously replacing each occurrence of the variable X_i in E by the term t_i, where $X_i = t_i$ is a binding in the substitution θ.

∎

Example 11 – **Another Substitution**

Suppose θ is the substitution $\{X = b, Y = a\}$, γ is the substitution $\{X = Y, Y = a\}$, and σ is the substitution $\{X = Y, Y = a, W = b\}$. Let E be the atom $p(X, Y, Z)$. Then

1. $E\theta = p(b, a, Z)$.
2. $E\gamma = p(Y, a, Z)$.
3. $E\sigma = p(Y, a, Z)$.

In the first case, the variables X and Y in E are replaced by b and a respectively. Note that the variable Z is not affected by the substitution θ since there is no binding of the form $Z = t$.

In the second case, it is important to observe that the replacement of variables by terms take place simultaneously. Thus the variable X in E is replaced by Y at the same time that Y is replaced by a. We do not further replace Y in the resulting expression by a to produce $p(a, a, Z)$.

The last case illustrates that there may exist bindings in the substitution that do not have any effect on the expression. In this since E does not contain the variable W, the binding $W = b$ in σ is superfluous.

∎

Definition 6

Suppose A and B are two expressions. A substitution θ is called a **unifier** of A and B if the $A\theta$ is syntactically identical to $B\theta$. We say A and B are **unifiable** whenever a unifier of A and B exists.

If, in addition, θ is the smallest possible substitution which unifies two atoms, we say that θ is the *most general unifier*, usually written *mgu*.

∎

simple queries

Let's see how these definitions apply to our example. When we issue the query

```
?- edge(a,X).
```

Prolog searches all `edge` related facts to find the first fact that unifies with the query. It turns out that in our program, the very first fact `edge(a,b)` is unifiable with the query since the substitution $\{X = b\}$ is a unifier of the two atoms `edge(a,X)` and `edge(a,b)`. The substitution corresponds to the answer returned by **Prolog**.

Previously we noted that for the query

```
?- edge(a,X).
```

either node `b` or node `e` may serve as a witness. **Prolog** returns `b` simply because the fact `edge(a,b)` appears before `edge(a,e)` in our program. So what good is `e` if we always get `b` as the answer? This is where the *backtracking* mechanism of **Prolog** comes in. When **Prolog** returns the answer `X = b`, the user has the option of commanding **Prolog** to search for other answers by typing a semicolon. The interaction between the user and **Prolog** is as follows.

```
?- edge(a,X).        %%% User poses a query
```

Prolog responds by the answer

```
X = b;               %%% The user types a semicolon after the answer
```

Prolog retries the query and returns another answer

```
X = e
```

If the user continues to type semicolon after each answer returned by **Prolog**, all possible answers will eventually be found.

conjunctive queries

Let us next examine how conjunctive queries are handled. These are queries that contain multiple atoms. In the case of ground queries, we have already discussed how **Prolog** answers. Let us then consider the non-ground conjunctive query

```
?- edge(a,X),edge(b,Y).
```

Since no variable is shared between atoms in the query, **Prolog** simply attempts to solve each atom independently. The only important part is that atoms are answered from left to right. In this case, **Prolog** first tries solving `edge(a,X)`. As we have seen already, this unifies with the fact `edge(a,b)` with the unifier $\{X = b\}$. Next the atom `edge(b,Y)` is tried. Searching through the set of facts related to `edge`, we find that the first unifying fact is `edge(b,d)` and the unifier is $\{Y = d\}$. Combining the two unifiers yields the substitution $\{X = b, Y = d\}$. This is the answer returned by **Prolog**.

```
?- edge(a,X), edge(b,Y).
```

Prolog returns the answers

```
X = b,
Y = d
```

An interesting question is what happens if the user requests for additional answers by typing a semicolon. This causes backtracking to take place, and the order in which atoms in the query are retried is based on the idea of "first tried, last re-tried". Equivalently we may think of this as "last tried, first re-tried". The essential idea is that upon backtracking, *Prolog* retries the last atom (or right most atom) in the query with which an alternative solution exists. In the above example, since `edge(b,Y)` is the solved last, it is retried first upon backtracking. Looking at our program, we see that in addition to the fact `edge(b,d)`, the fact `edge(b,c)` also unifies with the atom `edge(b,Y)` in the query with the unifier $\{Y = c\}$. Thus *Prolog* presents the answers

```
X = b,
Y = c
```

as the second solution. Note that in this case the first atom `edge(a,X)` is NOT retried. Next suppose we type yet another semicolon, prompting another backtracking to take place. Again, this causes `edge(b,Y)` to be retried. However, this time no more answer exists (i.e. we have exhausted all possible solutions for `edge(b,Y)`). This causes **Prolog** to trace back one more atom and retry `edge(a,X)`. This results in the new binding $\{X = e\}$. At this point, **Prolog** is not done. It now proceeds forward in the query to again solve `edge(b,Y)`, producing the answer

```
X = e,
Y = d
```

Finally, one last request for additional answers results in the

```
X = e,
Y = c
```

non-ground queries

In case a conjunctive query is non-ground, and contains variables that are shared among the atoms in the query, we need to modify the above description of how **Prolog** works slightly. Suppose the query is of the form

```
?- Atom1, Atom2, ..., Atomn.
```

The order in which the atoms are tried is still left to right, as was the case before. However, each time an atom is unified with a fact in the program, the unifier used must be applied to the remaining atoms in the query before proceeding. For example, if the query in question is

```
?- edge(a,X),edge(X,b).
```

then solving the first atom `edge(a,X)` causes a unification with the fact `edge(a,b)` using the unifier is $\theta = \{X = b\}$. Before trying the next atom `edge(X,b)` in the query, we must first apply the substitution θ to it. Computing `edge(X,b)`θ results in the atom `edge(b,b)`. This is the query that **Prolog** tries to solve next. Since `edge(b,b)` does not unify with any fact in the program, the query fails and backtracking takes place. As the last atom in the query with which an alternative solution exists is `edge(a,X)`, it is retried, causing this time a unification with the fact `edge(a,e)`. The new unifier is $\theta = \{X = e\}$. Again this is applied to the atom `edge(X,b)`. We have `edge(X,b)`θ = `edge(e,b)`. This is a ground atom that now matches with the fact `edge(e,b)` in the program. It follows that the query succeed and the answer returned by **Prolog** is

```
X = e
```

Programming in **Prolog** provides more flexibility than Scheme or Pascal/C for dealing with computations which may produce multiple (possibly infinitely many) possible results due to its backtracking capability. This allows multiple answers to be returned for a given input, as opposed to a single answer. However, this means that whatever we can do in the other languages, we should be able to do in **Prolog**. The exercise below involves writing a program to solve a problem that is not functional: For a given input, there may exist several outputs. This type of problem is generally called *relational.* [Can you figure out how to implement a *function* in **Prolog**? Try the standard factorial.]