Programming Language Concepts

CSCI-344 **Programming 7**
Term 20235

Prolog Programming

1 Introduction

In this programming assignment, you will implement a number of predicates in μProlog in order to gain familiarity with the language and to practice logic programming.

Download `prog07.P` and `prog07_tests.P`. The former is a template for your submission and also includes a number of supporting rules. The latter is a test suite for the assignment.

2 Description

Complete the following problems. Each one is to define one or more μProlog predicates. See Requirements and Submissions for important restrictions.

A. (5pts) Write a predicate `sum` (of arity 2) such that `sum(L,N)` succeeds when the number `N` is the sum of the elements in list `L`. (The sum of the empty list is 0.) The first argument may be assumed to always be instantiated with a ground term when `sum` is used in a query. Thus, `sum` behaves like a function from its first argument to its second argument.

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

```
?- sum([1,2,3,4,5], N).
N = 15;

no
?- sum([], N).
N = 0;

no
```

B. (5pts) Write a predicate `prod` (of arity 2) such that `prod(L,N)` succeeds when the number `N` is the product of the elements in the list `L`. (The product of the empty list is 1.) The first argument may be assumed to always be instantiated with a ground term when `prod` is used in a query. Thus, `prod` behaves like a function from its first argument to its second argument.

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

```
?- prod([1,2,3,4,5], N).
N = 120;

no
?- prod([], N).
N = 1;

no
```

C. (5pts) Write a predicate `avg` (of arity 2) such that `avg(L,N)` succeeds when the number `N` is the average of the elements in the list `L`. (The predicate `avg` should be unsatisfiable when the first argument is the empty list.) The first argument may be assumed to always be instantiated with a ground term when `avg` is used in a query. Thus, `avg` behaves like a function from its first argument to its second argument.

Note: Since μProlog does not support non-integer arithmetic, the number `N` is technically the floor of the arithmetic mean of the elements of the list `L`.

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

```
?- avg([1,2,3,4,5], N).
N = 3;

no
?- avg([], N).
no
```

D. (5pts) 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
```

E. (5pts) Write a predicate `partition` (of arity 2) such that `partition(L, P)` is satisfied when the list of lists `P` is a partitioning of the list `L`. (A partitioning of a list L is a list of (non-empty) lists such that the concatenation of the lists of lists is the list L).

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

```
?- partition([1,2,3,4], P).
P = [[1], [2], [3], [4]];

P = [[1], [2], [3, 4]];

P = [[1], [2, 3], [4]];

P = [[1], [2, 3, 4]];

P = [[1, 2], [3], [4]];

P = [[1, 2], [3, 4]];

P = [[1, 2, 3], [4]];

P = [[1, 2, 3, 4]];

no
?- partition([], P).
P = [];

no
?- partition(L, [[1],[2],[3,4,5]]).
L = [1, 2, 3, 4, 5];

no
?- partition(L, [[1,2],[],[3,4,5]]).
no
```

Hint: Note that a list can have multiple partitions, but a partition corresponds to at most one list. Thus, `partition` behaves like a (partial) function from its second argument to its first argument. Organize the logic for `partition` to exploit this.

F. (bonus 5pts) Write a predicate `balanced_partition` (of arity 2) such that `balanced_partition(L, P)` is satisfied when the list of lists `P` is a balanced partitioning of the list `L`. (A partitioning of a list L is a balanced partitioning if all of the lists in the partitioning differ in length by no more than one.)

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

```
?- balanced_partition([1,2,3,4,5],P).
P = [[1], [2], [3], [4], [5]];

P = [[1], [2], [3], [4, 5]];

P = [[1], [2], [3, 4], [5]];

P = [[1], [2, 3], [4], [5]];

P = [[1], [2, 3], [4, 5]];

P = [[1, 2], [3], [4], [5]];

P = [[1, 2], [3], [4, 5]];

P = [[1, 2], [3, 4], [5]];

P = [[1, 2], [3, 4, 5]];

P = [[1, 2, 3], [4, 5]];

P = [[1, 2, 3, 4, 5]];

no
?- balanced_partition([],P).
P = [];

no
?- balanced_partition(L, [[1],[2],[3,4,5]]).
no
```

G. (10pts) Complete Exercise 16 of Appendix D from *Programming Languages: Build, Prove, Compare (Supplement)* (p. S103), except that the predicate should be named `msort` (rather than `msorted`). The first argument may be assumed to always be instantiated with a ground term when `msort` is used in a query. Thus, `msort` behaves like a function from its first argument to its second argument.

Hint: You will need to introduce auxiliary predicates to implement splitting the list and merging two sorted lists.

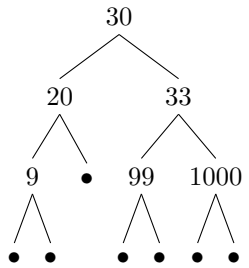
Hint: `msort` requires two base cases: the empty list and the singleton list (because splitting a singleton list would result in an empty list and a singleton list). And remember to make the rules for the base cases and the non-base case mutually exclusive (so that backtracking after using a base case cannot continue and use the non-base case).

The following three problems investigates writing Prolog predicates that manipulate binary trees.

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.

For example, the binary tree:



is represented by the structure

