Reading Assignment

- Webber: Chapters 19, 20 and 22

Prolog

Prolog presents a view of programming that is very different from most other programming languages.

A famous text book is entitled

"Algorithms + Data Structures = Programs"

This formula represents well the conventional approach to programming that most programming languages support.

In Prolog there is an alternative rule of programming:

"Algorithms = Logic + Control"

This rule encompasses a *non-procedural* view of programming.

Logic (what the program is to compute) comes first.

Then control (how to implement the logic) is considered.

In Prolog we program the logic of a program, but the Prolog system *automatically* implements the control.

Logic is essential—control is just efficiency.

Logic Programming

Prolog implements *logic programming*.

In fact Prolog means Programming in Logic.

In Prolog programs are statements of rules and facts.

Program execution is deduction—can an answer be inferred from known rules and facts.

Prolog was developed in 1972 by Kowalski and Colmerauer at the University of Marseilles.

Elementary Data Objects

- In Prolog *integers* and *atoms* are the elementary data objects.
- Integers are ordinary integer literals and values.
- *Atoms* are identifiers that begin with a lower-case letter (much like symbolic values in Scheme).
- In Prolog data objects are called *terms*.
- In Prolog we define *relations* among terms (integers, atoms or other terms).
- A *predicate* names a relation. Predicates begin with lower-case letters.
- To define a predicate, we write *clauses* that define the relation.

- There are two kinds of program clauses, *facts* and *rules*.
- A fact is a predicate that prefixes a sequence of terms, and which ends with a period (“.”).

As an example, consider the following facts which define “**fatherOf**” and “**motherOf**” relations.

```
fatherOf(tom,dick).
fatherOf(dick,harry).
fatherOf(jane,harry).
motherOf(tom,judy).
motherOf(dick,mary).
motherOf(jane,mary).
```

The symbols **fatherOf** and **motherOf** are predicates. The symbols **tom**, **dick**, **harry**, **judy**, **mary** and **jane** are atoms.

Once we have entered rules and facts that define relations, we can make queries (ask the Prolog system questions).

Prolog has two interactive modes that you can switch between.

To enter *definition mode* (to define rules and facts) you enter

`[user].`

You then enter facts and rules, terminating this phase with **`^D`** (end of file).

Alternatively, you can enter

`['filename'].`

to read in rules and facts stored in the file named **`filename`**.

When you start Prolog, or after you leave definitions mode, you are in *query mode*.

In query mode you see a prompt of the form

| **`?-`** or **`?-`** (depending on the system you are running).

In query mode, Prolog allows you to ask whether a relation among terms is *true* or *false*.

Thus given our definition of **`motherOf`** and **`fatherOf`** relations, we can ask:

```
| ?- fatherOf(tom,dick).
yes
```

A "yes" response means that Prolog is able to conclude from the facts and rules it has been given that the relation queried does hold.

```
| ?- fatherOf(georgeW,george).
no
```

A “no” response to a query means that Prolog is unable to conclude that the relation holds from what it has been told. The relation may actually be true, but Prolog may lack necessary facts or rules to deduce this.

Variables in Queries

One of the attractive features of Prolog is the fact that *variables* may be included in queries. A variable always begins with a capital letter.

When a variable is seen, Prolog tries to find a value (binding) for the variable that will make the queried relation true.

For example,

```
fatherOf(X,harry).
```

asks Prolog to find an value for **X** such that **X**'s father is **harry**.

When we enter the query, Prolog gives us a solution (if one can be found):

```
?- fatherOf(X,harry).
X = dick
```

If no solution can be found, it tells us so:

```
| ?- fatherOf(Y,jane).
no
```

Since solutions to queries need not be unique, Prolog will give us alternate solutions if we ask for them. We do so by entering a ";" after a solution is printed. We get a "no" when no more solutions can be found:

```
| ?- fatherOf(X,harry).
X = dick ;
X = jane ;
no
```

Variables may be placed anywhere in a query. Thus we may ask

```
| ?- fatherOf(jane,X).
X = harry ;
no
```

We may use more than one variable if we wish:

```
| ?- fatherOf(X,Y).
X = tom,
Y = dick ;
X = dick,
Y = harry ;
X = jane,
Y = harry ;
no
```

(This query displays all the **`fatherOf`** relations).

