Prolog Language Tutorial (II)

Outline

- Built-in Predicates and Functions
  - Input / Output
- Control
  - Cut
  - Negation as Failure
- Recursive Programming Examples
  - Member
  - Append
  - Reverse
  - Gcd
  - Tree traversal

Built-in Predicates

- Most Prolog systems provide many built-in predicates and functions such as
  - Artithmetic functions (+, -, mod, is, sin, cos, floor, exp, …)
  - Bit-wise operations (/\, \/, \, <<, >>)
  - Term comparison (==, \==, @<, @>, …)
  - Input/Output (read, write, nl, …)
  - Control
  - Meta-logical
  - …

Built-in Predicates and Functions

Term	Variable, Constant
Expression (Expr)	Arithmetic expression (must not contain any uninstantiated variables)

$\underbrace{\texttt{Term is Expression}}_{\text{evaluated}}$

Expr1 <Op> Expr2

Term1 <Op> Term2

Arithmetic functions	Used in Expression	+, -, *, /, // (integer division), mod
Arithmetic predicates	Expr1 <Op> Expr2	<, =<, >, >=, =:= (have the same value), =\=
Term comparators	Term1 <Op> Term2	==, \==, @<, @=<, @>, @>=

Reference: http://www.sics.se/sicstus/docs/latest/html/sicstus/Built-Intro.html#Built%20Intro

Built-in Predicates and Functions

```
?- 3 = 1+2.
no
```

```
?- 3 is 1+2.
yes
```

```
?- 5-2 is 1+2.
no
```

```
?- 4-1 =:= 1+2.
yes
```

```
?- X is 1+2.
X = 3
```

```
?- X+2 = 1+Y.
X = 1
Y = 2
```

```
?- X+1 is 3.
no
```

```
?- X is 1+2, Y is X+4.
X = 3
Y = 7;
no
```

```
?- X+2 =:= 1+2.
{Instantiation error:...}
?- 3 is X+1.
{instantiation error: ...}
```


Built-in Predicates

- `read(X)`
  - Read the next term from current input stream and unify it with X
  - If no more characters, X is unified with atom `end_of_file`
  - Eg. `get_num(X) :- read(X), number(X), X>=1, X=<9.`
- `write(X)`
  - Write the term X to the current output stream
- `nl`
  - Start a new line on the current output stream
  - Eg. `write('---+---+---'), nl, write('     '), write(5), nl.`

Built-in Predicates

- In Prolog, we can modify a (running) program during execution, thus creating the effect of global variable
- To insert a fact or rule, use *assert(clause)*
  - `assert(colour(apple,red))`
  - `assert((arc(A,C) :- arc(A,B), arc(B,C)))`
- To remove a fact or rule, use *retract(clause)*
  - `retract(drink(tea))`
  - `retractall(drink(_))`
    - `/* drink(tea), drink(coffee), drink(coke), … */`

http://www.sics.se/sicstus/docs/3.7.1/html/sicstus_10.html#SEC108

Built-in Predicates

- For simple programs, the use of modifying predicates like `assert, asserta, assertz, retract, retractall` is NOT encouraged
- However, they are useful for more advanced programming techniques like memorization.

Prolog Lists

- Lists are a collection of terms inside [and]
    - [chevy, ford, dodge]
    - loc_list([apple, broccoli, crackers], kitchen).
    - loc_list([desk, computer], office).
    - loc_list([flashlight, envelope], desk).
    - loc_list([stamp, key], envelope). loc_list(['washing machine'], cellar).
    - loc_list([nani], 'washing machine').
    - loc_list([], hall)

Prolog Lists

- Unification works on lists just as it works on other data structures.
    - ?- loc_list(X, kitchen).
      - X = [apple, broccoli, crackers]
    - ?- [_,X,_] = [apples, broccoli, crackers].
      - X = broccoli
- The patterns won't unify unless both lists have the same number of elements.