```
node(node(node(leaf,9,leaf),20,leaf),
     30,
     node(node(leaf,99,leaf),33,node(leaf,1000,leaf)))
```

H. (5pts) Write a predicate `btreeHeight` such that `btreeHeight(BT,N)` succeeds when the number `N` is the height of the binary tree `BT`. A leaf binary tree has height zero, while a node binary tree has height one greater than the maximum of the heights of its immediate sub-trees. The first argument may be assumed to always be instantiated with a ground term when `btreeHeight` is used in a query. Thus, `btreeHeight` behaves like a function from its first argument to its second argument.

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

```
?- btreeHeight(leaf,N).
N = 0;

no
?- btreeHeight(node(leaf,30,leaf),N).
N = 1;

no
?- btreeHeight(node(node(node(leaf,9,leaf),20,leaf),
                    30,
                    node(node(leaf,99,leaf),33,node(leaf,1000,leaf))),
               N).
N = 3;

no
```

I. (5pts) Write a predicate `btreeHighest` such that `btreeHighest(BT,X)` succeeds when `X` is an element of the binary tree `BT` that occurs at maximal height. The first argument may be assumed to always be instantiated with a ground term when `btreeHighest` is used in a query. Thus, `btreeHighest` behaves like a function from its first argument to its second argument.

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

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

X = 99;

X = 1000;

no
```

J. (5pts) Write a predicate `btreeInternal` such that `btreeInternal(BT,IBT)` succeeds when the binary tree `IBT` is an *internal tree* of the binary tree `BT`. Either of the arguments may be instantiated with a variable when `btreeInternal` is used in a query.

An internal tree IBT of a binary tree T is itself a (possibly empty) binary tree that is a contiguous set of elements from the binary tree T. Visually, one may "see" the internal tree IBT in the binary tree T, in the sense that the nodes of IBT correspond to the nodes of T, except that where IBT has leaves, T may have nodes.

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

```
?- btreeInternal(node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf))),
                     IBT).
IBT = leaf;

IBT = node(leaf, 30, leaf);

IBT = node(leaf, 30, node(leaf, 33, leaf));

IBT = node(leaf, 30, node(leaf, 33, node(leaf, 1000, leaf)));

IBT = node(leaf, 30, node(node(leaf, 99, leaf), 33, leaf));

IBT = node(leaf, 30, node(node(leaf, 99, leaf), 33, node(leaf, 1000, leaf)));

IBT = node(node(leaf, 20, leaf), 30, leaf);

IBT = node(node(leaf, 20, leaf), 30, node(leaf, 33, leaf));

IBT = node(node(leaf, 20, leaf), 30, node(leaf, 33, node(leaf, 1000, leaf)));

IBT = node(node(leaf, 20, leaf), 30, node(node(leaf, 99, leaf), 33, leaf));

IBT = node(node(leaf, 20, leaf),
           30,
           node(node(leaf, 99, leaf), 33, node(leaf, 1000, leaf)));

IBT = node(node(node(leaf, 9, leaf), 20, leaf), 30, leaf);

IBT = node(node(node(leaf, 9, leaf), 20, leaf), 30, node(leaf, 33, leaf));

IBT = node(node(node(leaf, 9, leaf), 20, leaf),
           30,
           node(leaf, 33, node(leaf, 1000, leaf)));

IBT = node(node(node(leaf, 9, leaf), 20, leaf),
           30,
           node(node(leaf, 99, leaf), 33, leaf));

IBT = node(node(node(leaf, 9, leaf), 20, leaf),
           30,
           node(node(leaf, 99, leaf), 33, node(leaf, 1000, leaf)));

IBT = leaf;

IBT = node(leaf, 20, leaf);

IBT = node(node(leaf, 9, leaf), 20, leaf);

IBT = leaf;

IBT = node(leaf, 9, leaf);

IBT = leaf;
```

```
IBT = leaf;

IBT = leaf;

IBT = leaf;

IBT = node(leaf, 33, leaf);

IBT = node(leaf, 33, node(leaf, 1000, leaf));

IBT = node(node(leaf, 99, leaf), 33, leaf);

IBT = node(node(leaf, 99, leaf), 33, node(leaf, 1000, leaf));

IBT = leaf;

IBT = node(leaf, 99, leaf);

IBT = leaf;

IBT = leaf;

IBT = leaf;

IBT = node(leaf, 1000, leaf);

IBT = leaf;

IBT = leaf;

no
?- btreeInternal(node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf))),
                 node(leaf,9,leaf)).
yes
?- btreeInternal(node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf))),
                 node(leaf,30,node(leaf,33,leaf))).
yes
?- btreeInternal(node(node(node(leaf,9,leaf),20,leaf),
                      30,
                      node(node(leaf,99,leaf),33,node(leaf,1000,leaf))),
                 node(leaf,30,node(leaf,99,leaf))).
no
```