Conjunction of Goals

More than one relation can be included as the "goal" of a query. A comma (",") is used as an AND operator to indicate a conjunction of goals—all must be satisfied by a solution to the query.

```
| ?-
fatherOf(jane,X),motherOf(jane,Y).

X = harry,

Y = mary ;

no
```

A given variable may appear more than once in a query. The same value of the variable must be used in all places in which the variable appears (this is called *unification*).

For example,

```
| ?-
fatherOf(tom,X),fatherOf(X,harry).
X = dick ;
no
```

Rules in Prolog

Rules allow us to state that a relation will hold depending on the truth (correctness) of other relations.

In effect a rules says,

"If I know that certain relations hold, then I also know that this relation holds."

A rule in Prolog is of the form

$\texttt{rel}_1$ `:-` $\texttt{rel}_2$`,` $\texttt{rel}_3$`, ...` $\texttt{rel}_n$`.`

This says $\texttt{rel}_1$ can be assumed true if we can establish that $\texttt{rel}_2$ and $\texttt{rel}_3$ and all relations to $\texttt{rel}_n$ are true.

$\texttt{rel}_1$ is called the *head* of the rule.

$\texttt{rel}_2$ to $\texttt{rel}_n$ form the *body* of the rule.

Example

The following two rules define a **grandMotherOf** relation using the **motherOf** and **fatherOf** relations:

```
grandMotherOf(X,GM) :-
  motherOf(X,M),
     motherOf(M,GM).
grandMotherOf(X,GM) :-
  fatherOf(X,F),
    motherOf(F,GM).
| ?- grandMotherOf(tom,GM).
GM = mary ;
no
| ?- grandMotherOf(dick,GM).
no
| ?- grandMotherOf(X,mary).
X = tom ;
no
```

As is the case for all programming, in all languages, you must be careful when you define a rule that it correctly captures the idea you have in mind.

Consider the following rule that defines a **`sibling`** relation between two people:

```
sibling(X,Y) :-
 motherOf(X,M),motherOf(Y,M),
 fatherOf(X,F), fatherOf(Y,F).
```

This rule says that **`X`** and **`Y`** are siblings if each has the same mother and the same father.

But the rule is wrong!

Why?

Let's give it a try:

```
| ?- sibling(X,Y).
X = Y = tom
```

Darn! That's right, you can't be your own sibling. So we refine the rule to force **X** and **Y** to be distinct:

```
sibling(X,Y) :-
 motherOf(X,M), motherOf(Y,M),
 fatherOf(X,F), fatherOf(Y,F),
 not(X=Y).
```

(A few Prolog systems use "**\+**" for not; but most include a **not** relation.)

```
| ?- sibling(X,Y).
X = dick,
Y = jane ;
X = jane,
Y = dick ;
no
```

Note that distinct but equivalent solutions
(like `X = dick,Y = jane` vs. `X = jane,Y = dick`) often appear in Prolog solutions. You may sometimes need to "filter out" solutions that are effectively redundant (perhaps by formulating stricter or more precise rules).

How Prolog Solves Queries

The unique feature of Prolog is that it automatically chooses the facts and rules needed to solve a query.

But how does it make its choice?

It starts by trying to solve each goal in a query, left to right (recall goals are connected using ",", which is the and operator).

For each goal it tries to *match* a corresponding fact or the head of a corresponding rule.

A fact or head of rule matches a goal if:

- Both use the same predicate.
- Both have the same number of terms following the predicate.

- Each term in the goal and fact or rule head match (are equal), possibly binding a free variable to force a match.