Prolog Lists

- List functions
  - [H|T]
    - separate list into head and tail
  - member
    - test if X is a member of a list
  - append
    - append two lists to form a third list

Prolog Lists

- Head and Tail of a List
- Syntax

  [H|T]
- Examples
- ?- [a|[b,c,d]] = [a,b,c,d].

  yes
- ?- [a|b,c,d] = [a,b,c,d].

  no

Prolog Lists

- More Examples

  ?- [H|T] = [apple, broccoli, refrigerator].
  - H = apple
  - T = [broccoli, refrigerator]

  ?- [H|T] = [a, b, c, d, e].
  - H = a
  - T = [b, c, d, e]

  ?- [H|T] = [apples, bananas].
  - H = apples
  - T = [bananas]

Prolog Lists

- More Examples

  ?- [One, Two | T] = [apple, sprouts, fridge, milk].
  - One = apple
  - Two = sprouts
  - T = [fridge, milk]

  ?- [a|[b|[c|[d|[]]]]] = [a,b,c,d].
  yes

Prolog Lists

- Testing if an element is in a list.
- Syntax
  – member(X, L).
- Example
  – member(apple, [apple, broccoli, crackers]).
  – member(X, CarList).
- Full Predicate defined as:
  member(H,[H|T]).
  member(X,[H|T]) :- member(X,T).

Prolog Lists

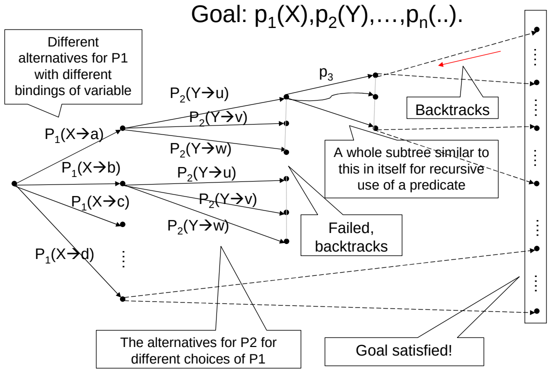
- Appending two lists to form a third.
- Syntax
  – append(L1, L2, L3).
- Example
  – append([a,b,c], [d,e,f], X).
  – X = [a,b,c,d,e,f]
- Full predicate defined as:
  append([],X,X).
  append([H|T1],X,[H|T2]) :- append(T1,X,T2).

Control

- The semantics of Prolog programs does not care about order
- Eg. `P :- Q,R,S.` should mean the same thing as `P :- S,R,Q.` because conjunction is commutative
- In practice, the order matters when side effects are involved, because most Prolog systems use left to right DFS, top to bottom order
- Besides placing the facts and rules in a suitable sequence, Prolog has other constructs to specify control information

Control – Visualizing Backtracking: DFS of a tree

Goal: $p_1(X),p_2(Y),\ldots,p_n(..)$.



Cut

- colour(black).
- colour(white).

- ?- colour(C).
- C = black? /* press ; to backtrack */
- C = white? /* press ; to backtrack */
- no

- Normally, Prolog backtracks when more solutions are needed, or the current instantiations fail a predicate

19

Cut

- Eg. we have the following facts
  - before(a,b).
  - before(b,c).
  - before(a,d).
  - before(b,d).

```
?- before(a,X).
X = b? /* press ; */
X = d? /* press ; */
no
```

- If we modify the first rule into
  - before(a,b) :- **!** .
- Then, only the first result of X is obtained
- Backtracking stops at the *Cut symbol (!) for the clause before*

```
| ?- before(a,X).
X = b ? ;
no
```

20

Cut – If-then-else

- Cut (written as ***!***)
  - For controlling the search
  - When it is first encountered as a goal, it succeeds
  - If backtracking returns to the cut, it fails the parent-goal (Head of the rule)
  - Using cut can usually reduce memory usage as less backtracking points are stored

```
x :- p, !, q.
x :- r.
```

<< If p is ture, then q is reached but not r
<< If p is not true, r is reached but not q

Overusing Cut will destroy the logic !!

21

Backtracking (the cut !)

```
correct(A) :- 1 is 1-0, 2 is 2-0, write(A).
correct(A) :- write(-A).
```

```
| ?- correct(1).
1
yes
```

```
correct(A) :- 1 is 1-0, 2 is 2-2, write(A).
correct(A) :- write(-A).
```

```
| ?- correct(1).
-(1)
yes
```

```
correct(A) :- 1 is 1-0, !, 2 is 2-0, write(A).
correct(A) :- write(-A).
```

```
| ?- correct(1).
1
yes
```

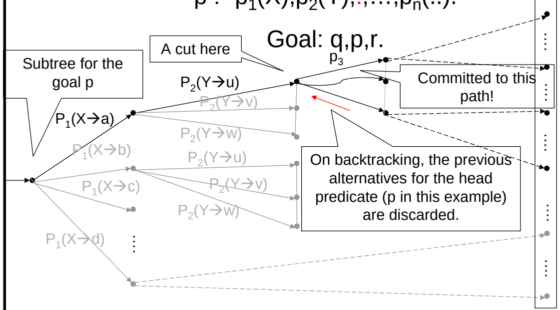
```
correct(A) :- 1 is 1-0, !, 2 is 2-2, write(A).
correct(A) :- write(-A).
```

```
| ?- correct(1).
no
```

22

Visualizing Backtracking with Cut

p :- p_1(X),p_2(Y),!,…,p_n(..).



