German University in Cairo
Faculty of Media Engineering and Technology
Prof. Dr. Slim Abdennadher

June 14, 2009

Concepts of Programming Languages
Spring term 2009
Final Exam

Bar Code

Instructions: Read carefully before proceeding.

1) No books or other aids are permitted for this test.
2) This exam booklet contains 14 pages, including this one. Three extra sheets of scratch paper are attached and have to be kept attached. **Note that if one or more pages are missing, you will lose their points. Thus, you must check that your exam booklet is complete**.
3) Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem or on the three extra sheets and make an arrow indicating that. **Scratch sheets will not be graded unless an arrow on the problem page indicates that the solution extends to the scratch sheets**.
4) **Duration of the exam:** 3 hours
5) When you are told that time is up, stop working on the test.

Good Luck!

Don't write anything below `;-)`

Exercise	1	2	3	4	5	6	7	8	$\sum$
Possible Marks	6	14	12	14	15	6	8	18	93
Final Marks									

Exercise 1 (6 Marks)

In the slides, we defined the predicate connected, that determined whether a pair of nodes in a graph were connected via one or more edges. Define a predicate `path_connects` that keeps track of the nodes visited while establishing a path between two nodes. Your solution should not refer to `connected`.

Given:

```
edge(a,b).
edge(b,c).
edge(b,d).
edge(c,e).
edge(d,e).
edge(e,f).
edge(g,h).
edge(f,a).  % This edge makes the graph cyclic
```

Your predicate should behave as follows:

```
?- path_connects(a,c,P).
P = [a, b, c] ;
P = [a, b, c, e, f, a, b, c] ;
P = [a, b, c, e, f, a, b, c, e|...] ;
P = [a, b, c, e, f, a, b, c, e|...] ;
P = [a, b, c, e, f, a, b, c, e|...] .

?- path_connects(a,d,[a,b,d]).
true ;
false.

?- path_connects(a,g,P).
ERROR: Out of local stack
```

Solution:

```
path_connects(X,Y,[X,Y]) :- edge(X,Y).

path_connects(X,Y,[X|Path]) :-
        edge(X,N),
        path_connects(N,Y,Path).
```

Exercise 2 (6 + 4 + 4 = 14 Marks)

a) Write a predicate `difference(NumList, Diff)` that, under the assumption that `NumList` is a list of numbers without duplicates, is true if `Diff` is the difference between two different numbers that occur in `NumList`. Example: `difference([1,12,4,9], D)` will (in some order) answer that D can be `3, 8, 11, 5, 8, 3`.

Solution:

```
member(A, [A|_]).
member(A, [_|As]) :- member(A, As).

difference(List, D) :-
    member(E1, List),
    member(E2, List),
    E1 > E2,
    D is E1-E2.
```

b) Write a predicate `all_diffs(NumList, DiffList)` that returns, `DiffList`, a bag of (hint!) all differences between two numbers in `NumList`. Example: `all_diffs([1,12,4,9], [3,8,11,5,8,3])`. Remark: only one bag is returned, not all permutations of it.

Solution:

```
all_diffs(List, Diffs) :-
    bagof(D, difference(List,D), Diffs).
```

c) Write a predicate `all_diffs_differ(NumList)` that is true if the bag in part b does not contain any duplicates. Example: `all_diffs_differ([1,12,4,9])` is false, while `all_diffs_differ([0,1,11,4,9])` is true.

Solution:

```
all_diffs_differ(List) :-
    bagof(D, difference(List,D), DiffBag),
    setof(D, difference(List,D), DiffSet),
       % or: setof(D, member(D, DiffBag), DiffSet),
    length(DiffBag, L),
    length(DiffSet, L).
```

Exercise 3 (6+6 =12 Marks)

a) Write a function `search:: String -> char -> [Int]` that returns the position of all occurrences of the second argument in the first. For example

```
> search "Bookshop" 'o'
[1,2,6]
> search "senses" 's'
[0,3,5]
```

You may like to use a helper function in your definition but you should not use any predefined functions.

Solution:

```
searchRec :: String -> Char -> [Int]
searchRec str goal = searchAux str goal 0
    where
      searchAux [] goal i = []
      searchAux (x:xs) goal i | x == goal  = i : searchAux xs goal (i+1)
                              | otherwise  = searchAux xs goal (i+1)
```

b) Write a function `duplicate :: String -> String` that takes a list of characters and returns a list where each character is repeated a number of times corresponding to its position in the list: the first character appears once, the second twice, etc. For example

```
> duplicate "abcd"
"abbcccdddd"
```

Solution:

```
duplicate :: String -> String
duplicate xs  =  a 1 xs
  where
  a n []     =  []
  a n (x:xs) =  b x n ++ a (n+1) xs
    where
    b x 0      =  []
    b x (m+1)  =  x : b x m
```

Exercise 4 (4 + 4 + 6 = 14 Marks)

a) Write a recursive function `andRec :: [Bool] -> Bool` that checks whether every item in a list is true. Then, write the same function using a predefined higher order function, this time called `andHigher`.

