Worksheet 2: Recursive Data structures

Due: Midnight, Monday, 7 February 2010.
Total marks: 47

Instructions

If you detect or suspect any errors, post a question on the Worksheet 2 discussion forum for clarification, or email the instructor!

The Worksheet should be solvable in 10 to 12 hours of work. If after two hours (not including reading the text and notes), you have not made significant progress, waste no further time, and consult with the TA or the instructor.

Submit your solutions electronically using the CMPT 340 Moodle page for Worksheet 1. Please allow yourself some extra time to upload and submit your work, to account for using a system that may be new to you.

Question 1: More numeric recursion (12 marks)

Prolog has an arithmetic operator that performs exponentiation:

```
?- X is 3**4.
X = 81.0 ?
yes
```

We will pretend that it does not, just for fun and practice.

We will design a Prolog program that computes X^N where X is any number, and N is a non-negative integer. Something like this:

```
?- exp(3,4,X).
X = 81 ?
yes
```

Write 4 versions of `exp/3`:

1. A simple version, based on the factorial code we have seen in class, has time and space complexity $O(N)$.
2. An iterative ("tail recursive") version is $O(N)$ time and $O(1)$ space.
3. Write a version that is $O(\log N)$ time and space.

   Hint: to get $O(\log N)$ behaviour in any algorithm, you have to reduce the "size" of the problem roughly by two on most[1] steps. Positive integers are either odd or even. Even integers can be divided by two without any remainder. Use Prolog's integer division operator, which is used in the following example:

[1]The word "most" here is a bit misleading. It's enough that it's more than a constant number of times.

```
?- X is 14 // 3.
X = 4 ?
yes
```

4. Write a version that is $O(\log N)$ time and and $O(1)$ space by converting part 3 to use tail recursion.

Evaluation

Marking will be based on the following attributes:

- Correct implementation: 2 marks each. A mark will be deducted if the format of your program is unacceptable, even if it is correct. A mark will be deducted if there are rules or facts that are redundant or unnecessary.
- Documentation: 1 mark each. The documentation should include the informal contract, and a description of the variables and the relationship being defined.

What to hand in

Your implementation of the predicates above, in a file called `w2q1.pl`. Make sure that you put student information in the file, and document anything that you might think will help the marker read your program.

Question 2: Peano practice (9 marks)

In class we studied the so-called "Natural numbers": $0, s(0), s(s(0)), \cdots$ The purpose of this study is to see what they can teach us about Prolog. In this question we'll practice a bit more.

1. Implement a Prolog program for the relationship "greater than" for Natural numbers. It should have the following contract:

   ```
   % greaterThan(+X,+Y)
   % X > Y for Natural numbers X, Y
   ```

   Test your program thoroughly!

2. Implement imteger division for Natural numbers. It should have the following contract:

   ```
   % divide(+X,+Y,-Q,-R)
   % Q is the quotient of X / Y
   % R is the remainder
   ```

   Use the repeated subtraction technique from WS01, modified to use the Natural numbers data structure.

3. Implement `exp/3` for Natural numbers (see Question 1; which of the versions should you use for this one?).

Evaluation

Marking will be based on the following attributes:

- Correct implementation: 2 marks each. A mark will be deducted if the format of your program is unacceptable, even if it is correct.
- Documentation: 1 mark each. The documentation should include the informal contract, and a description of the variables and the relationship being defined.

What to hand in

Your implementation of the predicates above, in a file called `w2q2.pl`. Make sure that you put student information in the file, and document anything that you might think will help the marker read your program.

Question 3: Free Association (16 marks)

An *association list* is a list of pairs, which can be used to store and retrieve data. The implementation we will develop will not be especially efficient, but it will be a way to practice lists with a fairly practical example.

An example association list is `[(a,1),(b,2),(c,0)]`. The use of the parentheses cause Prolog to interpret this as a list of 3 items, each being a pair. We will call the left-hand subterm the *key*, and the right subterm the *value*.

1. (6 marks) The first operation we want is to look up an association. For example, we may want to know the value that is associated with key `a`.

   ```
   ?- lookup(a,[(a,1),(b,2),(c,0)],X).
   X = 1
   yes
   ```

   (a) (3 marks) Write `lookup/3`. Write a recursive program; don't use `member/2`. (your program will be very similar to `member/2`).

   (b) (3 marks) Rewrite `lookup/3`, this time using `member/2`; your program won't be recursive.

2. (3 marks) Given two lists of exactly the same length, create an association list that pairs corresponding elements from both lists. E.g.,

   ```
   ?- zip([a,b,c],[1,2,0],Assoc).
   Assoc = [(a,1),(b,2),(c,0)]

   yes
   ```

   If you've designed the program in the natural Prolog way, you can use your `zip/3` program to unzip an association list as well:

```
?- zip(L1, L2, [(a,1),(b,2),(c,0)]).
L1 = [a,b,c]
L2 = [1,2,0]

yes
```

You do not need to "design" this behaviour into it. Try it!

3. (3 marks) Another association list operation is to update a pair. For example, we may want to change the value associated with `a` to `3`:

   ```
   ?- update(c,3,[(a,1),(b,2),(c,0)],New).
   New = [(a,1),(b,2),(c,3)]

   yes
   ```

   `update/4` should add the new value at the end only if an association doesn't exist. E.g.

