HW3

Due date: 12/12

Turn in your code for the starred (sub)problems.

1 [Type inference] (30%)

Infer the type, if any, of each λ-expression or recursive function below.

a) $\lambda f.\lambda a.\lambda b.\lambda c.c\ (f\ a)\ (f\ b)$

b) $\lambda f.\lambda a.\lambda b.b\ f\ (f\ a\ b)$

c) fix = $\lambda f.f\ (\lambda x.\text{fix}\ f\ x)$

2 [Equational reasoning] (45%)

The advocates of functional languages frequently say that the properties of functional programs can be proved mathematically.

For example, given the SML map function defined by

```
fun map f [] = []                       (map.1)
  | map f (x::xs) = f x :: map f xs     (map.2)
```

We may prove the following lemma by **structural induction**. (Remember it? Recall section 4.3 of *Discrete Mathematics and Its Applications*, 6th ed., Kenneth Rosen)

LEMMA map (f ∘ g) xs = map f (map g xs), for any functions f, g and list xs

Proof by structural induction on xs

Basis: xs = []

```
l.h.s = map (f ∘ g) []
      = []                        (map.1)
r.h.s = map f (map g [])
      = map f []                  (map.1)
      = []                        (map.1)
```

This establishes the base case.

Induction step: xs = y :: ys.

```
map (f ∘ g) (y :: ys)
= (f ∘ g) y :: map (f ∘ g) ys    (map.2)
= f (g y) :: map (f ∘ g) ys      (definition of ∘)
= f (g y) :: map f (map g ys)    (induction hypothesis)
= map f (g y :: map g ys)        (map.2)
= map f (map g (y :: ys))        (map.2)
```

This completes the proof.

In this problem, you are asked to prove some properties of foldr (fold right) and foldl (fold left) defined by:

```
fun foldr f a [] = a                          (foldr.1)
  | foldr f a (x::xs) = f(x,foldr f a xs);    (foldr.2)

fun foldl f a [] = a                          (foldl.1)
  | foldl f a (x::xs) = foldl f (f(a,x)) xs;  (foldl.2)
```

These two functions manipulate the elements of a list in different order. For example, foldr sums up the elements of a list from right to left:

foldr op+ 0 [1,2,3,4,5]
= op+(1,op+(2.op+(3,op+(4,op+(5,0)))))
= 1+(2+(3+(4+(5+0))))
= 15

But, foldl sums up the elements of a list from left to right:

foldl op+ 0 [1,2,3,4,5]
= op+(op+(op+(op+(op+(0,1),2),3),4),5)
= ((((0+1)+2)+3)+4)+5
= 15

Observe that f is a function taking a 2-tuple as an argument. For convenience, we shall use infix notation in the sequel. That is, let f = $\oplus$, we shall write x $\oplus$ y, instead of $\oplus$(x,y). [Example, let f = op+, we shall write x+y, instead of op+(x,y).]

a) Prove the following lemma: (10%)

   LEMMA Let $\oplus$ and $\otimes$ be two functions that satisfy
   x $\oplus$ (y $\otimes$ z) = (x $\oplus$ y) $\otimes$ z
   Then, y $\oplus$ (foldl $\otimes$ z xs) = foldl $\otimes$ (y $\oplus$ z) xs, for any list xs.

b) Use the lemma of part a) to prove the following theorem: (10%)

   THEOREM (The duality theorem)
   Let $\oplus$ and $\otimes$ be two functions that satisfy
   x $\oplus$ (y $\otimes$ z) = (x $\oplus$ y) $\otimes$ z and x $\oplus$ a = a $\otimes$ x,
   Then, for any list xs, foldr $\oplus$ a xs = foldl $\otimes$ a xs.

c) Use the theorem of part b) to prove the following corollary: (5%)

   COROLLARY
   If $\oplus$ is associative and a is the identity of $\oplus$, then for any list xs,
   foldr $\oplus$ a xs = foldl $\oplus$ a xs.

d) Define

val sumr = foldr op+ 0;

val suml = foldl op+ 0;

Use the corollary of part c) to prove that sumr = suml, i.e. for any list xs, sumr xs = suml xs. (5%)

e) Define

fun revr xs = foldr (fn (x,xs)=>xs@[x]) [] xs;

fun revl xs = foldl (fn (xs,x)=>x::xs) [] xs;

Use the theorem of part b) to prove that revr = revl, i.e. for any list xs, revr xs = revl xs. (10%)

f)* In fact, foldr and foldl are built-in SML functions.