23

Negation as Failure

- Prolog provides an "is-not-provable" operator \+ P.
- It is defined as if by
  - `\+(P) :- P, !, fail.`
  - `\+(P).`

<< fail is a predicate which is always false

- Examples
  - `legal(X) :- \+ illegal(X).`
  - `q :- \+(p).`

<< Q is true whenever P fails…

  - `different(X,Y) :- X=Y, !, fail.`
  - `different(X,Y).`

<< The head goal will be satisfied if X and Y can't be unified.

24

Recursive Programming Examples - member

- Define member(X,Y) to be true iff X (a term) is a member of the list Y
- In Prolog
  - member(X,[X|_]).
  - member(X,[_|T]) :- member(X,T).
- In Lisp
  - (defun member (x y)
    - (cond
      - ((null y) '())
      - ((equal x (car y)) t)
      - (t (member x (cdr y)))))

Recursive Programming Examples - member

- Define member(X,Y) to be true iff X (a term) is a member of the list Y
- In Prolog
  - member(X,[X|_]).
  - member(X,[_|T]) :- member(X,T).

```
?- member(a,[c,a,t]).
yes
```

```
?- member(d,[c,a,t]).
no
```

```
?- member(L,[c,a,t]).
L = c? ;
L = a? ;
L = t? ;
no
```


Recursive Programming Examples - member

- Note that in Prolog
  - we need NOT explicitly check for empty list
  - because an empty list cannot be unified with the clauses
  - so member(a,[]) is automatically not provable
- Also the Prolog version is more flexible in that it can be used in more than one direction
- Elements of the list can be obtained through repeated backtracking

Recursive Programming Examples

- A closer look at member(L,[c,a,t]).
  - member(X,[X|_]). << rule 1
  - member(X,[_|T]) :- member(X,T). << rule 2

Goal	Rule	Variables and subgoal
member(L,[c,a,t])	Rule 1	L→X→c, X→c, _→[a,t] succeeds
	Rule 2	L→X1, _→c, T→[a,t] Subgoal: member(X1,[a,t])
member(X1,[a,t])	Rule 1	X1→X2→a, X2→a, _→[t] succeeds, so gives L→X1→a
	Rule 2	X1→X3, _→a, T3→[t] Subgoal: member(X3,[t])
member(X3,[t])	Rule 1	X3→X4→t, X4→t, _→[] succeeds, so gives L→X1→X3→X4→t
	Rule 2	X3→X5, _→t, T5→[] Subgoal: member(X5,[]) which fails to unify with any clause

Recursive Programming Examples - append

- Define append(X,Y,Z) to be true iff the list X appended to the list Y gives the list Z
- In Prolog
  - append([],X,X).
  - append([H|T],P,[H|Q]) :- append(T,P,Q).
- In Lisp
  - (defun append (x y)
    - (cond
      - ((null x) y)
      - (t (cons (car x) (append (cdr x) y)))))

Recursive Programming Examples - append

- Define append(X,Y,Z) to be true iff the list X appended to the list Y gives the list Z
- In Prolog
  - append([],X,X).
  - append([H|T],P,[H|Q]):- append(T,P,Q).

```
?- append([a,b,c],[d,e],L).
L = [a,b,c,d,e]? ;
no
```

```
?- append([a,b,c],L,[a,b,c,d,e]).
L = [d,e] ;
no
```

```
?- append(L1,L2,[a,b]).
L1 = [], L2 = [a,b]? ;
L1 = [a], L2 = [b]? ;
L1 = [a,b], L2 = []? ;
no
```

Recursive Programming Examples - append

- Note that the explicit use of `car`, `cdr` and `cons` are replaced by unification
- Checking of empty list is also implicit
- Again, this `append` can be used in more than one direction

```
?- append([a,b,c],[d,e],L).
L = [a,b,c,d,e]? ;
no
```

```
?- append([a,b,c],L,[a,b,c,d,e]).
L = [d,e] ;
no
```

```
?- append(L1,L2,[a,b]).
L1 = [], L2 = [a,b]? ;
L1 = [a], L2 = [b]? ;
L1 = [a,b], L2 = []? ;
no
```


