Programming Language Concepts

CSCI-344 **Recitation 9**

Term 20161

Prolog Programming

1 Introduction

You will read, reason about, and write some simple clauses in the μProlog language.

2 Description

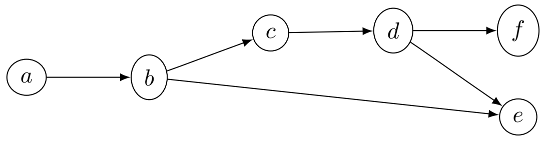
- Consider the following facts, which describe a collection of edges between nodes in a graph, and clauses, which purport to find paths between nodes in the graph:

  ```
  edge(a, b). edge(b, c). edge(c, d).
  edge(d, e). edge(b, e). edge(d, f).

  pathV1(X, Y) :- pathV1(X,Z), edge(Z, Y).
  pathV1(X, X).

  pathV2(X, X).
  pathV2(X, Y) :- edge(X, Z), pathV2(Z, Y).
  ```

  For convenience, here is pictoral representation of the graph:

  - Explain what happens when executing the query `pathV1(b,a)`.
  - Explain what happens when executing the query `pathV2(b,a)`.
  - Explain what happens when executing the query `pathV2(b,X)`.
  - Write a predicate `path` (of arity 3) such that `path(X,Y,L)` succeeds when there is a path from `X` to `Y` where `L` is the list of nodes along the path. For example:

    ```
    ?- path(b,f,L).
    L = [b, c, d, f].
    yes
    ?- path(b,b,L).
    L = [b].
    yes
    ?- path(b,a,L).
    no
    ```

- Write a predicate `notnull` (of arity 1) such that `notnull(L)` succeeds when `L` is a non-empty list.

  Here are some sample interactions with the `notnull` predicate:

  ```
  ?- notnull([]).
  no
  ?- notnull([a,b,c]).
  yes
  ?- notnull(L).
  L = [__17|__28];

  no
  ```

- Write a predicate `btreeMember` such that `btreeMember(X,BT)` succeeds when `X` is a member of the binary tree `BT`.

  To represent binary trees, we use the following functors:

  - `leaf` (a nullary functor): `leaf` represents the empty binary tree.
  - `node` (a functor of arity 3): `node(BTL,X,BTR)` represents the binary tree that has `BTL` as an immediate left sub-tree, has `X` as the element, and has `BTR` as an immediate right sub-tree.

  Here are some sample interactions with the `btreeMember` predicate:

  ```
  ?- btreeMember(33,
                 node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf)))).
  yes
  ?- btreeMember(42,
                 node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf)))).
  no
  ?- btreeMember(X,node(node(leaf,9,leaf),20,leaf)).
  X = 20;

  X = 9;

  no
  ```

- (Adapted from Exercise 22 of Chapter 11 from *Programming Languages: Build, Prove, and Compare* (p. 1052).)

  The primitive predicate `print` always succeeds and prints a term, but does nothing during backtracking. Write a predicate `backprint` that always succeeds and does nothing, but prints a term during backtracking. Perhaps surprisingly, `backprint` does not need to be a primitive predicate; you can write it in Prolog. Together, `print` and `backprint` make a crude tracing mechanism:

  ```
  ?- member(X, [1,2,3]), print(trying(x, X)), backprint(failed(x, X)),
     member(Y, [3,2,1]), print(trying(x, X, y, Y)), backprint(failed(y, Y)),
     X > Y.
  trying(x, 1)
  trying(x, 1, y, 3)
  failed(y, 3)
  trying(x, 1, y, 2)
  failed(y, 2)
  trying(x, 1, y, 1)
  failed(y, 1)
  failed(x, 1)
  trying(x, 2)
  trying(x, 2, y, 3)
  failed(y, 3)
  trying(x, 2, y, 2)
  failed(y, 2)
  trying(x, 2, y, 1)
  X = 2
  Y = 1.
  yes
  ```

- (From Programming 07: Prolog Programming) Write a predicate `swizzle` (of arity 3) such that `swizzle(L1,L2,L3)` succeeds when `L3` is a list with the first element of the list `L1` as the first element, the first element of the list `L2` as the second element, the second element of the list `L1` as the third element, the second element of the list `L2` as the fourth element, and so on. If the lists `L1` and `L2` are of unequal lengths, then the list `L3` concludes with the excess elements from the tail of the longer one.

