Programming Languages: An Active Learning Approach ©2008, Springer Publishing

Chapter 7 Logic Programming

Kent D. Lee

Logic Programming

- Imperative programming languages reflect the architecture of the underlying von Neumann stored program computer: Programs update memory locations under the control of instructions.
- Imperative programs are prescriptive.
- They dictate precisely how a result is to be computed by means of a sequence of statements to be performed by the computer.

Logic Programming

Example 7.1

Consider this program using the language developed in chapter **??**.

```
let  val m = ref 0
     val n = ref 0
in
  read(m);
  read(n);
  while !m >= !n do m:=!m-!n;
  writeln(!m)
end
```

Logic Programming

- Languages for Logic Programming are called:
  - **Descriptive languages:** Programs are expressed as known facts and logical relationships about a problem. Programmers assert the existence of the desired result and a logic interpreter then uses the computer to find the desired result by making inferences to prove its existence.
  - **Nonprocedural languages:** The programmer states only what is to be accomplished and leaves it to the interpreter to determine how it is to be accomplished.
  - **Relational languages:** Desired results are expressed as relations or predicates instead of as functions. Rather than define a function for calculating a square root, the programmer defines a relation, say $sqrt(x, y)$, that is true exactly when $y^2 = x$.

- Assembly language is a very prescriptive language, meaning that you must think in terms of the particular machine and solve problems accordingly. Programmers must think in terms of the von Neumann machine stored program computer model.
- C++ and Ruby are high-level languages and hence allow you to think in a more descriptive way about a problem. However, the underlying computational model is still the von Neumann machine.
- ML is a high-level language too, but allows the programmer to think in a mathematical way about a problem. This language gets away from the traditional von Neumann model in some ways.
- Prolog takes the descriptive component of languages to the maximum and allows programmers to write programs based solely on describing relationships.

Getting Started with Prolog

Example 7.2

Prolog programs describe relationships. A simple example is a database of facts about several people in an extended family and the relationships between them.

```
parent(fred, sophusw). parent(fred, lawrence).
parent(fred, kenny). parent(fred, esther).
parent(inger,sophusw). parent(johnhs, fred).
parent(mads,johnhs). parent(lars, johan).
parent(johan,sophus). parent(lars,mads).
parent(sophusw,gary). parent(sophusw,john).
parent(sophusw,bruce). parent(gary, kent).
parent(gary, stephen). parent(gary,anne).
parent(john,michael). parent(john,michelle).
parent(addie,gary). parent(gerry, kent).
male(gary). male(fred).
male(sophus). male(lawrence).
male(kenny). male(esther).
male(johnhs). male(mads).
male(lars). male(john).
male(bruce). male(johan).
male(sophusw). male(kent).
male(stephen). female(inger).
female(anne). female(michelle).
female(gerry). female(addie).
father(X,Y):-parent(X,Y),male(X).
mother(X,Y):-parent(X,Y), female(X).
```

Questions we might ask are (1) Is Gary's father Sophus? (2) Who are Kent's fathers? (3) For who is Lars a father? These questions can all be answered by Prolog given this database.

Fundamentals

- Prolog programs (databases) are composed of facts.
- Facts describe relationships between terms.
- A predicate is a function that returns true or false.
- Frequently terms include variables in predicate definitions to establish relationships between groups of objects.
- In example 7.2 the `father` predicate is defined by writing

```
father(X,Y):-parent(X,Y), male(X).
```

Fundamentals

☞ Practice 7.1

What are the terms in example 7.2? What is the difference between an atom and a variable? Give examples of terms, atoms, and variables from example 7.2.

▸ View Solution

Fundamentals

Example 7.3

To discover if Johan is the father of Sophus you start Prolog using `pl` or `swipl`, then consult the database, and pose the query.

```
$ pl
?- consult('family.prolog').
?- father(johan,sophus).
Yes
?-
```

Fundamentals

Example 7.4

When using a variable in a query Prolog will answer yes or no. If the answer is yes, Prolog will tell us what the value of the variable was when the answer was yes. If there is more than one way for the answer to be yes then typing a semicolon will tell Prolog to look for other values where the query is true.

```
?- father(X, sophus).
X = johan
Yes
?- parent(X,kent).
X = gary ;
X = gerry ;
No
?-
```

The final `No` is Prolog telling us there are no other ways for `parent(X,kent)` to be true.

The Prolog Program

- Prolog performs something called unification to search for a solution.
- Unification is simply a list of substitutions of terms for variables.
- Prolog uses depth first search with backtracking to search for a valid substitution.
- Unification is a symmetric operation.