Note that the queries like `btreeInternal(node(leaf,33,node(leaf,44,node(leaf,55,leaf))),IBT).` are effectively enumerations of the binary trees that are internal trees of the given binary tree. Note that in the interaction above, the Prolog interpreter may give the same internal tree more than once, because internal trees may be distinguished by their path through the tree, and, thus, may be considered "different" internal trees (that happen to be structurally equal).

Hint: `btreeInternal` requires one or more helper recursive predicates.

Hint: An *internal tree* is to a binary tree as a *sublist* is to a list. (A sublist SL of a list L is itself a (possibly empty) list that is a contiguous set of elements from the list L.) Review the implementations of the `sublist` predicate.

K. (15pts) This problem investigates writing Prolog predicates that solves a logic puzzle.

Consider the following logic puzzle:

> Alex, Bret, Chris, Derek, Eddie, Fred, Greg, Harold, and John are nine students who live in a three storey building, with three rooms on each floor. A room in the West wing, one in the centre, and one in the East wing. If you look directly at the building, the left side is West and the right side is East. Each student is assigned exactly one room.
>
> (a) Harold does not live on the bottom floor.
> (b) Fred lives directly above John and directly next to Bret (who lives in the West wing).
> (c) Eddie lives in the East wing and one floor higher than Fred.
> (d) Derek lives directly above Fred.
> (e) Greg lives directly above Chris.
>
> Can you find where each of their rooms is?
>
> `http://www.brainbashers.com/showpuzzles.asp?puzzle=ZQJZ`

To begin, we need to represent the students and the building as Prolog data structures. To do so, we use the following Prolog terms:

- `alex` (a nullary functor): `alex` represents the student Alex.
- ...
- `john` (a nullary functor): `john` represents the student John.
- `floor` (a functor of arity 3): `floor(SW,SC,SE)` represents a floor that has student `SW` living in the West wing, student `SC` living in the center, and student `SE` living in the East wing.
- `west` (a nullary functor): `west` represents the West wing room.
- `center` (a nullary functor): `center` represents the center room.
- `east` (a nullary functor): `east` represents the East wing room.
- `building` (a functor of arity 3): `building(FB,FM,FT)` represents a building that has floor `FB` as the bottom floor, floor `FM` as the middle floor, and floor `FT` as the top floor.
- `bottom` (a nullary functor): `bottom` represents the bottom floor.
- `middle` (a nullary functor): `middle` represents the middle floor.
- `top` (a nullary functor): `top` represents the top floor.

We can then assert that a student lives in a room on a floor of the building with the following predicates:

- `studentLivesInRoomOnFloor` (a predicate of arity 3): `studentLivesInRoomOnFloor(S,R,F)` succeeds when the student `S` lives in the room `R` on the floor `F`.
- `studentLivesInRoomOnFloorOfBldg` (a predicate of arity 4): `studentLivesInRoomOnFloorOfBldg(S,R,F,B)` succeeds when the student `S` lives in the room `R` on the floor `F` of the building `B`.
- `lives` (a predicate of arity 4): `lives` is an alias for `studentLivesInRoomOnFloorOfBldg`.

Write a predicate `puzzle_soln` (of arity 1) such that `puzzle_soln(BLDG)` is satisfied when the building `BLDG` is a solution to the logic puzzle.

Hint: Introduce auxiliary predicates to simplify the rule(s) for `puzzle_soln`.

Hint: To make the `puzzle_soln` predicate execute faster, interleave predicates that assert that a student lives in a room on a floor of the building with predicates that assert the additional conditions.

Note that this puzzle has exactly one solution.

The following two problems investigate writing Prolog predicates that manipulate regular expressions.

To begin, we need to represent a regular expression as a Prolog data structure. To do so, we use the following Prolog terms:

- `epsilon` (a nullary functor): `epsilon` represents the regular expression that matches the empty list.
- `char` (a functor of arity 1): `char(A)` represents the regular expression that matches the singleton list containing the atom `A`.
- `seq` (a functor of arity 2): `seq(RE1,RE2)` represents the regular expression that matches any list that can split into two lists (such that appending the two lists yields the original list) where the regular expression `RE1` matches the first list and the regular expression `RE2` matches the second list.
- `alt` (a functor of arity 2): `alt(RE1,RE2)` represents the regular expression that matches any list where the regular expression `RE1` matches the list or the regular expression `RE2` matches the list.
- `star` (a functor of arity 1): `star(RE)` represents the regular expression that matches the empty list and matches any list that can be split into one or more lists (such that concatenating the lists yields the original list) where the regular expression `RE` matches each list.

