COMP 302: Programming Languages and Paradigms

Winter 2015: Assignment 3

This assignment is due on Wednesday, March 11, 2015 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:

You are to write two programs that produce a list of all values in a binary search tree whose keys satisfy a given property.

The data type for trees is:

```
datatype 'a tree = Empty | Node of (int * 'a) * 'a tree * 'a tree
```

The value stored at each node is a pair consisting of a key and an associated string. You can assume that the ordering of the values is that of a binary search tree.

The first program will use value carrying exceptions and the second continuations.

Problem 1.1 (20 marks)

Implement a function `collect` of type

```
collect: (int -> bool) -> string tree -> (int * string) list
```

`collect` expects a function `p : int -> bool` and a binary search tree, T. It returns a list of all pairs such that the key satisfies the predicate p.

You must use a value-carrying exception:

```
exception Found of (int * string) list
```

to accumulate all of the entries in the list. The final list should be in sorted order (sorted by keys). That is, you should do an inorder traversal of the tree.

Problem 1.2 (20 marks)

Implement a function `gather` of type

```
gather : (int -> bool) -> string tree ->

   ((int * string) list -> (int * string) list) (*continuation *)

   -> (int * string) list
```

The function gather p T cont = lst, takes a predicate p, a binary search tree, T and a continuation cont. Use the continuation to accumulate the result. The final list should be in sorted order.

Problem 2 (25 marks)

Consider the following two programs for computing n^k. The first one is a naive version and the second uses tail-recursion.

Prove that they return the same results if applied to the same arguments.

```
fun pow (n,0) = 1
  | pow (n,k) = n * pow(n, k-1)


fun pow_tl(n,0,acc) = acc
  | pow_tl(n,k,acc) = pow_tl(n,k-1,n*acc)
```

Before you begin your proof, state carefully the statement you want to prove, describe how your proof proceeds, and state your induction hypothesis.

Hint: Generalize the result you want to prove first. Think about how to generalize the statement to take account of changes to the accumulator.

Problem 3:

Church numerals provide a means of representing values as functions. The method is named for Alonzo Church who encoded data in the lambda calculus in this way. (He showed that this method of encoding data can be used to represent computable functions. The Church-Turing thesis asserts that any computable function can be represented by this Church encoding).

We define the type of a Church numeral as

```
type 'a church = ('a -> 'a) * 'a  -> 'a
```

We can think of a church numeral as a function applied n times. For example :

```
0   is represented as    fn (f,x) => x
1   is represented as    fn (f,x) => f(x)
2   is represented as    fn (f,x) => f(f(x))
```

Problem 3.1 (15 marks)

Write a function

```
create: int -> 'a church
```

which, given an integer argument, produces the corresponding church numeral.

Make sure that the resulting church numeral you return does not have any recursive calls to create.

Problem 3.2 (5 marks)

Write a function

```
churchToInt : 'a church -> int
```

which given a church numeral as its argument will produce the corresponding integer value.

Problem 3.3 (15 marks)

Write a function

```
SUCC : ’a church -> ’a church
```

which given a church numeral as its argument will produce the church numeral representing its successor.

Do not convert to an integer and then after incrementing it, convert it back. Your function must work directly with church numerals.