The Prolog Program

Example 7.5

In the following example we find out that `gary` is the father of `kent`. We also find out who `gary` is the father of.

```
?- father(X,kent).
X = gary ;
No
?- father(gary,X).
X = kent ;
X = stephen ;
X = anne ;
No
```

The Prolog Program

☞ Practice 7.2

Write predicates that define the following relationships.

1. brother
2. sister
3. grandparent
4. grandchild

Depending on how you wrote grandparent and grandchild there might be something to note about these two predicates. Do you see a pattern? Why?

▸ View Solution

Lists

Example 7.6

Append can be written as a relationship between three lists. The result of appending the first two lists is the third argument to the append predicate. The first fact below says appending the empty list to the front of `Y` is just `Y`. The second fact says that appending a list whose first element is `H` to the front of `L2` results in `[H|T3]` when appending `T1` and `L2` results in `T3`.

```
append([],Y,Y).
append([H|T1], L2, [H|T3]) :- append(T1,L2,T3).
```

Try out append both backwards and forwards!

Lists

Example 7.7

The definition of `append` can be used to define a predicate called `sublist` as follows:

```
sublist(X,Y) :- append(_,X,L), append(L,_,Y).
```

Stated in English this says that `X` is a sublist of `Y` if you can append something on the front of `X` to get `L` and something else on the end of `L` to get `Y`. The underscore is used in the definition for values we don't care about.
To prove that `sublist([1],[1,2])` is true we can use the definition of `sublist` and `append` to find a substitution for which the predicate holds. Here is an example of using sublist to prove that `[1]` is a sublist of `[1,2]`.

```
rule 1: sublist(X,Y) :- append(_,X,L), append(L,_,Y)
rule 2: append([ ], Y, Y).
rule 3: append([H l T], Y, [H l L]) :- append(T,Y,L).
```

$$\begin{tikzcd}
 & \begin{array}{c}\text{sublist([1],[1,2])}\\ \textit{Rule 1}\end{array} \arrow[dl] \arrow[dr] & \\
\begin{array}{c}\text{append(A,[1],L)}\\ \textit{Rule 2, A = [ ], L =[1]}\end{array} \arrow[d] & & \begin{array}{c}\text{append(L,R,[1,2])}\\ \textit{Rewrite using L = [1]}\end{array} \arrow[d] \\
\begin{array}{c}\text{append([ ],[1],[1])}\\ \textit{Fact}\end{array} & & \begin{array}{c}\text{append([1],R,[1,2])}\\ \textit{Rule 3}\end{array} \arrow[d] \\
 & & \begin{array}{c}\text{append([ ],R,[2])}\\ \textit{Rule 2, R = [2]}\end{array} \arrow[d] \\
 & & \begin{array}{c}\text{append([ ],[2],[2])}\\ \textit{Fact}\end{array}
\end{tikzcd}$$

Lists

☞ Practice 7.3

What is the complexity of the append predicate? How many steps does it take to append two lists?

▸ View Solution

Lists

☞ Practice 7.4

Write the reverse predicate for lists in Prolog using the append predicate. What is the complexity of this reverse predicate?

▸ View Solution

The Accumulator Pattern

- The slow version of reverse from practice problem 7.4 can be improved upon.
- The accumulator pattern can be applied to Prolog as it was in SML.

```
fun reverse(L) =
    let fun helprev (nil, acc) = acc
          | helprev (h::t, acc) = helprev(t,h::acc)
    in
      helprev(L,[])
    end
```

The Accumulator Pattern

☞ Practice 7.5

Write the reverse predicate using a helper predicate to make a linear time reverse using the accumulator pattern.

▸ View Solution

Unification and Arithmetic

- Writing `X=Y` in a predicate definition is never necessary.
- Instead, everywhere `Y` appears in the predicate, write `X` instead.
- There is no point in unifying a variable to a term if that variable is used only once in a predicate definition.
- Prolog warns us when we do this by saying

```
Singleton variables: [X]
```

- If this happens, look for a variable called `X` (or whatever the variable name is) that is used only once in a predicate definition and replace it with an underscore (i.e. _).
- To compute 6*5 and assign the result to the variable X the Prolog programmer writes **X is 6*5** as part of a predicate.
- Arithmetic can only be satisfied in one direction, from left to right.

Unification and Arithmetic

☞ Practice 7.6

Write a length predicate that computes the length of a list.

