University of Malta
BSc(Hons.) IT Year II

Problem Solving Techniques in Artificial Intelligence

Lecture 2

Prolog Review

- Simple Read/Write
- Failure
- Negation as Failure
- The Cut
- Arithmetic
- List Traversal

Department of Computer Science and AI
1998/9

Simple Read and Write

Simple reading and writing can be achieved using read/1 and write/1.

read/1

read/1 reads a term (of any complexity) from the current input stream (the terminal by default) and unifies it with its argument. Thus

?- read(X). (you type, e.g. "foo.")
X=foo
yes

Note that the argument can be complex, resulting in the binding of several variables:

?- read(X+Y). (you type, e.g. "3+2.")
X=3
Y=2
yes

write/1

write/1 writes its argument to the current output stream (again the terminal by default).

?- write(happy(christmas,X)).

happy(christmas,_57)

true ?

Failure

If at any point there is no clause that matches a given query, the query is said to *fail*. So given the database

```
happy(eeny).
happy(meeny).
```

the query *?- happy(X)* succeeds.

```
?- happy(X)
X=eeny;
```

Upon typing the semicolon, it succeeds again

```
X=meeny;
```

After a second semicolon, it fails, because there are no more results.

no

The built-in `fail`

Another way to provoke a failure, is by means of the built-in `fail`. This can be used to provoke backtracking: With the database and query as follows

```
happy(eeny).
happy(meeny).
happy(miny).
happy(mo).

?- happy(X), write(X), nl, fail.
```

the output is:

```
eeny
meeny
miny
mo
no
```

Note that the complex query as a whole fails.

Negation by Failure

If the query p succeeds, then the query \+p fails, and vice versa. The symbol \+, pronounced "not" expresses negation by failure.

Note that if X is an unbound variable, the query \+p(X) always leaves the variable X unbound.

Checkpoint

Assuming the above database, what is the difference in behaviour between between

```
?- happy(X), write(X).

?- sad(X), write(X).
?- \+ \+ sad(X), write(X)
?- \+ \+ happy(X), write(X)
```

The Cut

The cut predicate, written "!" tells the system to omit some of the backtrack points. When a cut is executed

- It succeeds immediately.
- Alternatives to the present clause will not be executed.
- Alternatives preceding the cut in the present clause will not be attempted.

Cut Example

```
a(X) :- b(X).
a(w).
b(X) :- c(X), !, d(X).
b(y).
b(w).
c(w).
d(z).

compare:
?- b(y). (succeed)
?- b(w). (fail)
?- a(w). (succeed)
```

Exercise

Given the following database:

```
x(X) :- a(X), b(X), !.
x(X) :- c(X).
a(aa).
b(aa).
c(cc).
```

what is the effect of the following queries

(a) ?- x(aa).
(b) ?- x(cc).
(c) ?- x(What), write(What), nl, fail.

What do foo and baz do?

```
foo(X) :- X, !, fail.
foo(X).

baz([_|_]) :- !, fail.
baz(_).
```

Expressing Repetition 1: repeat

The built-in predicate `repeat` is considered to have an infinite number of solutions, so that execution can backtrack over it an infinite number of times:

```
?- repeat,
    write('I run forever...'),
   fail.
```

More commonly the execution is stopped by executing a cut after some test has succeeded, e.g.

```
copy_terms_from_input :-
  repeat,
     read(X),
     write(X), nl,
     X == end_of_file, % the test
  !.
```

- Note the use of till the end of line.
- Note use of "==" which would fail if X were an unbound variable.
- What would happen if the "!" were omitted from this example?

Abbreviations

Disjunction

If two subgoals are joined by ";" rather than ",", they are alternatives. Thus the clause

p :- q ; r.

is entirely equivalent to the pair of clauses

p :- q.
p :- r.

If-then

```
p -> q ; r
```

The system first tries to execute p. If this succeeds, it executes q, otherwise it executes r.

Exercise Translate the following definition into one not involving **->**.

```
f :- p -> q ; r
```

Modifying the Database

- asserta/1 adds a new clause to the front of the database.
- assertz/1 adds a new clause to the back of the database.
- retract/1 removes a clause in the database. For instance retract(green(X)) removes the first clause unifying with green(X), simultaneously binding X to whatever was in the argument position.

Exercise

Assuming that there are no clauses for "green" in the database, what would be the effect of:

```
?- assertz(green(kermit)),
   asserta(green(eggs+ham)),
   assertz(green(cheese)),
   asserta(green(grass)),
   green(X), write(X), nl, fail.
```

Arithmetic

Recall that an expression like '2+3' is simply an unintepreted term that is completely distinct from the term "5".

In order to evaluate such a term as an arithmetic expression, we use the built-in predicate "is", whose left operand is either a variable or a numerical constant, and whose right operand is an arithmetic expression involving the operators +,-,*,/ and parentheses. Thus.

```
?- N is 2+3.
N = 5
```

Note that in order to evaluate the expression, the argument to "is" must be fully instantiated at the time the query is made, so (a) is fine, whilst (b) will give an error.

(a) ?- X=2, Y=3, N is X+Y.
(b) ?- Y=3, N is X+Y.

Exercise

1. ?- N = 3+2, N is N.
2. ?- N = 3+2, M is N.

Recursion

Prolog has two mechanisms for repeating: failure, which we have already seen, and recursion, in which a procedure calls itself.

```
% sum(M,N,S)
% S is the sum of M and N

sum(N,0,N).
sum(M,N,S) :-
 N>0,
 N1 is N-1,
 M1 is M+1,
 sum(M1,N1,S).
```

Trace

```
?- sum(2,2,Z).
   Call:  (  7) sum(2, 2, _G287) ? creep
   Call:  (  8) 2>0 ? creep
   Exit:  (  8) 2>0 ? creep
 ^ Call:  (  8) _L142 is 2-1 ? creep
 ^ Exit:  (  8) 1 is 2-1 ? creep
 ^ Call:  (  8) _L143 is 2+1 ? creep
 ^ Exit:  (  8) 3 is 2+1 ? creep
   Call:  (  8) sum(3, 1, _G287) ? creep
   Call:  (  9) 1>0 ? creep
   Exit:  (  9) 1>0 ? creep
 ^ Call:  (  9) _L156 is 1-1 ? creep
 ^ Exit:  (  9) 0 is 1-1 ? creep
 ^ Call:  (  9) _L157 is 3+1 ? creep
 ^ Exit:  (  9) 4 is 3+1 ? creep
   Call:  (  9) sum(4, 0, _G287) ? creep
   Exit:  (  9) sum(4, 0, 4) ? creep
   Exit:  (  8) sum(3, 1, 4) ? creep
   Exit:  (  7) sum(2, 2, 4) ? creep
```

List Traversal

```
% list-pr(L)
% Print each element of list L

list-pr([]).
list-pr([X|Y]) :-
 write(X),
 list-pr(Y).

?- list-pr([hello,world]).
   Call:  (  7) list-pr([hello, world]) ? creep
   Call:  (  8) write(hello) ? creep
hello   Exit:  (  8) write(hello) ? creep
   Call:  (  8) list-pr([world]) ? creep
   Call:  (  9) write(world) ? creep
world   Exit:  (  9) write(world) ? creep
   Call:  (  9) list-pr([]) ? creep
   Exit:  (  9) list-pr([]) ? creep
   Exit:  (  8) list-pr([world]) ? creep
   Exit:  (  7) list-pr([hello, world]) ? creep
```