Reading Assignment

- Sethi: Chapter 11
- Scott: Section 11.3

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),
 \+(X=Y).
```

In Quintus prolog "\+" represents not; most other Prologs 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`.