Lecture Overview

Prolog

1. **Introduction**
    - **Interaction**
    - **Terms**
2. **Clauses and predicates**
    - **Clauses**
    - **Predicates**
    - **Variables**
3. **Satisfying goals**

Prolog

A standard free Prolog can be downloaded from

http://www.swi-prolog.org

This is the version of prolog that we will use as reference for lectures.

Prolog is a **logic** language. A prolog program is a collection of **facts** and **rules**. Prolog uses these facts and rules to try to prove **goals**, which are typically entered at a prolog **system prompt**.

The system prompt is **?-** and we will write this before any user input. Like Haskell, we can store programs in files and load them into Prolog, and we can type things at the system prompt.

```
?- write('Hello, world!').

Hello, world!
true

?-
```

To Prolog, “write(‘Hello, world’)” is a goal. A goal is terminated by a period. Multiple goals can be separated with commas. The “write…” goal has the **side-effect** of writing a string to the output, but after that, it **evaluates** to **true**. Prolog treats every goal as a theorem to prove. Merely by executing the write, Prolog gets a **true**, so it considers the goal accomplished (or the theorem proved).

Note that Prolog responds with the side-effect (the writing of “Hello, world!”) and with the evaluation of the goal, which is **true**. It follows that up with another system prompt.

Here’s how to make a **query** out of multiple goals:

```
?- write('Hello!'), nl, write('Goodbye!'), nl.
```

The **nl** goal has the side-effect of printing a newline.

In order for the query to succeed (i.e. return **true**), each of the constituent goals must succeed (in order).

Both **write** and **nl** are predefined by the Prolog system. They are known as **built-in predicates**, or BIPs. Two other BIPs are:

```
?- halt
```

which causes the Prolog system to terminate, and

```
?- statistics
```

which prints system statistics.

A Prolog program has components known as **clauses**, each terminated with a period. A clause can be a **fact** or a **rule**. For example:

```
dog(fido).
cat(felix).

animal(X):-dog(X).
```

This program has three clauses. The first two are facts and the last is a rule. The facts represent the statements “fido is a dog” and “felix is a cat”. The rule represents “X is an animal if X is a dog”. The operator :- is read as “if”. Note the inclusion of a blank line for readability; Prolog ignores such blanks.

If we keep the above program in a file, say, “prog1.pl”, then we may load it into the Prolog system using the BIP **consult**.

```
?- consult(‘prog1.pl’).
```

consult will succeed (return **true**) if the file exists and contains a well-formed Prolog program.

Loading a program causes its clauses to be placed in a storage area known as the **Prolog database**. Entering a query causes Prolog to search its database for clauses with which it can prove the goals of the query.

In the example, **dog**, **cat**, and **animal** are **predicates**. They each take one argument. **fido** and **felix** are **atoms** (an atom is a constant that is not a number). Finally, **X** is a **variable**.

With the program consulted (loaded), we can query about fido and felix:

```
?- animal(fido).
true

?- animal(felix).
false

?- cat(X).
X = felix.
```

In that last query, we are asking "is there a cat named X" where X is a variable that can be filled in with anything. Prolog responds by telling us that by assigning X to be felix, we can make the query true.

If there are multiple assignments of a variable which will make a query true, then Prolog prints one of them and pauses. You can then type ; (semicolon) to get the next assignment, or return if you don't want to see the other assignments.

Suppose we had the program

```
dog(fido)
dog(rover)

cat(felix).
cat(mittens).
cat(tom).
cat(phidippides).
```

Then if we had the query

```
?- cat(X).
```

Prolog would respond with

```
X = felix
```

and if we pressed ; then it would print

```
X = mittens
```

and if we again pressed ; it would print

```
X = tom
```

and if we again pressed ; it would print

```
X = phidippides.

?-
```

with the period and the prompt indicating that it was finished and there were no more alternatives to list. We could also use the BIP **listing**:

```
?- listing(cat).
cat(felix).
cat(mittens).
cat(tom).
cat(phidippides).

true.
```

listing causes Prolog to print the clauses that define listing’s argument (in this case, cat), in the order in which they were loaded into the database.