Solution:

```
andRec :: [Bool] -> Bool
andRec []  =  True
andRec (x:xs)  =  x && andRec xs

andHigher :: [Bool] -> Bool
andHigher xs  =  foldr (&&) True xs
```

b) Write a recursive function `unequalsRec :: [(Int,Int)] -> [(Int,Int)]` that removes all pairs `(x,y)` where `x==y`. Then, write the same function using a predefined higher order function, this time called `unequalsHigher`.

Solution:

```
unequalsRec :: [(Int,Int)] -> [(Int,Int)]

unequalsRec [] = []
unequals ((x,y):xs) = if (x/=y) then (x,y):unequals xs
                      else unequals xs

unequalsHigher xys  =  filter unequal xys
    where unequal (x,y)  =  x /= y
```

c) Write a function `filterAll` that takes a list of functions and a list and returns the sublist of every element (from the second argument) from which all functions (from the first argument) evaluate to true. For example:

```
> filterAll [even, >0] [-1,-4,2,3,6]
[2,6]
```

Give the type of the function.

Solution:

1. Short solution (Many thanks to Rana Ashraf):

```
filterAll [] l = l
filterAll (f:fs) (x:xs) = filterAll fs (filter f (x:xs))
```

2. A more complicated solution (has procedural nature): The idea is to apply the filter on the list for all predicates and the result will be a list of lists. Then an intersection of these lists should be done:

```
filterAll :: [a -> Bool] -> [a] -> [a]
filterAll l1 l2 = intersectAll (filterAll2 l1 l2)

filterAll2 [] l = []

filterAll2 (f:fs) l = (filter2 f l):filterAll2 fs l

intersect s [] = []
intersect s (x:xs) | (member x s) = x:intersect s xs
                   | otherwise = intersect s xs
```

```
intersectAll [] = []
intersectAll [x,y] = intersect x y
intersectAll (x:y:xs) = intersectAll ((intersect x y):xs)

member x [] = False
member x (y:xs) = if (x==y) then True else member x xs
```

Exercise 5 (5+5+5=15 Marks)

Consider the following polymorphic datatype, representing binary trees where every node (including leaf nodes) are labeled:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

a) Recall the definition of the predefined function `zip` on two lists `xs` and `ys`, and returns a list which is as long as the shorter of the two input lists. Generalize the definition of `zip` to work on trees, with the output being another tree that matches as much as possible the shapes of the input trees. Give the type of the function.

For example given the following tree

```
        3
      /   \
     /     \
    2       4
   / \     / \
  /   \   6   2
 5     3

        6
      /   \
     /     \
    1       4
           / \
          1   3
```

The result of applying the `zipTree` method will be:

```
      (3,6)
      /   \
     /     \
  (2,1)   (4,4)
           / \
          /   \
       (6,1) (2,3)
```

Solution:

```
zipTree :: Tree a -> Tree b -> Tree (a,b)
zipTree (Leaf x) (Leaf y) = Leaf (x,y)
zipTree (Leaf x) (Node y _ _) = Leaf (x,y)
zipTree (Node y _ _) (Leaf x) = Leaf (x,y)
zipTree (Node x t1 t2) (Node y t3 t4) =
             Node (x,y) (zipTree t1 t3) (zipTree t2 t4)
```

b) The predefined function `map` works on lists. Generalize `map` to work on trees. Give the type of the function.

Solution:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leaf x) = Leaf (f x)
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)
```

c) Recall the definition of the predefined function `foldr` on lists:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e []  =  e
foldr f e (x : xs)  =  f x (foldr f e xs)
```

Generalize the definition of `foldr` to work on trees. **Hint:** The second argument of `foldrTree` should be a function. Give the type of the function.

Solution:

```
foldrTree :: (a -> b -> b -> b) -> (a -> b) -> Tree a -> b
foldrTree f g (Leaf x)  =  g x
foldrTree f g (Node x t1 t2)  =  f x (foldrTree f g t1) (foldrTree f g t2)
```

Exercise 6 (6 Marks)

This exercise is about the Haskell project.