  Here are some sample interactions with the `swizzle` predicate:

  ```
  ?- swizzle([1,2,3],[a,b,c],L).
  L = [1, a, 2, b, 3, c];

  no
  ?- swizzle([1,2,3],[a,b,c,d,e,f],L).
  L = [1, a, 2, b, 3, c, d, e, f];

  no
  ?- swizzle(L1,L2,[a,b,c,d,e,f]).
  L1 = []
  L2 = [a, b, c, d, e, f];

  L1 = [a, b, c, d, e, f]
  L2 = [];

  L1 = [a]
  L2 = [b, c, d, e, f];

  L1 = [a, c, d, e, f]
  L2 = [b];

  L1 = [a, c]
  L2 = [b, d, e, f];

  L1 = [a, c, e, f]
  L2 = [b, d];

  L1 = [a, c, e]
  L2 = [b, d, f];

  no
  ```

3 Requirements and Submission

You may use the reference interpreter (see Appendix A), but there may only be one active laptop in each group.

At the end of class, submit the group's solutions as hard-copy; be sure to include the names of all group members in the submission.

A Interpreter

A reference μProlog interpreter is available on the CS Department Linux systems (e.g., `glados.cs.rit.edu` and `queeg.cs.rit.edu` and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/uprolog
```

Use the reference interpreter to check your code.

B μProlog List Predicates

```
list(nil).
list(cons(H,T)) :- list(T).

head([H|T], H).
tail([H|T], T).

last([X], X).
last([H|T], X) :- last(T, X).

length([], 0).
length([H|T], N) :- length(T, M), N is M + 1.

append([], YS, YS).
append([X|XS], YS, [X|ZS]) :- append(XS, YS, ZS).

member(X, [X|T]).
member(X, [H|T]) :- member(X, T).

member_via_append(X,L) :- append(_, [X|_], L).

snoc([], X, [X]).
snoc([H|T], X, [H|T_X]) :- snoc(T, X, T_X).

reverseA([], []).
reverseA([H|T], TR_H) :- reverseA(T, TR), snoc(TR, H, TR_H).

reverseB([], []).
reverseB([H|T], LR) :- reverseB(T,TR), append(TR,[H],LR).

revappendC([], L, L).
revappendC([H|T], L2, L3) :- revappendC(T, [H|L2], L3).
reverseC(L, LR) :- revappendC(L, [], LR).

reverse(L, LR) :- reverseA(L, LR).

palindrome(L) :- reverse(L, L).

zip([], YS, []).
zip(XS, [], []).
zip([X|XS], [Y|YS], [pair(X,Y)|ZS]) :- zip(XS, YS, ZS).

permutation([], []).
permutation(L, [H|T]) :- append(XS, [H|YS], L), append (XS, YS, ZS), permutation(ZS, T).

ordered([]).
ordered([A]).
ordered([A,B|L]) :- A =< B, ordered([B|L]).

naive_sort(L,SL) :- permutation(L,SL), ordered(SL).

partition(Pivot, [A|XS], [A|YS], ZS) :- A =< Pivot, partition(Pivot, XS, YS, ZS).
partition(Pivot, [A|XS], YS, [A|ZS]) :- Pivot < A,  partition(Pivot, XS, YS, ZS).
partition(Pivot, [], [], []).

quicksort([], []).
quicksort([X|XS], SL) :-
  partition(X, XS, Lows, Highs),
  quicksort(Lows, SLows), quicksort(Highs, SHighs),
  append(SLows, [X|SHighs], SL).
```

C Building Puzzle

```
; Baker, Cooper, Fletcher, Miller, and Smith live in a five-story
; building. Baker doesn't live on the 5th floor and Cooper doesn't
; live on the first.  Fletcher doesn't live on the top or bottom
; floor, and he is not on a floor adjacent to Smith or Cooper. Miller
; lives on some floor above Cooper.
;
; Who lives on what floors?
[clause].

puzzle_soln(BLDG) :-
    empty_building(BLDG),
    location(baker,BN,BLDG),
    location(cooper,CN,BLDG),
    location(fletcher,FN,BLDG),
    location(miller,MN,BLDG),
    location(smith,SN,BLDG),
    floor_neq(BN, fifth),
    floor_neq(CN, first),
    floor_neq(FN, fifth), floor_neq(FN, first),
    floor_nadj(FN, SN),
    floor_nadj(FN, CN),
    floor_gt(MN, CN).

