Brian Simpson

Professor Vu

COP 4020

02/16/21

Homework #4

1. When trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

```
occurs x (Leaf y)          = x == y
occurs x (Node l y r) | x == y = True
                      | x < y = occurs x l
                      | x > y = occurs x r
```

The standard prelude defines

```
Data Ordering = LT | EQ | GT
```

Together with a function

```
Compare :: Ord a => a -> a -> Ordering
```

that decides if one value in an ordered type is less than (LT), equal to (EQ), or greater than (GT) another value. Using this function, redefine the function

occurs :: Ord a => a -> Tree a -> Bool for search trees.

Why is this new definition more efficient than the original version?

occurs :: Ord a => a -> Tree a -> Bool

occurs x (Leaf y) = x == y

occurs x (Node l y r) = case Compare x y of

LT -> occurs x l

EQ -> True

GT -> occurs x r

This definition is more efficient than the original one because it only requires one comparison between x and y to be made for each node in the search tree that is traversed, as opposed to the previous implementation which sometimes required two checks to be made for a single node. Thus in the worst case we have effectively doubled the efficiency.

2. Extend the abstract machine to support the use of multiplication.

Abstract Machine:

```
-----------

data Expr = Val Int | Add Expr Expr

type Cont = [Op]

data Op = EVAL Expr | ADD Int

eval :: Expr -> Cont -> Int

eval (Val n) c = exec c n

eval (Add x y) c = eval x (EVAL y : c)

{-|

 eval evaluates an expression in the context of a control stack. That is, if the expression is an integer, it is already fully evaluated, and we begin executing the control stack. If the expression is an addition, we evaluate the first argument, x, placing the operation EVAL y on top of the control stack to indicate that the second argument, y, should be evaluated once evaluation of the first argument is completed.

-}
```

```
exec :: Cont -> Int -> Int

exec [] n = n

exec (EVAL y : c) n = eval y (ADD n : c)

exec (ADD n : c) m = exec c (n+m)
```

{-|

exec executes a control stack in the context of an integer argument. That is, if the control stack is empty, we return the integer argument as the result of the execution. If the top of the control stack is an operation EVAL y, we evaluate the expression y, placing the operation ADD n on top of the remaining stack to indicate that the current integer argument, n, should be added together with the result of evaluating y once this is completed. And finally, if the top of the stack is an operation ADD n, evaluation of the two arguments of an addition expression is now complete, and we execute the remaining control stack in the context of the sum of the two resulting integer values.

-}

```
value :: Expr -> Int

value e = eval e []
```

Example:

```
*Main> value (Add (Val 3) (Val 4))

7
```

Extend the abstract machine to support the use of multiplication:

```
data Expr = Val Int | Mult Expr Expr

type Cont = [Op]

data Op = EVAL Expr | MULT Int

eval :: Expr -> Cont -> Int

eval (Val n) c = exec c n

eval (Mult x y) c = eval x (EVAL y : c)

exec :: Cont -> Int -> Int

exec [] n = n

exec (EVAL y : c) n = eval y (MULT n : c)

exec (MULT n : c) m = exec c (n*m)

value :: Expr -> Int

value e = eval e []
```