Prolog III

Lists

• \([\ ]\) is the empty list.
• \([x, 2+2, [a, b, c]]\) is a list of three elements.
• The first element in the list is its “head”.
• The list with the head removed is the “tail”.

Matching Heads and Tails

• If \([a, b, c] = [Head \mid Tail]\), then
  \[a = Head\] and \([b, c] = Tail\]
• If \([a, b, c] = [X, Y \mid Tail]\), then
  \[a = X, \ b = Y,\] and \([c] = Tail\]
• If \([a, b, c] = [X, Y, Z \mid Tail]\), then
  \[a = X, \ b = Y, \ c = Z,\] and \([\ ] = Tail\]
• The tail of a list is always itself a list.
• \([X \mid Y, Z]\) isn’t legal.
Making Use of Unification

- Prolog has no functions. But you can use a parameter as an “output variable.”
  - \( \text{first}([\text{Head} \mid \text{Tail}], X) :- X = \text{Head}. \)
- You can use unification in parameter lists to do much of the needed work
  - \( \text{first}([X \mid _], X). \)
  - \( \text{second}([_, X \mid _], X). \)
  - \( \text{third}([_, _, X \mid _], X). \)

Structures and Lists

- The “univ” operator, =.., can be used to convert between structures and lists:
  - \( \text{loves(chuck, X) =.. [loves, chuck, X]} \)
- Double quotes indicate a list of ASCII values:
  - ”abc” = [97, 98, 99]
  - This isn’t usually very useful

Recursion

- Recursion is fully supported
  - \( \text{element}(1, [X \mid _], X). \)
  - \( \text{element}(N, [\_ \mid X], Y) :- M \text{ is } N - 1, \text{element}(M, X, Y). \)
- This is the typical way to process lists: do something with the head, recur with the tail.

member

- \( \text{member}(X, [X \mid _]). \)
- \( \text{member}(X, [_ \mid Y]) :- \text{member}(X, Y). \)
- As usual, base cases go first, then recursive cases.
- There is in general no need for a “fail” case, because that’s automatic.
  - \( \text{member}(\_, []). :- \text{fail}. \)
Accumulated Information

• If you reach a clause, you can assume that the earlier clauses of the same predicate have failed.
• `member(X, [X | _ ]).`
• If you fail this clause, the first element is not the one you want, so `member(X, [ _ | Y ] :- member(X, Y).`

Backtracking and Beads

• Each Prolog call is like a “bead” in a string of beads:
  
  \[
  \text{loves(chuck, X) :- female(X), rich(X).}
  \]

Fail Loops

• It is possible to build a “fail loop” in Prolog
• `print_elements(List) :- member(X, List), write(X), nl, fail.`
• But recursion is almost always better:
  `print_elements([Head|Tail]) :- write(Head), nl, print_elements(Tail).`

Forcing a predicate to succeed

\[
\text{notice_objects_at(Place) :- at(X, Place), write('There is a '), write(X), write(' here.'), nl, fail.}
\]

\[
\text{notice_objects_at(_).}
\]
Forcing a predicate to fail

```
loves(chuck, X) :-
    really_ugly(X), !, fail.
loves(chuck, X) :-
    female(X), rich(X).
```

"Wrapping" another predicate

```
• The buzz_off/0 predicate might succeed or fail. This is usually what we want.
• But sometimes we want to ignore failure.

optional_buzz_off :-
    buzz_off.
optional_buzz_off.
```

Asserting Clauses

```
• assert(new_clause).
  - assert(path(garden, n, toolshed)).
  - assert(( loves(chuck,X) :- female(X) ,
            rich(X) )).
• asserta(new_clause).
• assertz(new_clause).
```

Removing clauses

```
• retract(clause).
  - retract(path(garden, n, toolshed)).
  - retract(path(X, Y, X)).
  - retract(( loves(chuck,X) :- female(X) ,
            rich(X) )).
• abolish(path, 3).
```
Marking Clauses as “Dynamic”

• Standard Prolog allows you to assert and retract clauses without any restrictions.
• Sicstus and some others require you to mark variable clauses as “dynamic.”
• :- dynamic i_am_at/1, at/2, alive/0.
• The “:-” at the beginning says “do it now.”

Solving problems with dynamic

• If Prolog already knows a clause, and it's static, it's too late to mark it dynamic.
• Prolog must see :- dynamic functor/arity before it sees any clauses of functor/arity.
  – This includes clauses loaded in from an earlier consult.
• You can restart Sicstus Prolog, or…
• …you can use abolish(funnel, arity)

Arithmetic

• The equals sign, =, means “unify.”
• 2+2 does not unify with 4.
• To force arithmetic to be performed, use “is”:
  X is 2 + 2, X = 4.
• Comparisons =:= /= > >= < <= also force their operands to be evaluated.
• + - * / mod, when evaluated, have their usual meanings.

The End