empty_building(building(_,_,_,_,_)).

location(P,first,building(P,_,_,_,_)).
location(P,second,building(_,P,_,_,_)).
location(P,third,building(_,_,P,_,_)).
location(P,fourth,building(_,_,_,P,_)).
location(P,fifth,building(_,_,_,_,P)).

append([], L2, L2).
append([H1|T1], L2, [H1|L3]) :- append(T1,L2,L3).
member(X, [X|T]).
member(X, [H|T]) :- member(X, T).

eqInList(X, X, L) :- member(X, L).
neqInList(X, Y, L) :- append(L1, [X|L2], L), member(Y, L1).
neqInList(X, Y, L) :- append(L1, [X|L2], L), member(Y, L2).
adjInList(X, Y, L) :- append(L1, [X,Y|L2], L).
nadjInList(X, Y, L) :- append(L1, [Z,X|L2], L), member(Y, L1).
nadjInList(X, Y, L) :- append(L1, [X,Z|L2], L), member(Y, L2).
ltInList(X, Y, L) :- append(L1, [X|L2], L), member(Y, L2).
gtInList(X, Y, L) :- append(L1, [X|L2], L), member(Y, L1).

floors([first,second,third,fourth,fifth]).
floor_eq(F1, F2) :- floors(FS), eqInList(F1, F2, FS).
floor_neq(F1, F2) :- floors(FS), neqInList(F1, F2, FS).
floor_adj(F1, F2) :- floors(FS), adjInList(F1, F2, FS).
floor_nadj(F1, F2) :- floors(FS), nadjInList(F1, F2, FS).
floor_gt(F1, F2) :- floors(FS), gtInList(F1, F2, FS).
```

Solutions

- path predicates
  - The query `pathV2(b,a)` goes into an infinite loop.
  - The query `pathV2(b,a)` fails (but terminates).
  - The query `pathV2(b,X)` succeeds with `X = b` (and continues to succeed with `X = c`, `X = d`, `X = e`, `X = f`, and `X = f`, before failing).
  - `path`

    ```
    pathAux(X, X, []).
    pathAux(X, Y, [Z|L]) :- edge(X, Z), pathAux(Z, Y, L).
    path(X, Y, [X|L]) :- pathAux(X, Y, L).
    ```

- `notnull`

  ```
  notnull([_|_]).
  ```

- `btreeMember`

  ```
  btreeMember(X, node(_, X, _)).
  btreeMember(X, node(BT, _, _)) :- btreeMember(X, BT).
  btreeMember(X, node(_, _, BT)) :- btreeMember(X, BT).
  ```

- `backprint`

  ```
  backprint(X).
  backprint(X) :- print(X), fail.
  ```

  Note that the following is an incorrect definition of `backprint`:

  ```
  backprint(X).
  backprint(X) :- print(X).
  ```

  Without the (always failing) predicate `fail` in the second clause, after backtracking to `backprint` and choosing the second clause, the interpreter will (redundantly) re-execute the entire computation that led to the failure and backtracking to `backprint`:

  ```
  ?- member(X, [1,2,3]), print(trying(x, X)), backprint(failed(x, X)),
     member(Y, [3,2,1]), print(trying(x, X, y, Y)), backprint(failed(y, Y)),
     X > Y.
  trying(x, 1)
  trying(x, 1, y, 3)
  failed(y, 3)
  trying(x, 1, y, 2)
  failed(y, 2)
  trying(x, 1, y, 1)
  failed(y, 1)
  failed(x, 1)
  trying(x, 1, y, 3)
  failed(y, 3)
  trying(x, 1, y, 2)
  failed(y, 2)
  trying(x, 1, y, 1)
  failed(y, 1)
  trying(x, 2)
  trying(x, 2, y, 3)
  failed(y, 3)
  trying(x, 2, y, 2)
  failed(y, 2)
  trying(x, 2, y, 1)
  X = 2
  Y = 1.
  yes
  ```

  Note the second printings of `trying(x, 1, y, 3)`, `trying(x, 1, y, 2)`, and `trying(x, 1, y, 1)`.