Logic expressions are either propositional variables, e.g. p, q, or compound expressions. Your custom type must support the following Boolean expressions:

- $\neg p$ (negation)
- $p \wedge q$ (conjunction),
- $p \vee q$ (disjunction), and
- $p \rightarrow q$ (implication)

Given the following datatype to implement any logic expression.

```
data LogicExpr = Var Char | Not  LogicExpr | And LogicExpr  LogicExpr |
                Or  LogicExpr  LogicExpr | Imp LogicExpr  LogicExpr  deriving Show
```

You implemented in the project the following functions:

```
evaluate :: LogicExpr -> [(Variable,Bool)] -> Bool
generateTruthTable:: LogicExpr -> [[(Variable,Bool)]]
```

Assuming that they are working fine. Implement a function that checks whether a logic expression is a tautology. A logic expression is a *tautology* if it is true under any possible valuation.

For example

- $\neg p \vee p$ is a tautology
- $\neg p$ is not a tautology .

Solution:

The solution depends on the implementation of the `generateTruthTable` function. If the function will return the evaluation of the logic expression for each valuation, then the `tautology` function should just check that all these values correspond to `True`.

Exercise 7 (4+4=8 Marks)

a) Consider the following statements:

```
int x = 5;
int y = 3;
int *p, *q;
p = &x;
q = &y;
y = *p;
p = q;
*p *= *q;
```

Draw a diagram showing the values of `x, y, p`, and `q` after the execution of the above statements.

Solution:

```
x _____________          p_____________
 | 5            |          |             |
 |              |<-|-|-|--|-|             |
 |______________|         /|_____________|
                         /
                        /
                       /
y _____________        /    q _____________
 | /3/, /5/, 25|      /      |             |
 |             |<-----------|             |
 |_____________|             |_____________|
```

b) Suppose `x` is an array of integers, and we have just executed this code:

```
for(i=0;i<5;i++)
   x[i] = i*i;
```

Suppose that `x[0]` is stored at address 4530. What is the value of each of the following expressions? Justify your answer.

1. `x`

   Solution:

   `4530`

2. `&x[0]`

   Solution:

   `4530, same as x`

3. `*x`

   Solution:

   `0, same as x[0]`

4. `x[1]`

   Solution:

   `1`

5. `&x[1]`

   Solution:

   `4534 (four bytes from &x[0] )`

6. `x+2`

   Solution:

   `4538 (four more bytes)`

7. `*(x+2)`

   Solution:

   `4, same as x[2]`

8. `*(&x[2] +1)`

   Solution:

   `9, same as x[3], which is *(x+2+1).`

Exercise 8 (6+6+6=18 Marks)

Given the following fragment of C code to define a binary tree:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>


typedef struct TreeNode
{
        int data;
        TreeNode* left;
        TreeNode* right;
};

TreeNode* CreateTreeNode(int data)
{
          TreeNode* ptr = (TreeNode*) malloc ( sizeof(TreeNode));

          (*ptr).data = data ;

          (*ptr).left = NULL ;
          (*ptr).right = NULL ;

          return ptr;
}

void DeleteTreeNode(TreeNode* ptr)
{
     if( ptr == NULL) return;

     DeleteTreeNode( (*ptr).right);
     DeleteTreeNode( (*ptr).left);

     free( ptr ); //this is the actual line in which tree nodes are removed from memory

}

typedef struct Tree
{
        TreeNode* root;
};
```

a) Implement the following method

```
Tree* CreateTree()
```

that creates a binary search tree.

Solution:

```
Tree* CreateTree()
{
      Tree* ptr = (Tree*) malloc ( sizeof(Tree));

      (*ptr).root = NULL ;
```

```
      return ptr;
}
```

b) Implement the following method

```
void DeleteTree(Tree* treePtr)
```

to delete a tree.

Solution:

```
void DeleteTree( Tree* treePtr)
{
     DeleteTreeNode( (*treePtr).root);
     free(treePtr);
}
```

c) Implement the following method

```
 int GetTreeHeight(Tree* ptr)
```

that returns the height of a tree. A null tree is defined to have height -1. A tree with one level has a height 0 and so on. The method should be implemented recursively.

Solution:

```
int GetTreeHeightHelper(TreeNode* ptr)
{
     if ( ptr == NULL ) return -1;

     int rightHeight = GetTreeHeightHelper ( (*ptr).right);
     int leftHeight = GetTreeHeightHelper ( (*ptr).left );

    if ( rightHeight > leftHeight) return 1 + rightHeight;
    return 1 + leftHeight ;
 }

 int GetTreeHeight(Tree* ptr)
 {
     return GetTreeHeightHelper( (*ptr).root) ;
 }
```

Extra Sheet

Extra Sheet

Extra Sheet