However, the built-in foldl has a different definition.

```
fun foldl f a [] = a
  | foldl f a (x::xs) = foldl f (f(x,a)) xs;
```

In this definition, a is the 2nd argument (i.e. right operand) of f; whereas in ours, it is the 1st argument (i.e. left operand) of f.

With this definition, we have

foldl op+ 0 [1,2,3,4,5]

= op+(5,op+(4,op+(3,op+(2,op+(1,0)))))

= 5+(4+(3+(2+(1+0))))

= 15

Use this definition of foldl to define the function revl of part e). (5%)

3 [Higher-order function, Church numeral] (20%)

A natural number n may be represented by the higher-order function λf.λx.f^n x, called the nth Church numeral. For examples, 0, 1, 2, and 3 are represented by the Church numerals λf.λx.x, λf.λx.f x, λf.λx.f (f x) , λf.λx.f (f (f x)), respectively.

You are asked to write the following four SML functions, where $\bar{n}$ denotes the nth Church numeral.

a)* n2c converts a natural number to the corresponding Church numeral, i.e.

n2c $n \Rightarrow \bar{n}$

b)* c2n converts a Church numeral to the corresponding natural number, i.e.

c2n $\bar{n} \Rightarrow n$

c)* ++ is an infix operator for adding two Church numerals, i.e.

$\bar{m}$ ++ $\bar{n} \Rightarrow \overline{m+n}$

d)* ** is an infix operator for multiplying two Church numerals, i.e.

$\overline{m}$ ** $\overline{n}$ $\Rightarrow$ $\overline{mn}$

Requirements

1 Both ++ and ** are left associative. ** is of precedence level 7, and ++ is of precedence level 6.

2 Define $\overline{m}$ ++ $\overline{n}$ and $\overline{m}$ ** $\overline{n}$ directly. Do NOT convert $\overline{m}$ and $\overline{n}$ to m and n, respectively, and then convert $m+n$ back to $\overline{m+n}$

Sample run

– (c2n o n2c) 7;	– c2n (n2c 3**n2c 4++n2c 5);
val it = 7 : int	val it = 17 : int

4 [Concrete data type in ML] (35%)

A natural number may also be represented by a data structure.

Define

datatype Nat = Zero | Succ of Nat; (* Succ means Successor *)

Then, 0, 1, 2, and 3 are represented by

Zero, Succ Zero, Succ (Succ Zero), and Succ (Succ (Succ Zero)), respectively.

We shall call these representations as Nat numerals.

a) Draw a diagram showing the underlying data structure of the 3rd Nat numeral Succ (Succ (Succ Zero)).

Next, write the following SML functions, where $\overline{n}$ denotes the nth Nat numeral.

b)* n2N converts a natural number to the corresponding Nat numeral, i.e.

n2N n $\Rightarrow$ $\overline{n}$

c)* N2n converts a Nat numeral to the corresponding natural number, i.e.

N2n $\overline{n}$ $\Rightarrow$ n

d)* +++ is an infix operator for adding two Nat numerals, i.e.

$\overline{m}$ +++ $\overline{n}$ $\Rightarrow$ $\overline{m+n}$

e)* *** is an infix operator for multiplying two Nat numerals, i.e.

$\overline{m}$ *** $\overline{n}$ $\Rightarrow$ $\overline{mn}$

f)* !!! is a function for computing the factorial of a Nat numeral, i.e.

!!! $\overline{n}$ $\Rightarrow$ $\overline{n!}$

g)* ! is a function for computing the factorial of a natural number, i.e.

! n $\Rightarrow$ $n!$

This function shall convert n to $\overline{n}$, compute the factorial of $\overline{n}$ to obtain $\overline{n!}$, and then convert $\overline{n!}$ back to $n!$. Also, define it by function composition.

Requirements

1 Both +++ and *** are left associative. *** is of precedence level 7, and +++ is of precedence level 6.

2 Define $\overline{m}$ +++ $\overline{n}$ and $\overline{m}$ *** $\overline{n}$ directly. Do NOT convert $\overline{m}$ and $\overline{n}$ to m and n, respectively, and then convert $m+n$ back to $\overline{m+n}$

Sample run

– ! 7;

val it = 5040 : int

5* [Concrete data type in ML] (20%)

Given a string that represents a prefix arithmetic expression formed by single-digit numbers and operators +,-,*,/,and %, write a ML function

eval : string -> int

that uses an expression tree (see below) to compute the value of the given prefix expression. You may assume that the string represents a legal prefix expression.