Since these terms represent data structures, you should not give predicates/clauses for them; rather, you will give rules for predicates that use these terms in their parameters.

L. (15pts) Write a predicate `re_match` (of arity 2) such that `re_match(RE,L)` succeeds when the regular expression `RE` matches the list of atoms `L`. The first argument may be assumed to always be instantiated with a ground term when `re_match` is used in a query; the second argument will likely (but need not) be instantiated with a ground term when `re_match` is used in a query.

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

```
?- re_match(alt(char(a),star(char(b))),[]).
yes
?- re_match(alt(char(a),star(char(b))),[a]).
yes
?- re_match(alt(char(a),star(char(b))),[a,b]).
no
?- re_match(alt(char(a),star(char(b))),[a,b,b]).
no
?- re_match(alt(char(a),star(char(b))),[b]).
yes
?- re_match(alt(char(a),star(char(b))),[b,b]).
yes
?- re_match(alt(char(a),star(char(b))),[b,b|Z]).
Z = [];

Z = [b];

Z = [b, b];

Z = [b, b, b];

Z = [b, b, b, b];

Z = [b, b, b, b, b].
yes
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[]).
no
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a]).
yes
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[b]).
no
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a,b]).
yes
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a,b,b]).
yes
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a,b,b,c]).
yes
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a,b,b,c,c]).
```

```
no
?- re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),[a,b|Z]).
Z = [c];

Z = [];

Z = [b, c];

Z = [b];

Z = [b, b, c];

Z = [b, b];

Z = [b, b, b, c];

Z = [b, b, b];

Z = [b, b, b, b, c].
yes
```

Note that the queries like `re_match(alt(char(a),star(char(b))),Z)` and `re_match(seq(char(a),seq(star(char(b)),alt(char(c),epsilon))),Z)` are effectively enumerations of the lists that are matched by the regular expression. However, the deterministic backtracking of Prolog means that the interpreter may attempt to enumerate an infinite set of lists before enumerating an alternative; consider the query `re_match(alt(star(char(a)),star(char(b))),Z)`, that produces the solutions `Z = []`, `Z = [a]`, `Z = [a,a]`, `Z = [a,a,a]`, ..., without producing any solution of the form `Z = [b, b, ..., b, b]`. To enumerate all lists up to a given length that match a predicate, use a query of the form `between(0,10,N), ofLength(N,Z), re_match(alt(star(char(a)),star(char(b))),Z).`.

The direct implementation of `re_match` (using `append` for the `seq` case) can be a little slow. One can do better with a predicate `re_match_aux(RE,L,LS)` that is satisfied when the regular expression `RE` matches a prefix of the list `L` and `LS` is the suffix of the `L` that is not matched by the regular expression `RE`.

M. (bonus 5pts) Write a predicate `re_reverse` (of arity 2) such that `re_reverse(RE,RRE)` succeeds when the regular expression `RRE` is the reversal of the regular expression `RE`. (That is, the regular expression `RE` matches a list of atoms iff the regular expression `RRE` matches the reversal of the list of atoms.)

3 Requirements and Submission

In addition to the specifications given in the problems, your functions must not use any non-logical features of μProlog; the use of `print`, `!` (cut), or `not` in any problem will result in zero credit for that problem.

Helper predicates may be defined.

Your submission must be a valid μProlog program. In particular, it must pass the following test:

```
$ cat prog07.P | /usr/local/pub/mtf/plc/bin/uprolog -q > /dev/null
```

without any error messages. If your submission produces error messages (e.g., syntax errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit `prog07.P` to the `Programming 07` Dropbox on MyCourses by the due date.

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.

A.1 Interactive mode

Simply executing

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

will run the interpreter interactively, but without line editing.

Executing

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

or

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

will run the interpreter interactively with line editing. (See the manual pages for `rlwrap` and `ledit` for more details.)

A.2 Batch mode

Executing

```
$ cat prog07.P | /usr/local/pub/mtf/plc/bin/uprolog
```

will run the interpreter on the contents of the file `prog07.P`, but with prompts printed.

Executing

```
$ cat prog07.P | /usr/local/pub/mtf/plc/bin/uprolog -q
```

will run the interpreter on the contents of the file `prog07.P` without prompts printed.

Executing

```
$ cat prog07.P prog07_tests.P | /usr/local/pub/mtf/plc/bin/uprolog -q
```

will run the interpreter on the contents of the files `prog07.P` and `prog07_tests.P` without prompts printed.