For example, assume we wish to match the following goal:

```
x(a,B)
```

This can match the fact

```
x(a,b).
```

or the head of the rule

```
x(Y,Z) :- Y = Z.
```

But `x(a,B)` can't match

`y(a,b)` (wrong predicate name) or

`x(b,d)` (first terms don't match) or

`x(a,b,c)` (wrong number of terms).

If we succeed in matching a rule, we have solved the goal in question; we can go on to match any remaining goals.

If we match the head of a rule, we aren't done—we add the body of the rule to the list of goals that must be solved.

Thus if we match the goal **`x(a,B)`**

with the rule

```
x(Y,Z) :- Y = Z.
```

then we must solve **`a=B`** which is done by making **`B`** equal to **`a`**.

Backtracking

If we reach a point where a goal can't be matched, or the body of a rule can't be matched, we *backtrack* to the last (most recent) spot where a choice of matching a particular fact or rule was made. We then try to match a different fact or rule. If this fails we go back to the next previous place where a choice was made and try a different match there. We try alternatives until we are able to solve all the goals in our query or until all possible choices have been tried and found to fail. If this happens, we answer "no" the query can't be solved.

As we try to match facts and rules we try them in their order of definition.

Example

Let's trace how

```
| ?- grandMotherOf(tom,GM).
```

is solved.

Recall that

```
grandMotherOf(X,GM) :-
  motherOf(X,M),
     motherOf(M,GM).

grandMotherOf(X,GM) :-
  fatherOf(X,F),
    motherOf(F,GM).
fatherOf(tom,dick).
fatherOf(dick,harry).
fatherOf(jane,harry).
motherOf(tom,judy).
motherOf(dick,mary).
motherOf(jane,mary).
```

We try the first **grandMotherOf** rule first.

This forces **X** = **tom**. We have to solve

```
motherOf(tom,M),
     motherOf(M,GM).
```

We now try to solve

```
motherOf(tom,M)
```

This forces **M** = **judy**.

We then try to solve

```
motherOf(judy,GM)
```

None of the **motherOf** rules match this goal, so we backtrack. No other **motherOf** rule can solve

```
motherOf(tom,M)
```

so we backtrack again and try the second **grandMotherOf** rule:

```
grandMotherOf(X,GM) :-
   fatherOf(X,F),
      motherOf(F,GM).
```

This matches, forcing **X** = **tom**.

We have to solve

```
fatherOf(tom,F),
motherOf(F,GM).
```

We can match the first goal with

```
fatherOf(tom,dick).
```

This forces **F** = **dick**.

We then must solve

```
motherOf(dick,GM)
```

which can be matched by

```
motherOf(dick,mary).
```

We have matched all our goals, so we know the query is true, with **GM** = **mary**.

List Processing in Prolog

Prolog has a notation similar to "cons cells" of Lisp and Scheme.

The "." functor (predicate name) acts like cons.

Hence `.(a,b)` in Prolog is essentially the same as `(a . b)` in Scheme.

Lists in Prolog are formed much the same way as in Scheme and ML:

`[]` is the empty list

`[1,2,3]` is an abbreviation for `.(1, .(2, .(3,[])))`

just as

`(1,2,3)` in Scheme is an abbreviation for `(cons 1 (cons 2 (cons 3 () )))`

The notation `[H|T]` represents a list with `H` matching the head of the list and `T` matching the rest of the list.

Thus `[1,2,3]` ≡ `[1| [2,3]]` ≡ `[1,2| [3]]` ≡ `[1,2,3| [] ]`

As in ML, "_" (underscore) can be used as a wildcard or "don't care" symbol in matches.

Given the fact

```
p([1,2,3,4]).
```

The query

```
| ?- p([X|Y]).
```

answers

```
X = 1,
Y = [2,3,4]
```

The query

```
p([_,_,x|Y]).
```

answers

```
X = 3,
Y = [4]
```

List Operations in Prolog

List operations are defined using rules and facts. The definitions are similar to those used in Scheme or ML, but they are *non-procedural*.