▸ View Solution

Input and Output

Example 7.8

```
? - readln(L,_,_,_,lowercase).
```

Reading input from the keyboard, no matter which predicate is used, causes Prolog to prompt for the input by printing a `|:` to the screen. If the `readln` predicate is invoked as shown above, entering the text below will instantiate `L` to the list as shown.

```
|: + 5 S R
L = [+, 5, s, r] ;
No
?-
```

Input and Output

Example 7.9

When a query is made in Prolog, each variable is given a unique name to avoid name collisions with other predicates the query may be dependent on. Prolog assigns these unique names and they start with an underscore character. If an uninstantiated variable is printed, you will see it’s Prolog assigned unique name.

```
?- print(X).
_G180
X = _G180 ;
No
```

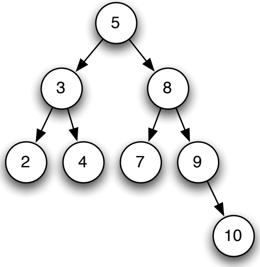
Structures

Example 7.10

Consider implementing a lookup predicate for a binary search tree in Prolog. A tree may be defined recursively as either `nil` or a `btnode(Val,Left,Right)` where `Val` is the value stored at the node and `Left` and `Right` represent the left and right binary search trees. The recursive definition of a binary search tree says that all values in the left subtree must be less than `Val` and all values in the right subtree must be greater than `Val`. For this example, let's assume that binary search trees don't have duplicate values stored in them.
A typical binary search tree structure might look something like this:

```
btnode(5,
  btnode(3,
    btnode(2, nil, nil),
    btnode(4, nil, nil)),
  btnode(8,
    btnode(7, nil, nil),
    btnode(9, nil,
      btnode(10, nil, nil))))
```

which corresponds to the tree shown graphically here.



Structures

☞ Practice 7.7

Write a lookup predicate that looks up a value in a binary search tree like the kind defined in example 7.10.

▸ View Solution

Parsing in Prolog

Example 7.11

Consider the following context-free grammar for English sentences.

Sentence ::= Subject Predicate .
Subject ::= Determiner Noun
Predicate ::= Verb | Verb Subject
Determiner ::= a | the
Noun ::= professor | home | group
Verb ::= walked | discovered | jailed

Parsing in Prolog

☞ Practice 7.8

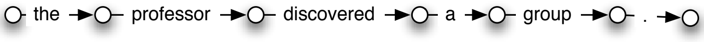
Construct the parse tree for “the professor discovered a group.”

▸ View Solution

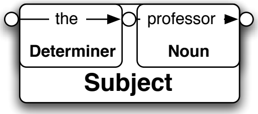
Parsing in Prolog

Example 7.12

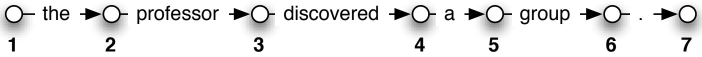
This is a graph representation of a sentence. Two terminals are contiguous in the original string if they share a common node in the graph.



A sequence of contiguous labels constitutes a nonterminal if the sequence corresponds to the right-hand side of a production rule in the grammar. The contiguous sequence may then be labeled with the nonterminal. In the diagram below three nonterminals are identified.



To facilitate the representation of this graph in Prolog the nodes of the graph are given labels. Positive integers are convenient labels to use.



The graph for the sentence can be represented in Prolog by entering the following facts. These predicates reflect the end points of their corresponding labeled edge in the graph.

```
the(1,2).
professor(2,3).
discovered(3,4).
a(4,5).
group(5,6).
period(6,7).
```

Parsing in Prolog

- The rule for the sentence predicate is

```
subject(K,L) :- determiner(K,M), noun(M,L).
```

Parsing in Prolog

☞ Practice 7.9

Construct the predicates for the rest of the grammar.

▸ View Solution

Parsing in Prolog

Example 7.13

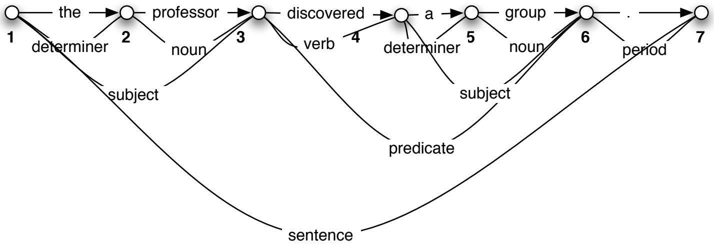
The syntactic correctness of the sentence, “the professor discovered a group.” can be determined by either of the following queries

```
?- sentence(1,7).
yes
? - sentence(X,Y).
X = 1
Y = 7
```