Recursive Programming Examples - append

- A closer look at `append(L1,L2,[a,b]).`
  - `append([],X,X).` << rule 1
  - `append([H|T],P,[H|Q]) :- append(T,P,Q).` << rule 2

<table>
<tr><th>Goal</th><th>Rule</th><th>Variables and subgoal</th></tr>
<tr><td>append(L1,L2,[a,b])</td><td>Rule 1</td><td>L1→[], L2→X→[a,b], X→[a,b] succeeds</td></tr>
<tr><td></td><td>Rule 2</td><td>L1→[H|T]→[a|T], L2→P, H→a, Q→[b]
Subgoal: append(T,P,[b]).</td></tr>
<tr><td>append(T,P,[b])</td><td>Rule 1</td><td>T→[], P→X1→[b], X1→[b] so gives L1→[a|T]→[a], L2→P→[b] in the original goal</td></tr>
<tr><td></td><td>Rule 2</td><td>T→[H1|T1]→[b|T1], P→P1, H1→b, Q1→[]
Subgoal: append(T1,P1,[]).</td></tr>
<tr><td>append(T1,P1,[]).</td><td>Rule 1</td><td>T1→[], P1→X2→[], X2→[] so gives T→[b|T1]→[b|[]]→[b], P→P1→[],
so gives L1→[a|T]→[a|[b]]→[a,b], L2→P→[] in the original goal</td></tr>
</table>

Recursive Programming Examples - reverse

- Define `reverse(X,Y)` to be true iff `X` (a list) is the reverse of the list `Y`
- Define a helper predicate: `rev(P,Q,R)` iff `append(reverse(P),Q)` gives `R`
- The helper predicate may seem more complicated than `reverse` itself, but in fact is simple and efficient
- `Q` serves as an accumulator of partially reversed list, `R` serves as final return value

Recursive Programming Examples - reverse

- In Prolog
  - `reverse(X,Y):-rev(X,[],Y).`
  - `rev([],R,R).`
  - `rev([H|T],Q,R):-rev(T,[H|Q],R).`
- In each recursion of `rev(P,Q,R)`, head of `P` is taken out and cons to `Q`, then it recurs until it reaches `rev([],Q',R)` in which case `Q'` is the answer we want
- `rev(P,Q,R)` can be used to append the reverse of `P` to `Q`

Recursive Programming Examples - reverse

- In Lisp
  - `(defun reverse (x) (rev x '()))`
  - `(defun rev (p q)`
    - `(cond`
      - `((null p) q)`
      - `(t (rev (cdr p) (cons (car p) q)))))`
- Again, the Prolog version is more flexible as `reverse` and `rev` can be used in both directions

Recursive Programming Examples - gcd

- Define `gcd(X,Y,Z)` to be true iff `Z` (integer) is the gcd (Greatest Common Divisor) of integers `X` and `Y`
- By Euclidean Algorithm, we have in Prolog
  - `gcd(X,0,X) :- X > 0.`
  - `gcd(X,Y,Z) :- X < Y, gcd(Y,X,Z).`
  - `gcd(X,Y,Z) :- X >= Y, Y > 0, P is X mod Y, gcd(Y,P,Z).`
- In Lisp (roughly)
  - `(defun gcd (x y)`
    - `(cond`
      - `((equal y 0) x)`
      - `((< x y) (gcd y x))`
      - `(t (gcd y (mod x y)))))`

Recursive Programming Examples – tree traversal

- Represent a binary tree using `t(Q,L,R)`, where `L` and `R` are its left and right child respectively, `Q` is the term on the node
- Leaves are `t(Q,nil,nil)`, empty subtree is represented as atom `nil`
- Now suppose we wish to traverse a binary tree in pre-order (root first, then left subtree, then right subtree) and print out the terms on the leaves separated by newlines

Recursive Programming Examples – tree traversal

- We use `pr_tree(T)` to represent our desired function as there is no output in this case, only the side effect is wanted
- In Prolog
  - `pr_tree(nil).`
  - `pr_tree(t(Q,L,R)) :- write(Q), nl, pr_tree(L), pr_tree(R).`
- Note that we rely on the left to right searching order of the underlying Prolog

Some Links

- Visit the QuickStart Languages web site
  http://actlab.csc.villanova.edu/quickstart
- Read up on logical languages
  http://en.wikipedia.org/wiki/Logic_programming
- Devote your life to this tutorial
  http://web.archive.org/web/20041028043137/http://cs.wwc.edu/~cs_dept/KU/PR/Prolog.html