3BA2 Prolog

Mike Brady

Prolog is Different

- Prolog is a different kind of programming language: it is a *declarative language*.
- The Prolog Language is also very similar to a form of logic called Horn Clauses; Prolog programs are executed as if they were logic statements, so that Prolog is an instance of a *Logic Programming Language*.
- The two properties of Prolog – its declarativeness and its similarity to logic - make it unusual.

Declarative Languges

- In a declarative language, what is specified is not what the machine will do exactly, but instead, a program consists of a set of definitions or "declarations" of how things are meant to be.
- In Prolog, for instance, a program is a set of true things - axioms.
- So here is a part of a Prolog program that defines the member relationship between an object and a list of objects:

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

Prolog Statement

- A Prolog statement is terminated by a full stop. The definition of membership of a list takes two statements.
- The first declares that anything `X` is a member of any list whose first element is `X` and the rest of which is the list `R`.
- The second statement declares that `X` is a member of any list as long as `X` is a member of the list less its first element.
- Together, these two statements *define* list membership.
- One of the key properties of a logic programming language is that these statements have the status of logical axioms - they are held to be true, and can be used to derive proofs of theorems later on, just like a normal formal system. They are *definitions*.

Prolog Syntax

- All Prolog objects are *terms*. Terms may be *simple* terms or *complex* terms.
- Simple terms have no internal structure.
- Complex terms have some organisation, and are often called *structured terms*.

Simple Terms – Atoms

- Simple terms have no internal structure. The simple terms are:

  Atoms - simple objects that have names.

  Syntax: Atoms must begin with a lowercase letter and have no non-alphanumeric characters, or they must be in quotes.

  Examples:

  ```
  a, michael, guinness, person, 'Funny.Lst'
  ```

Simple Terms – Numbers

- *Numbers* - in some Prologs, only integers are allowed.

Simple Terms – Variables

- *Variables* - a Prolog variable 'stands for' one particular Prolog object. During execution, the object may be bound to the variable (e.g. during clause matching). Once instantiated, the value of the variable may not change!

  Syntax: variable names begin either with an underscore or an uppercase letter, e.g. `_1034, A, X, Logic, Result`. The scope of a variable name is the clause it is in. When a variable is named only once in a clause, you can use the *anonymous variable:* `_`.

Complex Terms

- Complex terms have some organisation, and are often called *structured terms*. A complex term is composed of a functor and its arguments. A functor is an object that has a name and requires one or more arguments, depending on its arity. Each argument in a structured term is another term. So, for instance, the following are valid complex terms:

```
b(a,b,d), parent(john,bill),
direction(up,down,left,right),
+(3,2), :-(try(X),,(call(X),fail)).
```

Operators

- Operators
  Structured terms with an arity of `1` or `2` can be declared to be *operators*, which changes the way they are written [only].

  For instance, if + is declared to be an infix operator, then `+(3,2)` can be written `3+2`.

  Another example: if `:-` is declared to be infix, and `,` is also declared infix, then `:-(try(X),,(call(X),fail))` can be written:

```
try(X) :- call(X),fail.
```

- Thus, everything in Prolog is a Prolog term of some sort. Even the clauses are themselves Prolog terms.

Lists

- Lists are a very important kind of structured term. In fact, a list is structured term with a functor name `.` and an arity of `2`.
- The first argument is the *head* of the list & the second is the *tail*. The tail should be either a variable, another list, or the empty list `[]`, which is an atom(!)

Lists – Syntactic Sugar

- The following are lists:

```
.(a,[]), .(a,.(b,[])),
.(1,.(2,.(3,X))).
```

  In order to make these look more palatable, a special syntactic sugar is used to disguise them:

Lists – Syntactic Sugar (2)

- Firstly `.(Head,Tail)` can be written: `[Head|Tail]`

  so you could write:

  `[a|[]] [a|[b|[]]] [1|[2|[3|X]]]`

  Secondly, by supressing all `|[]` and replacing all instances like `X|[Y...` with `X,Y...` you get:

  `[a] [a,b], [1,2,3|X]`

Proving Something about Lists

- We can prove that the statement:

  `member(a,[3,a,491,b(c,d)]).`

  is true by reducing it to the axioms above:

  `member(a,[3,a,491,b(c,d)]) :- member(a,[a,491,b(c,d)]).`

  (true by 2nd statement)

  `member(a,[a,491,b(c,d)]).`

  (true by first statement.)
- QED.

Logic and Pattern Matching

- In the previous example, we tried to prove the assertion by matching it with the head(s) of the axioms.

  This matching is central to the operation of the Prolog interpreter, and it is an active kind of matching, and it is called *unification*.

- Terminology: the assertion we are trying to prove is called the *current goal*. The axioms are generally known as *clauses*. The first term in a clause is called the *clause head*.

  To match a goal, a clause head must have the same *functor* as the goal (i.e. it must have the same name & arity – number of arguments), and each of the arguments of the goal must match the arguments of the clause head.

Unification occurs as follows:

- If two arguments are identical, then they match.
- If one is an unbound variable, and the other is not, then the variable is bound to the other's value, and they then match.
- If both are unbound variables, then one variable is bound to the other, so that henceforward they are the same variable.
- Otherwise, matching fails.
- Thus, matching is an active process, rather than a passive operation.

Example

- Consider the unification of the following goal and clause:

```
concatenate([a,b,c],[1,2,3],X).
concatenate([H|T],R,[H|U]) :-
    concatenate(T,R,U).
```

- After unification, the variables are bound as follows:

```
X = [H|U] = [a|U]
H = a
T = [b,c]
R = [1,2,3]
```

U is unbound

- The set of bindings that is made in a successful unification is called a *substitution*.

Clause Instances

- A Prolog clause is an axiom that could be used in a proof. It typically specifies variables that could be used. E.g.

  append([],X,X)

- When it comes to actually *using* an axiom, a separate copy is constructed, with its own unique set of variables. This copy is called a *clause instance* and a clause instance is said to be *instantiated* when it's created.
- From an implementation point of view, a clause instance is very similar to a *function instance*, where each instance of the function has its own separate stack frame containing its private copies of local variables, etc.

Prolog and Theorem Proving

- A *theorem* is a "A proposition that has been or is to be proved on the basis of explicit assumptions."
- The goal of the Prolog interpreter is to prove an assertion, i.e. to make a theorem out of the assertion, using the clauses provided in the program and also using predicates provided as part of the Prolog system – the *built-in predicates*.

  Frequently, for that reason, the assertion is called, simply, the *goal*, and Prolog is called a *theorem prover*.

Prolog and Inferencing

- Like any theorem prover, Prolog works by using inferring new theorems from existing axioms and theorems. This activity – making inferences – is called *inferencing*.
- Prolog makes an inference by matching its current goal with a clause instance it can be unified with. This is called *resolution* – the current goal is *resolved* with a clause instance, and is replaced by the [possibly empty] body of the clause instance.