Parsing in Prolog

Example 7.14

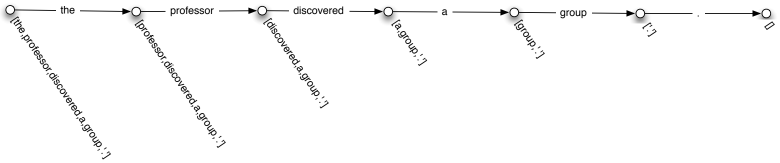
These are the paths in the graph of the sentence.



Difference Lists

Example 7.15

This is the difference list representation of the graph.



Difference Lists

Example 7.16

This is the connect predicate and the grammar rewritten to use the connect predicate.

```
c([H|T],H,T).
```

The `c` (i.e. connect) predicate says that the node labeled `[H|T]` is connected to the node labeled `T` and the edge connecting the two nodes is labeled `H`. This predicate can be used for the terminals in the grammar in place of the facts given above.

```
determiner(K,L) :- c(K,a,L).
determiner(K,L):- c(K,the,L).

noun(K,L) :- c(K,professor,L).
noun(K,L) :- c(K,home,L).
noun(K,L) :- c(K,group,L).

verb(K,L) :- c(K,walked,L).
verb(K,L) :- c(K,discovered,L).
verb(K,L) :- c(K,jailed,L).
```

Difference Lists

- The syntactic correctness of the sentence, “the professor discovered a group.” can be recognized by the following query.

```
?- sentence([the,professor,discovered,a,group,'.'], [ ]).
yes


?- sentence(S,[ ]).
```

- Some care must be taken when asking for all sentences of a grammar. If the grammar contained a recursive rule, say

```
Subject ::= Determiner Noun | Determiner Noun ``and'' Subject
```

- then the language would allow infinitely many sentences, and the sentence generator will get stuck with ever lengthening subject phrases.

Prolog Grammar Rules

Example 7.17

The grammar of the English language example takes the following form as a logic grammar in Prolog:

```
sentence --> subject, predicate,['.'].
subject --> determiner, noun.
predicate --> verb, subject.
determiner --> [a].
determiner --> [the].
noun --> [professor]; [home]; [group].
verb --> [walked]; [discovered]; [jailed].
```

Building an AST

Example 7.18

The grammar rule

```
sentence(sen(N,P)) --> subject(N), predicate(P), ['.'].
```

will be translated into the Prolog rule

```
sentence(sen(N,P),K,L) :- subject(N,K,M),
                          predicate(P,M,R),c(R,'.',L).
```

A query with a variable representing a tree produces that tree as its answer.

```
?- sentence(Tree, [the,professor,discovered,a,group,'.'],[]).
Tree = sen(sub(det(the),noun(professor)),
           pred(verb(discovered),sub(det(a),noun(group))))
```

Building an AST

☞ Practice 7.10

Write a grammar for the subset of English sentences presented in this text to parse sentences like the one above. Include parameters to build abstract syntax trees like the one above.

▸ View Solution

Exercises

1. In these exercises you should work with the relative database presented at the beginning of this chapter.
   1.1 Write a rule (i.e. predicate) that describes the relationship of a sibling. Then write a query to find out if Anne and Stephen are siblings. Then ask if Stephen and Mike are siblings. What is Prolog's response?
   1.2 Write a rule that describes the relationship of a brother. Then write a query to find the brothers of SophusW. What is Prolog's response?
   1.3 Write a rule that describes the relationship of a niece. Then write a query to find all nieces in the database. What is Prolog's response?
   1.4 Write a predicate that describes the relationship of cousins.
   1.5 Write a predicate that describes the relationship of distant cousins. Distant cousins are cousins that are cousins of cousins but not cousins. In other words, your cousins are not distant cousins, but second cousins, third cousins, and so on are distant cousins.

2. Write a predicate called odd that returns true if a list has an odd number of elements.
3. Write a predicate that checks to see if a list is a palindrome.
4. Show the substitution required to prove that sublist([a,b],[c,a,b]) is true. Use the definition in example 7.7 and use the same method of proving its true.
5. Write a predicate that computes the factorial of a number.
6. Write a predicate that computes the nth fibonacci number in exponential time complexity.
7. Write a predicate that computes the nth fibonacci number in linear time complexity.
8. Write a predicate that returns true if a third list is the result of zipping two others together. For instance,

   ```
   zipped([1,2,3],[a,b,c],[pair(1,a),pair(2,b),pair(3,c)])
   ```

   should return true since zipping [1,2,3] and [a,b,c] would yield the list of pairs given above.