Sample run

– eval "+2*34";	– eval "+*/83%-9246";
val it = 14 : int	val it = 12 : int

Requirements

1 First, use the built-in SML function *explode* to convert a string into a list of characters. For example,

– explode "+2*34";

val it = [#"+",#"2",#"*",#"3",#"4"] : char list

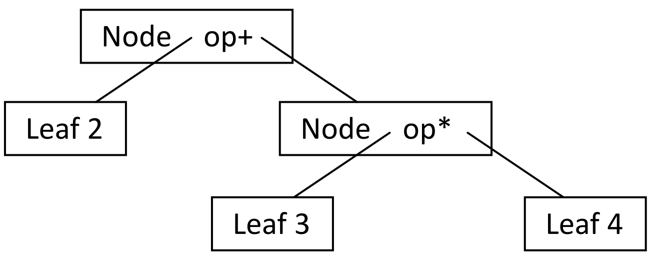
Now, a prefix expression is represented by a list of characters, which is then converted to an expression tree, as described below.

2 Define a concrete data type etree in SML as

datatype etree = Leaf of int | Node of etree*(int*int->int)*etree;

With this data type, the preceding prefix expression is represented as

Node (Leaf 2,op+,Node (Leaf 3,op*,Leaf 4)), as depicted below.



3 Define a function

mketree : char list -> etree * char list

that converts a list of characters xs to a 2-tuple (et, ys), where et is an expression tree corresponding to the initial sublist of xs that forms a prefix expression and ys is the remaining sublist of xs. (Note: It is assumed that the list xs isn't empty.)

For example,

– mketree [#"*",#"2",#"3",#"-",#"4",#"5"];

val it = (Node (Leaf 2,*fn*,Leaf 3),[#"-",#"4",#"5"]) : etree * char list

– mketree [#"2",#"*",#"3",#"4"];

val it = (Leaf 2,[#"*",#"3",#"4"]) : etree * char list

– mketree [#"+",#"2",#"*",#"3",#"4"];

val it = (Node (Leaf 2,*fn*,Node (Leaf 3,*fn*,Leaf 4)),[]) : etree * char list

where each blue-colored part is an initial prefix expression and *fn* denotes either op+ or op*.

4 Use the function mketree to define the function

prefix2etree : char list -> etree

For example,

– prefix2etree [#"+",#"2",#"*",#"3",#"4"];

val it = Node (Leaf 2,fn,Node (Leaf 3,fn,Leaf 4)) : etree

5 Define the function

inorder : etree -> int

to evaluate an expression tree by inorder traversal.

For example,

– inorder (prefix2etree [#"+",#"2",#"*",#"3",#"4"]);

val it = 14 : int

6 Finally, use all of the above to define the function

eval : string -> int

This function shall be defined by composition.

7 Finally, the datatype etree and the functions mketree, prefix2etree, and inorder shall all be local to the function eval.

6 Consider the following C++ program

```
#include <iostream>
using namespace std;
class X {
public:
    X(int n) : p(new int(n)) { cout << "X(" << *p << ") constructed\n"; }
    X(const X& rhs) : p(new int(*rhs.p)) { cout << "X(" << *p << ") copied\n"; }
    X(X&& rhs) : p(rhs.p) { rhs.p=nullptr; cout << "X(" << *p << ") moved\n"; }
    ~X()
    {
        if (p!=nullptr) {
            cout << "X(" << *p << ") destructed\n"; delete p;
        } else cout << "Moved X object: Nothing to destruct\n";
    }
private:
    int* p;
};
void q(int n) { if (n>0) throw X(n); }
int p(int n)
{
    try { q(n); }
    catch (int& c) { cout << c; }
    catch (X& c) { p(n-1); }
}
int main() { p(2); }
```

a) Show the output of the program. (5%)
Hint: Compile and run under C++11, e.g.
bsd2> g++47 –std=c++11 file.cpp

b) Draw a picture showing the contents of type_info table, catch table, run time stack, exception stack, and heap at the point when the exception object X(2) is constructed. (10%)

c) Repeat b), but this time at the point when function q is called with n=0. (10%)