Homework 2
COSE212, Fall 2017

Hakjoo Oh

Due: 10/31, 24:00

Academic Integrity / Assignment Policy

- *All assignments must be your own work.*
- Discussion with fellow students is encouraged including how to approach the problem. However, your code must be your own.
  - Discussion must be limited to general discussion and must not involve details of how to write code.
  - You must write your code by yourself and must not look at someone else's code (including ones on the web).
  - Do not allow other students to copy your code.
  - Do not post your code on the public web.
- Violating above rules gets you 0 points for the entire HW score.

Problem 1 (10pts) Binary trees can be defined as follows:

```
type btree = Empty | Node of int * btree * btree
```

For example, the following `t1` and `t2`

```
let t1 = Node(1,Empty,Empty)
let t2 = Node(1,Node(2,Node(3,Empty,Empty),Empty),Node(4,Empty,Empty))
```

are binary trees.

Write a function

```
mirror: btree -> btree
```

that exchanges the left and right subtrees all the ways down. For example,

```
mirror t1 = Node (1, Empty, Empty)
```

```
mirror t2 = Node(1,Node(4,Empty,Empty),Node(2,Empty,Node(3,Empty,Empty)))
```

Problem 2 (10pts) Natural numbers can be defined as follows:

```
type nat = ZERO | SUCC of nat
```

For instance, `SUCC ZERO` denotes 1 and `SUCC (SUCC ZERO)` denotes 2. Write three functions that add, multiply, exponentiate natural numbers:

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
natexp : nat -> nat -> nat
```

For example,

```
# let two = SUCC (SUCC ZERO);;
val two : nat = SUCC (SUCC ZERO)
# let three = SUCC (SUCC (SUCC ZERO));;
val three : nat = SUCC (SUCC (SUCC ZERO))
# natadd two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
# natmul two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
# natexp two three;;
- : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))))
```

Problem 3 (20pts) Consider the formulas of propositional logic:

$$
\begin{array}{rcll}
F & \rightarrow & \mathit{true} & \\
 & | & \mathit{false} & \\
 & | & P & \text{variables} \\
 & | & \neg F & \text{negation (``not'')} \\
 & | & F_1 \wedge F_2 & \text{conjunction (``and'')} \\
 & | & F_1 \vee F_2 & \text{disjunction (``or'')} \\
 & | & F_1 \rightarrow F_2 & \text{implication (``implies'')} \\
 & | & F_1 \leftrightarrow F_2 & \text{iff (``if and only if'')}
\end{array}
$$

which translates to the following type definition in OCaml:

```
type formula =
    True
  | False
  | Var of string
  | Neg of formula
  | And of formula * formula
  | Or of formula * formula
  | Imply of formula * formula
  | Iff of formula * formula
```

We say a formula F is *satisfiable* iff there exists a variable assignment that makes the formula true. For example, the formula $P \wedge \neg Q$ is satisfiable because

it evaluates to true when P is true and Q is false. The formula $P \wedge \neg P$ is not satisfiable since it always evaluates to false.

Write a function

```
sat: formula -> bool
```

that determines the satisfiability of a given formula. For example,

```
sat (And (Var "P", Neg (Var "Q")))
```

returns true.

Problem 4 (20pts) Write a function

```
diff : aexp * string -> aexp
```

that differentiates the given algebraic expression with respect to the variable given as the second argument. The algebraic expression `aexp` is defined as follows:

```
type aexp =
 | Const of int
 | Var of string
 | Power of string * int
 | Times of aexp list
 | Sum of aexp list
```

For example, $x^2 + 2x + 1$ is represented by

```
Sum [Power ("x", 2); Times [Const 2; Var "x"]; Const 1]
```

and differentiating it (w.r.t. "x") gives $2x + 2$, which can be represented by

```
Sum [Times [Const 2; Var "x"]; Const 2]
```

Note that the representation of $2x + 2$ in `aexp` is not unique. For instance, the following also represents $2x + 2$:

```
Sum
 [Times [Const 2; Power ("x", 1)];
  Sum
   [Times [Const 0; Var "x"];
    Times [Const 2; Sum [Times [Const 1]; Times [Var "x"; Const 0]]]];
  Const 0]
```

Problem 5 (10pts) Consider the following expressions:

```
type exp = X
         | INT of int
         | ADD of exp * exp
         | SUB of exp * exp
         | MUL of exp * exp
         | DIV of exp * exp
         | SIGMA of exp * exp * exp
```

Implement a calculator for the expressions:

```
calculator : exp -> int
```

For instance,

$$\sum_{x=1}^{10}(x * x - 1)$$

is represented by

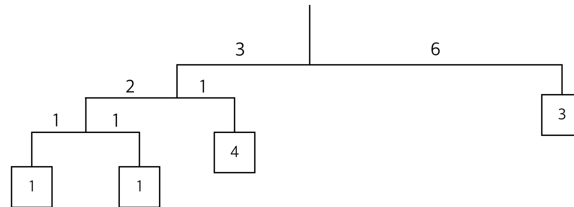
```
SIGMA(INT 1, INT 10, SUB(MUL(X, X), INT 1))
```

and evaluating it should give 375.

Problem 6 (10pts) A binary mobile consists of two branches, a left branch and a right branch. Each branch is a rod of a certain length, from which hangs either a weight or another binary mobile. In OCaml datatype, a binary mobile can be defined as follows:

```
type mobile = branch * branch     (* left and rigth branches *)
and branch = SimpleBranch of length * weight
           | CompoundBranch of length * mobile
and length = int
and weight = int
```

A branch is either a simple branch, which is constructed from a length together with a weight, or a compound branch, which is constructed from a length together with another mobile. For instance, the mobile



is represented by the following:

```
(CompoundBranch (3,
  (CompoundBranch (2, (SimpleBranch (1, 1), SimpleBranch (1, 1))),
   SimpleBranch (1, 4))),
 SimpleBranch (6, 3))
```

Define the function

```
balanced : mobile -> bool
```

that tests whether a binary mobile is balanced. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right

branch (that is, if the length of the left rod multiplied by the weight hanging from that rod is equal to the corresponding product for the right side) and if each of the submobiles hanging off its branches is balanced. For example, the example mobile above is balanced.

Problem 7 (20pts) Binary numerals can be represented by lists of 0 and 1:

```
type digit = ZERO | ONE
type bin = digit list
```

For example, the binary representations of 11 and 30 are

`[ONE;ZERO;ONE;ONE]`

and

`[ONE;ONE;ONE;ONE;ZERO]`,

respectively. Write a function

`bmul: bin -> bin -> bin`

that computes the binary product. For example,

`bmul [ONE;ZERO;ONE;ONE] [ONE;ONE;ONE;ONE;ZERO]`

evaluates to `[ONE;ZERO;ONE;ZERO;ZERO;ONE;ZERO;ONE;ZERO]`.