   ```
   ?- update(d,0,[(a,1),(b,2),(c,0)],New).
   New = [(a,1),(b,2),(c,0),(d,0)]

   yes
   ```

   (Yes, adding it at the end is sensible.)

Evaluation

Marking will be based on the following attributes:

- Correct implementation: 2 marks each. One mark will be deducted if the format of your program is unacceptable, even if it is correct.
- Documentation: 1 mark each. The documentation should include the informal contract, and a description of the variables and the relationship being defined.

What to hand in

Your implementation of the predicates above, in a file called `w2q3.pl`. Make sure that you put student information in the file, and document anything that you might think will help the marker read your program.

Be helpful to the marker by clearly indicating the code for each part.

Question 4: Binary Search Trees (10 marks)

This question concerns the use of recursive data structures, in the form of binary search trees. We will represent a binary search tree using the following:

- An empty binary tree will be represented by the atom `empty`.

- `tree(Key,Left,Right)` for non-empty trees (where `Key` is the search key, and `Left` and `Right` are binary search trees (subtrees)). [2]

Note that a leaf node in a tree has two empty binary trees as children, and that the variable `_` should not be used to represent an empty tree. Remember: `_` is a variable, and can represent *any* tree, not only the empty one.

We can write a program to check if a key is in a BST as follows:

```
% memberBST(+Key,+Tree)
% succeeds if the Key is in the Tree

memberBST(Key, tree(Key,_,_)).
memberBST(Key, tree(K2,L,_)) :-
  Key < K2,
  memberBST(Key,L).
memberBST(Key, tree(K2,_,R)) :-
  Key > K2,
  memberBST(Key,R).
```

In this implementation, we are assuming that all keys will be unique, so we don't have to worry about equality.

1. (5 marks) Implement `insertBST(+Tree1, +Key, -Tree2)` which relates a given `Tree1` (the original tree) and a given `Key` to the tree `Tree2` which is the result of inserting `Key` into `Tree1`. Note that `Tree1` does not change, but rather, we are building a slightly modified tree `Tree2`. This approach is very common in Prolog.

   As an example:

   ```
   % insert the key 2 into an empty tree:
   ?- insertBST(empty,2,T1).
   T1 = tree(2,empty,empty)
   yes
   % a chain of insertions: first 2, then 1
   ?- insertBST(empty,2,T1), insertBST(T1,1,T2).
   T1 = tree(2,empty,empty)
   T1 = tree(2,tree(1,empty,empty),empty)
   yes
   ```

   Be aware that Prolog does not "save" the trees from query to query; the second example above shows how to construct a conjunctive query to get sequential behaviour.

2. (5 marks) Write a program `removeBST(+Tree1, +Key, -Tree2)` that "removes" a given element from a given binary search tree. Note that the original tree is not changed; rather, a new tree is constructed with the given key gone. We will do this in steps.

   As an example:

[2] There are lots of other ways to represent BSTs. For an interesting alternative, look at PiP, Chapter 7. The code there will not be useful here, though.

```
?- removeBST(tree(2,tree(1,empty,empty),empty), 2, T).
T = tree(1,empty,empty)
yes
```

The removal of the key depends on where it is found. However, it will always be the root of some subtree in the whole tree. If we try to delete it, we'll be left with a tree that has a hole in it (which is not really allowed). We could fill the hole, though, with a key that has the binary search tree property: it must be bigger than anything on the left, and smaller than anything on the right. It may sound tricky, but you can find a key with exactly that property as a leaf of the left subtree. The following two steps implement this idea:

(a) (4 marks of 5) Write a program that, given a BST, removes the biggest key, and returns the BST containing all the remaining keys. In other words, `deleteMax(+Tree1,?Key,?Tree2)` is true when `Key` is the biggest key in `Tree1`, and `Tree2` contains the same keys as `Tree1` except that `Key` is missing. Procedurally, given `Tree1`, you can use this to return two things: a `Key` and the resulting `Tree2`.

Hint: the biggest key in a BST is found by stepping recursively into the right subtree. (Draw a picture of the situation! It's easier drawn than described.)

(b) (1 mark of 5) Write `removeBST(Tree1, Key, Tree2)` using `deleteMax(+Tree1,?Key,?Tree2)`. Once the key you are looking for (k) is located, you can replace k with the largest key k' in the tree that is smaller than k; this maintains the binary search tree property. You can find this key in the left subtree for k.

My descriptions above are sometimes procedural. Try to think in terms of relationships as well as in terms of procedure. Both aspects are helpful for different purposes. Here, we want to keep in mind the property of BSTs, namely that the keys are organized in a specific way. It will help!

Evaluation

Marking will be based on the following attributes:

- Correct implementation: 4 marks for each part. One mark will be deducted if the format of your program is unacceptable, even if it is correct.
- Documentation: 1 mark each. The documentation should include the informal contract, and a description of the variables and the relationship being defined.

What to hand in

Your implementation of the predicates above, in a file called `w2q4.pl`. Make sure that you put student information in the file, and document anything that you might think will help the marker read your program.

Be helpful to the marker by clearly indicating the code for each part.