The query

```
?- cat(X),dog(Y).
```

gives all possible combinations of a cat and a dog.

```
X = felix,
Y = fido ;
X = felix,
Y = rover ;
X = mittens,
Y = fido ;
X = mittens,
Y = rover                    etc.
```

In contrast, the query

```
?- cat(X), dog(X)
```

gives all names which are **both** a cat and a dog. Since there are no such names in the database, this query does not succeed (it returns **false**).

Comments

Comments are written enclosed in /* … */ delimiters.

Data objects in Prolog

1. **Numbers.** Integers are allowed, possibly preceded by a + or – sign. Floats are allowed as well.
2. **Atoms**. Atoms are constants without numerical values. There are three ways in which an atom can be written:
   a. A sequence of letters, numerals, and underscores, starting with a lower case letter.
   b. Any sequence of characters enclosed in single quotes.

   c. Any sequence of one or more special characters from a list that includes + - * / > < = & # @

3. **Variables**. A variable is a name used to stand in for a term that is to be determined in a query. They can also be used in facts and rules. A variable is written as a sequence of letters, numerals, and underscores, beginning with an upper case letter or underscore.

   The special case of _ is reserved for the **anonymous variable**.

4. **Compound terms.** A compound term is a structured data type that begins with an atom called the **functor**. The functor is followed by a sequence of one or more **arguments**, which are enclosed in parentheses and separated by commas.

   $functor(t_1, t_2, \ldots t_n)$ $(n \geq 1)$

Think of a compound term as a record data structure. The functor is the name of the record and the arguments are the record fields.

The number of arguments of a compound term is called its **arity**.

Each argument must be a term (of any kind).

5. **Lists**. A list is a term. Lists elements can be any terms, and they are written enclosed in square brackets and separated by commas.

```
[dog, cat, y, predi(A, b, c), [p, q, R], z]
```

The empty list is written as [].

Clauses and Predicates

Apart from comments and blank lines, Prolog programs consist of a succession of **clauses**. A clause can run over more than one line or there may be several on the same line. A clause is terminated by a period followed by a whitespace character.

There are two types of clause: **facts** and **rules**. Facts are of the form

head.

where **head** is an atom or a compound term. **head** is called the *head of the clause*. Here are some examples of facts:

```
canadaDay.
likes(john,mary).
likes(X,prolog).
dog(rover).
```

A rule is of the form:

head:-$\mathbf{t_1}$**,**$\mathbf{t_2}$**,…,**$\mathbf{t_k}$**.** ($k \geq 1$)

Here, **head** is again called the *head of the clause* (or the *head of the rule*).

:- is called the *neck of the clause (rule)* and is sometimes known as the **neck operator.** It is read as "if".

t_1,t_2,…,t_k is called the *body of the clause (*or *rule*). These are conditions that must all be true for the head to be true.

Another way to look at a rule is to consider the head to be a goal that needs to be proven, and the body as subgoals that each need to be shown true in order to establish the goal.

Predicates

Unfortunately, the same name may be used as the functor for compound terms of different arities. The name may also be used as an atom. For instance, the following is legal:

```
parent(victoria, albert).
parent(john).
animal(parent).
```

These are three distinct things all called **parent**. Don't do this, though, as it is confusing!

All the clauses for which the head has a given combination of functor and arity comprise a definition of a **predicate**. The clauses do not have to appear as consecutive lines of the program but it makes the program easier to read if they do.

For instance, in the program:

```
parent(victoria, albert).
parent(X, Y):-father(X, Y).
parent(X, Y):-mother(X, Y).
father(john, henry).
mother(jane, henry).
```

the first three clauses define a predicate with the name **parent**, with arity two. We'll sometimes write this as **parent/2**. (But that's not valid Prolog text, just a notation we use when writing about Prolog.)

