COMP 302: Programming Languages and Paradigms

Winter 2015: Assignment 5

This assignment is due on Tuesday, April 14 at midnight. It must be submitted using MyCourses**.** The cutoff time is automated. Assignments submitted within the next hour will be accepted but marked late. The system will not accept any attempts to submit after that time.

Problem 1 (40 marks)

We use the following datatype to encode terms of a small SML-like language.

Implement a function `free-list : exp -> string list` which computes the set of free variables in an expression of this language.

```
datatype exp = Nat of int | Bool of bool |Plus of exp * exp |
               Mult of exp * exp | If of exp * exp * exp |
               And of exp * exp | Not of exp | Eq of exp * exp |
               Var of string | Let of exp * (string * exp) |
               Fun of string * string * exp |
               Apply of exp * exp
```

The term `let x = e1 in e2` is represented as `Let(e1,("x",e2))`. (The variable is nearer the expression that it is binding).
Variables are normally written using the Var constructor but in the binding occurrences we omit this constructor and just write the string.
For functions, `fun foo(n) = body` is represented as `Fun("foo","n",body).`
Here are some examples of terms written using the above datatype:

```
val ex1 = Let(Nat(5),("a",(Plus(Var("a"),Nat(2)))))
val ex2 = Plus(Var("a"),Nat(2))
val ex3 = Let(Var("y"),("x",Plus(Var("x"),Var("z"))))
val ex4 = Let(Nat(3),("z",(Let(Nat(2),("y",ex3)))))
val ex5 = Fun("fac","n",If(Eq(Var("n"),Nat(0)),Nat(1),
    Mult(Var("n"),(Apply(Var("fac"),(Mult(Var("n"),Nat(1))))))))
val ex6 = Fun("f","n",Plus(Nat(3),Var("n")))
val ex7 = Fun("g","n",If(Eq(Var("n"),Nat(0)),Nat(0),
Plus(Nat(1),Apply(Var("g"),(Minus(Var("n"),Nat(1)))))))
```

To solve the problem, you might want to write a program without worrying about having multiple copies of a variable, then use a function to remove duplicates from the list

Question 2:
The Catalan numbers can be defined by the recurrence relation

$$C_0 = 1, and\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

The first few Catalan numbers are:

```
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, …
```

Part 1: (10 marks)
Implement a function `Catalan:int -> int` which, when given an integer n, computes the n-th Catalan number. For full credit, use the higher order function foldl in your solution.

Part 2: (20 marks)
Define a stream of integers as follows:

```
datatype realSeq = Cons of real * (unit -> realSeq);
```

You are to compute the sequence of Catalan numbers using lazy streams. We will use real numbers since the sequence grows quickly and an implementation with integers will overflow.

Use the following methodology.

First define an infinite stream `helperSeq : int -> realSeq` where the i-th term in the sequence is $\frac{2(2i+1)}{i+2}$. You can use the SML function Real.fromInt to avoid type errors when doing the division.

Then use the helperSeq to compute the sequence of Catalan numbers using lazy evaluation.

Question 3 (30 marks)

You are to implement a memory cell. A cell supports three operations: Get, Put and Restore. The following data-type defines these instructions.

```
datatype ’a instr = Put of ’a | Get | Restore
```

Put(v) stores a value v and overwrites the old one. Get retrieves a value. In addition, we can restore the previous value using the instruction Restore. If we call Restore once, we get the most recent value Put in the cell. If we call restore twice, we get the value we had Put two times ago.

To enable us to do this, this we would like to keep a stack of values that were stored in the cell. This can be modelled using a list. If one calls restore more often than the number of values entered, you should raise an exception and output the message “Nothing to Restore”

Write an implementation of this as an object using closures and higher order functions. Cells are created with an initial value. The following illustrates an example of how cells are made and used.

```
- val cell0 = makeCell 0;
val cell0 = fn : int instr -> int
```

This will create a cell cell0 which is initialized with 0. Initially the stack holding previous values is empty. We can use the instructions Put, Get, and Restore to update the cell, get its current value and restore its previous value. For example:

```
- cell0 (Put(5));
val it = 5 : int
- cell0 (Put(8));
val it = 8 : int
- cell0 (Get);
val it = 8 : int
- cell0 (Restore);
val it = 5 : int
- cell0 (Restore);
val it = 0 : int
- cell0 (Restore);
uncaught exception Error
```