9. Write a predicate that counts the number of times a specific atom appears in a list.
10. Write a predicate that returns true if a list is three copies of the same sublist. For instance, the predicate should return true if called as

    ```
    threecopies([a, b, c, a, b, c, a, b, c]).
    ```

    It should also return true if it were called like

    ```
    threecopies([a,b,c,d,a,b,c,d,a,b,c,d]).
    ```

Solution to Practice Problem 7.1

Terms include atoms and variables. Atoms include sophus, fred, sophusw, kent, johan, mads, etc. Atoms start with a lowercase letter. Variables start with a capital letter and include X and Y from the example.

▸ Go Back to Practice Problem

Solution to Practice Problem 7.2

1. `brother(X,Y) :- father(Z,X), father(Z,Y), male(X).`
2. `sister(X,Y) :- father(Z,X), father(Z,Y), female(X).`
3. `grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`
4. `grandchild(X,Y) :- grandparent(Y,X).`

Grandparent and grandchild relationships are just the inverse of each other.

▸ Go Back to Practice Problem

Solution to Practice Problem 7.3

The complexity of append is O(n) in the length of the first list.

Go Back to Practice Problem

Solution to Practice Problem 7.4

```
reverse([],[]).
reverse([H|T],L) :- reverse(T,RT), append(RT,[H],L).
```

This predicate has O(n^2) complexity since append is called n times and append is O(n) complexity.

▸ Go Back to Practice Problem

Solution to Practice Problem 7.5

```
reverseHelp([],Acc,Acc).
reverseHelp([H|T], Acc, L) :- reverseHelp(T,[H|Acc],L).
reverse(L,R):-reverseHelp(L,[],R).
```

Go Back to Practice Problem

Solution to Practice Problem 7.6

```
len([],0).
len([H|T],N) :- len(T,M), N is M + 1.
```

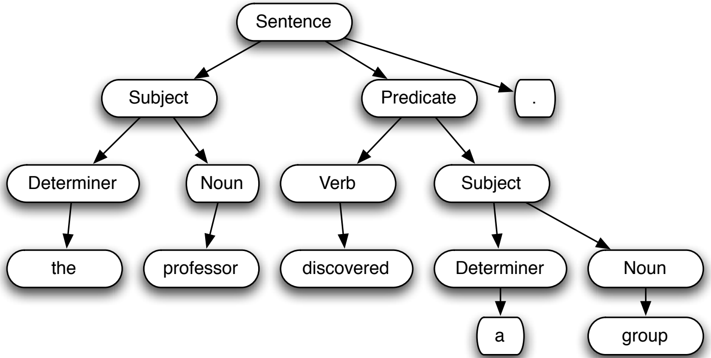
▸ Go Back to Practice Problem

Solution to Practice Problem 7.7

```
lookup(X,btNode(X,_,_)).
lookup(X,btNode(Val,Left,_)) :- X < Val, lookup(X,Left).
lookup(X,btNode(Val,_,Right)) :- X > Val, lookup(X,Right).
```

Go Back to Practice Problem

Solution to Practice Problem 7.8



Go Back to Practice Problem

Solution to Practice Problem 7.9

```
sentence(K,L) :- subject(K,M), predicate(M,N), period(N,L).
subject(K,L) :- determiner(K,M), noun(M,L).
predicate(K,L) :- verb(K,M), subject(M,L).
determiner(K,L) :- a(K,L); the(K,L).
verb(K,L) :- discovered(K,L); jailed(K,L); walked(K,L).
noun(K,L) :- professor(K,L); group(K,L); home(K,L).
```

Go Back to Practice Problem

Solution to Practice Problem 7.10

```
sentence(sen(N,P)) --> subject(N), predicate(P), ['.'].
subject(sub(D,N)) --> determiner(D), noun(N).
predicate(pred(V,S)) --> verb(V), subject(S).
determiner(det(the)) --> [the].
determiner(det(a)) --> [a].
noun(noun(professor)) --> [professor].
noun(noun(home)) --> [home].
noun(noun(group)) --> [group].
verb(verb(walked)) --> [walked].
verb(verb(discovered)) --> [discovered].
verb(verb(jailed)) --> [jailed].
```

Go Back to Practice Problem