The query

```
?- listing(parent)
```

gives a listing of all of the clauses for the predicate **parent** no matter what the arity.

Declarative and Procedural Interpretation of Rules

Rules have both a **declarative** and a **procedural** interpretation. The rule

```
chases(X,Y):-dog(X), cat(Y), write(X), write(‘ chases ‘),
    write(Y), nl.
```

has declarative interpretation:

chases(X,Y) is true if dog(X) is true and cat(Y) is true
and write(X) is true, and …

but it also has the procedural interpretation:

to satisfy chases(X, Y), first satisfy dog(X), then satisfy
cat(Y), then satisfy write(X), then …

Clauses that are atoms are interpreted declaratively.

```
canadaDay.
```

means

canadaDay is true.

Users cannot redefine BIPs. Some BIPs are **write/1, nl/0, repeat/0, member/2, append/3, consult/1,** and **halt/0.**

Simplifying entry of goals

In developing or testing programs it can be tedious to enter repeatedly at the system prompt a lengthy series of goals such as:

```
?-dog(X),large(X),write(X),write(‘ is a large dog’),nl.
```

A common technique is to define a predicate such as **go/0** or **start/0** with the above sequence of goals as the body of the rule. For example,

```
go:-dog(X),large(X),write(X),write(‘ is a large dog’),nl.
```

Then one only needs type the small predicate (e.g. **go**) to get Prolog to try to achieve the goals.

Recursion

Recursion is frequently used when defining predicates. Recursion can be direct or indirect. Here’s an example of direct recursion:

```
likes(john, X):-likes(X,Y),dog(Y)
```

(“John likes anyone who likes a dog.”)

Predicates

Predicates are relations between a number of values (its arguments) which can be either **true** or **false**. This contrasts with functions, which can evaluate to any type of object (e.g. a number, a string, a point, etc.).

Loading clauses

As mentioned earlier, the BIP **consult/1** causes Prolog to read the clauses in the file given as consult's argument. If the same file is consulted again during a Prolog session, then **all the clauses from the first consultation are removed from the database** before the new consultation's clauses are loaded.

Square brackets are an abbreviation for **consult/1:**

```
    ?-['testfile1.pl].
```

means the same as

```
    ?-consult('testfile1.pl').
```

Loading clauses from more than one file can have interesting effects, particularly if both files try to define the same predicate. (Only the last version of the predicate will be kept.) So beware!

Variables

Variables can be used in the head or body of a clause, and in goals entered at the system prompt. Their interpretation depends on where they are used.

Variables in goals.

Variables in goals can be interpreted as "find values of the variables that make the goal satisfied". For example, the goal

```
    ?-animal(A)
```

means to find values of **A** such that **animal(A)** is satisfied.

The lexical scope of variables is restricted to the clause in which they are used. In the following program, there are two different variables called X, one in each rule.

```
    dog(rover).
    large(rover).
    large_animal(X):-animal(X),large(X).
    animal(X):-dog(X).
```

Quantification.

If a variable appears in the head of a rule or a fact it is taken to indicate that the rule or fact applies **for all possible values of the variable**. Such a variable is said to be **universally quantified**. For example,

```
animal(X):-dog(X).
```

means that for any value that X can take on, animal(X) is true if dog(X) is true.

On the other hand, if a variable appears in the body of a rule it is part of a goal that is satisfied if **there exists a value of the variable** that satisfies the goal. Such a variable is said to be **existentially quantified.** For example,

```
dogowner(X):-dog(Y),owns(X,Y)
```

means that for any value X can take on, X is a dogowner if **there exists** a Y such that Y is a dog and X owns Y.

The anonymous variable.