That is, you don't given an execution order. Instead, you give recursive rules and non-recursive "base cases" that characterize the operation you are defining.

Consider **`append`**:

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

The first fact says that an empty list (argument 1) appended to any list **`L`** (argument 2) gives **`L`** (argument 3) as its answer.

The rule in line 2 says that if you take a list that begins with **H** and has **T1** as the rest of the list and append it to a list **L** then the resulting appended list will begin with **H**.

Moreover, the rest of the resulting list, **T3**, is the result of appending **T1** (the rest of the first list) with **L2** (the second input list).

The query

```
| ?- append([1],[2,3],[1,2,3]).
```

answers

```
Yes
```

because with **H=1**, **T1=[]**, **L2 =[2,3]** and **T3=[2,3]** it must be the case that **append([],[2,3],[2,3])** is true and fact (1) says that this is so.

Inverting Inputs and Outputs

In Prolog the division between "inputs" and "outputs" is intentionally vague. We can exploit this. It is often possible to "invert" a query and ask what *inputs* would compute a given output. Few other languages allow this level of flexibility.

Consider the query

```
append([1],x,[1,2,3]).
```

This asks Prolog to find a list **x** such that if we append **[1]** to **x** we will get **[1,2,3]**.

Prolog answers

```
x = [2,3]
```

How does it choose this answer?

First Prolog tries to match the query against fact (1) or rule (2).

Fact (1) doesn't match (the first arguments differ) so we match rule (2).

This gives us **`H=1`**, **`T1=[]`**, **`L2=X`** and **`T3 = [2,3]`**.

We next have to solve the body of rule (2) which is

```
append([],L2,[2,3]).
```

Fact (1) matches this, and tells us that **`L2=[2,3]=X`**, and that's our answer!

The Member Relation

A common predicate when manipulating lists is a membership test—is a given value a member of a list?

An "obvious" definition is a recursive one similar to what we might program in Scheme or ML:

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

This definition states that the first argument, **X**, is a member of the second argument (a list) if **X** matches the head of the list *or* if **X** is (recursively) a member of the rest of the list.

Note that we don't have to "tell" Prolog that **X** *can't* be a member of an empty list—if we don't tell

Prolog that something is true, it automatically assumes that it must be false.

Thus saying nothing about membership in an empty list is the same as saying that membership in an empty list is impossible.

Since inputs and outputs in a relation are blurred, we can use **`member`** in an unexpected way—to iterate through a list of values.

If we want to know if any member of a list **`L`** satisfies a predicate **`p`**, we can

simply write:

```
member(X,L),p(X).
```

There is no explicit iteration or searching. We simply ask Prolog to find an **`X`** such that **`member(X,L)`** is true (**`X`** is in **`L`**) *and* **`p(X)`** is true. Backtracking will find the "right" value for **`X`** (if any such **`X`** exists).

This is sometimes called the "guess and verify" technique.

Thus we can query

```
member(X,[3,-3,0,10,-10]),
   (X > 0).
```

This asks for an **`X`** in the list **`[3,-3,0,10,-10]`** which is greater than **`0`**.

Prolog answers

```
X = 3 ;

X = 10 ;
```

Note too that our "obvious" definition of member is not the only one possible.

An alternative definition (which is far less obvious) is

```
member(X,L) :-
   append(_,[X|_],L).
```

This definition says **X** is a member of **L** if I can take some list (whose value I don't care about) and append it to a list that begins with **X** (and which ends with values I don't care about) and get a list equal to **L**.

Said more clearly, **X** is a member of **L** if **X** is anywhere in the "middle" of **L**.

Prolog solves a query involving **member** by partitioning the list **L** in *all possible ways*, and checking to see if **X** ever is the head of the

second list. Thus for **`member(X,[1,2,3])`**, it tries the partition **`[]`** and **`[1,2,3]`** (exposing **`1`** as a possible **`X`**), then **`[1]`** and **`[2,3]`** (exposing **`2`**) and finally **`[1,2]`** and **`[3]`** (exposing **`3`**).