Suppose you have a **person** predicate with fields for the name, the gender, the occupation, and the city they live in. You have several facts that look like:

```
person(stacey, female, electrical engineer, vancouver).
```

You could write a rule for when a person is female:

```
	female(X):-person(X, female, A, B).
```

However, you really aren't concerned with the values of A and B here. In that case, you can use the anonymous variable _ rather than a named variable.

```
	female(X):-person(X, female, _, _).
```

This usage is similar to Haskell. Note that there is no assumption that the anonymous variables share the same value. Each appearance of the anonymous variable denotes a new variable.

Satisfying goals

We look more closely at how Prolog satisifies goals. The process starts when the user types a query at the command prompt.

The Prolog system attempts to satisfy each goal in the query in turn, working from left to right. When a goal involves variables, this generally involves binding them to values. If all the goals succeed in turn, the whole query succeeds, and the system will output the values of the variables used in the query.

A **call term** is an atom or a compound term. It cannot be a number, variable, or list. Every goal must be a call term.

Goals relating to BIPs are evaluated in a way predefined by the Prolog system (consider **write/1** and **nl/0**). Goals relating to user-defined predicates are evaluated by examining the database of rules and facts loaded by the user.

Prolog attempts to satisfy a goal by matching it with the heads of clauses in the database, working from top to bottom.

For example, the goal

```
?-dog(X).
```

might be matched with the fact

```
dog(fido).
```

and give the output

```
X = fido
```

Any goal that cannot be satisfied using the facts and rules in the Prolog database **fails**. There is no intermediate position, such as “unknown” or “not proven”. This means that the database is assumed to have all knowledge about the world (the “closed world assumption”).

Unification

Prolog uses a very general form of matching known as **unfication.** (We’ve seen it before in type unification.) Unification generally involves one or more variables being given values in order to make two call terms identical; this is known as **binding** the variables to the values. For example, the terms **dog(X)** and **dog(fido)** can be unified by binding variable **X** to the atom **fido**.

Initially, all variables are unbound. Unlike in most langauges, once a variable is bound, it can be made unbound again and then perhaps be bound to a new value by the process of **backtracking**.

To unify two call terms, we follow the following algorithm:

```
if call terms are both atoms
    if they are the same atom
        succeed
    else
        fail

if call terms are not both compound terms
    fail

if call terms do not have the same functor and arity
    fail

if arguments to the call terms unify pairwise
    succeed
else
    fail
```

To unify other terms:

- Two numbers unify iff they are the same number.
- Two unbound variables always unify, with the two variables becoming bound to each other.
- An unbound variable and a term that is not a variable always unify, with the variable becoming bound to the term.
- A bound variable is treated as the value to which it is bound.
- Two lists unify iff they have the same number of elements and their elements can be unified pairwise, working from left to right.
- All other combinations of terms fail to unify.

Here are some examples of unification:

```
person(X, Y, Z)
person(john, smith, 27)
```

succeeds with X bound to *john*, Y bound to *smith*, and Z bound to *27*.

```
person(john, Y, 23)
person(X, smith, 27)
```

fails because 23 cannot be unified with 27.

```
pred1(X, X, man)
pred1(london, dog, A)
```

fails. In the first argument, X is unified with *london*, resulting in X being **bound** to the value *london*. In the second argument, X is treated as the value it is bound to (*london*) and this cannot be unified with *dog*.

```
pred2(X, X, man)
pred2(london, london, A)
```

succeeds with X bound to *london*, and A bound to *man*.

```
pred3(alpha, pred4(X, X, Y))
pred3(P, pred4(no, yes, maybe))
```

fails because X cannot unify with both *no* and *yes*.

```
pred3(alpha, pred4(X, X, Y))
pred3(P, pred4(no, no, maybe))
```

succeeds with P bound to *alpha*, X bound to *no*, and Y bound to *maybe*.

Evaluating goals

Suppose that we have a database with the clauses:

```
capital(london, england).
european(england).
pred(X, 'european capital') :-
    capital(X, Y), european(Y), write(X), nl.
```

and we wish to evaluate the query

```
?-pred(london, A).
```

The goal in the query will unify with the head of the rule in the database, with X bound to *london* and A bound to 'european capital'. Prolog now tries to evaluate the rule by evaluating the terms in the body of the rule, in turn.

- First it tries to evaluate **capital(X, Y)**, which due to X being bound to *london* is interpreted as **capital(london, Y)**. To evaluate this, it looks for matching terms in the database, and finds

capital(london, england), which matches (unifies) with Y becoming bound to *england*. So capital(X, Y) is satisfied.

- Next it tries to evaluate **european(Y)**, which is treated as **european(england)** because Y is bound to *england*. This is matched to the fact **european(england)** in the database. A match to a fact always succeeds, so european(X, Y) is satisfied.
- Next it tries to evaluate **write(X)**, which is treated as **write(london)** because X is bound to *london*. **write** writes **london** to the output and succeeds.
- Next it tires to evaluate **nl**. This writes a newline to the output and succeeds.

Since all four goals on the right-hand side of the rule succeeded, the rule itself succeeded, and the query succeeds with A bound to ‘european capital’.

If at any time one of the predicates being evaluated fails, then the Prolog system backs up to the previous predicate and tries to find another way of satisfying it. If this fails, it backs up again, etc. This process is known as **backtracking**.

Summary: evaluating a sequence of goals

Evaluate the goals in turn, working from left to right. If they all succeed, the whole sequence of goals succeeds. If one fails, go back through the previous goals in the sequence one by one from right to left trying to resatisfy them. If they all fail, the whole sequence fails. As soon as one succeeds, start working through the goals from left to right again.

Summary: evaluating/re-evaluating a goal

Search through the clauses in the database, working from top to bottom (start at the top for evaluation, or after the last clause matched for re-evaluation). Search until a clause is found whose head matches with the goal. If the matching clause is a fact, the goal succeeds. If it is a rule, evaluate the sequence of goals in its body. If the sequence succeeds, the goal succeeds. If not, continue searching through the database for further matches. If the end of the database is reached, the goal fails.

Renaming common variables

Sometimes in matching a term to another both will have a variable in common. For example, suppose we are matching

```
farmer(john, X).
```

with

```
farmer(X, smith).
```

We want this match to succeed.

Because of the lexical scoping of variables, these two Xs actually refer to different variables. So Prolog systematically replaces the variable names in the clause being matched with variable names that do not appear elsewhere. For instance, the above example could be made into matching

```
farmer(john, X).
```

with

```
farmer(X_1239, smith)
```

where 1239 is some unique sequence number.

Declarative programming

It should be clear that the order in which the clauses defining a predicate occur in the database and the order of goals in the body of a rule are of vital importance when evaluating a user's query.

It is part of the philosophy of logic programming that programs should be written to minimize the effect of these two factors as much as possible. Programs that do so are called fully or partly *declarative*.

Here's a simple example of a fully declarative program:

```
dog(fido). dog(rover). dog(ruff). dog(spike).
cat(bill). cat(fluffy). cat(fuzzy). cat(hairy).

large(rover). large(william). large(tweety).
large(fluffy). large(hairy).

large_animal(X):- dog(X), large(X).
large_animal(Z):- cat(Z), large(Z).
```

We can rearrange the clauses in this program into any order and still get the same results for any query. (The results may be in a different order.) We can also rearrange the subgoals in the rules defining **large_animal/1** without changing the results returned.

To contrast, here's a nondeclarative program for signum:

```
signum(X):-X>0, write(positive), nl.
signum(0):-write(zero), nl.
signum(X):-write(negative), nl.
```

This relies on the third clause only being reached when X is negative (or not a number!). Putting the third clause in first place would change the behaviour.

A better, declarative, way to write this is:

```
signum(X):-X>0, write(positive), nl.
signum(0):-write(zero), nl.
signum(X):-X<0, write(negative), nl.
```

Now the order of the clauses makes no difference to the result.

Keeping programs declarative greatly reduces the likelihood of making errors that are hard to detect.