The University of Hong Kong

COMP3258: Functional Programming

Assignment 3

Deadline: 23:59, Dec 2, 2022 (HKT)

---

1. Please do not import other modules in your code.
2. You can create extra/helper/auxiliary function.
3. We encourage to use the template file provided. And please do not modify the type signatures we provided.
4. In this assignment, the coding style will be assessed and graded. Please improve your coding style before the submission.
5. For absurd cases (don't be confused with base case), feel free to raise exception or give a default value for it.
6. Please submit a single Haskell file, named as `A3_XXX.hs`, with `XXX` replaced by your UID, and follow all the type signatures strictly. In the case that Moodle rejects your .hs file, please submit it as `A3_XXX.zip`, which includes only one file `A3_XXX.hs`.
7. Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment, **and explain it in a detailed manner**. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

0 IO Interaction (20 pts)

Problem 1. (10 pts) (Two-player Nim Game)

Implement the game of nim in Haskell, `nim :: Int -> IO ()`.

There are **n** rows of stars on the board, where **n** is the parameter for the function. For example, we take n as 5.

```
Main> nim 5

*     | 1
**    | 2
***   | 3
****  | 4
***** | 5
```

The rules of the nim game are simple:

- Two players take turns to remove one or more stars from a single row.
- The winner is the player who removes the last star(s) from the board.

Example of running the game (Here we use highlighted color to demo, which is not necessary in your implementation):

```
Main> nim 5
*     | 1
**    | 2
***   | 3
****  | 4
***** | 5

Player 1
Enter a row number: 4
Stars to remove: 4

*     | 1
**    | 2
***   | 3
      | 4
***** | 5

Player 2
Enter a row number: 5
Stars to remove: 3

*     | 1
```

```
**     | 2
***    | 3
       | 4
**     | 5

Player 1
Enter a row number: 5
Stars to remove: 2

*      | 1
**     | 2
***    | 3
       | 4
       | 5

Player 2
Enter a row number: 3
Stars to remove: 3

*      | 1
**     | 2
       | 3
       | 4
       | 5

Player 1
Enter a row number: 2
Stars to remove: 2

*      | 1
       | 2
       | 3
       | 4
       | 5

Player 2
Enter a row number: 1
Stars to remove: 1

       | 1
       | 2
       | 3
       | 4
       | 5
```

```
Player 2 wins!
```

If the input of the row number exceeds the existing rows, or stars to remove are bigger than the left stars, please point out the mistake and let the player enter again. For example,

```
Main> nim 5
*     | 1
**    | 2
***   | 3
****  | 4
***** | 5

Player 1
Enter a row number: 6
Stars to remove: 1

Warning: There are only 5 rows in the game. Try again.

Player 1
Enter a row number: 3
Stars to remove: 4

Warning: There are only 3 stars in the row 3. Try again.

Player 1
Enter a row number: 2
Stars to remove: 1

*     | 1
*     | 2
***   | 3
****  | 4
***** | 5
```

Problem 2. (10 pts) (One-player Nim Game)

Implement the alternative nim game `nimAI :: Int -> IO ()` which lets one-player combat with AI.

The strategy of the AI is naive: it will always pick the row with the most stars (if there're multiple options, pick the top), and always take all the stars in that row. Player will take the move first, for example

```
Main> nimAI 5
*     | 1
**    | 2
***   | 3
****  | 4
***** | 5

Player
Enter a row number: 4
Stars to remove: 4

*     | 1
**    | 2
***   | 3
      | 4
***** | 5

AI
Enter a row number: 5
Stars to remove: 5

*     | 1
**    | 2
***   | 3
      | 4
      | 5

Player
Enter a row number: 3
Stars to remove: 1

*     | 1
**    | 2
**    | 3
      | 4
      | 5

AI
Enter a row number: 2
```

```
Stars to remove: 2

*      | 1
       | 2
**     | 3
       | 4
       | 5

Player
Enter a row number: 3
Stars to remove: 1

*      | 1
       | 2
*      | 3
       | 4
       | 5

AI
Enter a row number: 1
Stars to remove: 1

       | 1
       | 2
*      | 3
       | 4
       | 5

Player
Enter a row number: 3
Stars to remove: 1

       | 1
       | 2
       | 3
       | 4
       | 5

Player wins!
```

1 Functional Parsing (35 pts)

Let's consider the following Expr data type for expression trees.

```
data Binop = Add | Sub | Mul | Div | Mod deriving (Eq, Show)
data Expr = Bin Binop Expr Expr
          | Val Int
          | Var String deriving (Eq, Show)
```

It might be a little bit different from the expression tree you have seen, because we are going to reuse `Binop` later. `Binop` stands for binary operations, including addition `Add`, subtraction `Sub`, multiplication `Mul`, division `Div`, modulo `Mod`. `Expr` contains all binary operations over expressions, together with integer literal `Val` and variable `Var`.

We use an environment `Env` to determine the values for variables:

```
type Env = [(String, Int)]
```

The library function `lookup` could be used for searching in an environment.

Problem 3. (5 pts) Implement a function `eval :: Env -> Expr -> Maybe Int` to evaluate expression trees. `eval` should return `Nothing` if the divisor is `0` in the division and modulo cases. Also, if a variable cannot be found in the environment, `Nothing` should be returned (Optional: use the `Maybe` monad in your code).

Expected running results:

```
*Main> eval [] (Bin Add (Val 2) (Val 3))
Just 5
*Main> eval [("x", 2)] (Bin Add (Var "x") (Val 3))
Just 5
*Main> eval [("x", 2)] (Bin Add (Var "y") (Val 3))
Nothing
*Main> eval [] (Bin Div (Val 4) (Val 2))
Just 2
*Main> eval [] (Bin Mod (Val 4) (Val 0))
Nothing
```

Problem 4. (20 pts) Then let's write a parser for those expression trees. You may want to review previous and lecture slides and tutorials when doing this question. Implement a function `pExpr :: Parser Expr` for parsing `Expr`s. The grammar is provided as below:

```
expr      := term op_term
op_term   := ('+' | '-') term op_term | ''
term      := factor op_factor
op_factor := ('*' | '/' | '%') factor op_factor | ''
factor    := '(' expr ')' | integer | identifier
```

You can assume the identifiers start with a lowercase letter and may contain any alphabetic or numeric characters after the first one. `''` in the last alternative case in `op_term` and `op_factor` means empty.

Notice:

- Use the `token` function in `Parsing.hs` to remove leading and trailing spaces.
- Your parser should reflect the left-associativity of the operators. See the second example below.

Expected running results:

```
*Main> parse pExpr "1 + 2"
[(Bin Add (Val 1) (Val 2),"")]
*Main> parse pExpr "1 + 2 + 3"
[(Bin Add (Bin Add (Val 1) (Val 2)) (Val 3),"")]
*Main> parse pExpr "1 + x"
[(Bin Add (Val 1) (Var "x"),"")]
*Main> parse pExpr "1 + x * 3"
[(Bin Add (Val 1) (Bin Mul (Var "x") (Val 3)),"")]
*Main> parse pExpr "1 + x * 3 / 5"
[(Bin Add (Val 1) (Bin Div (Bin Mul (Var "x") (Val 3)) (Val 5)),"")]
```

Problem 5. (10 pts) The compilation in practice can be very complicated. In order to produce efficient machine programs, there are usually many optimization heuristics. One of the simplest heuristics is *Constant folding*, namely, calculation between constants is calculated directly during compilation time instead of at runtime.

Your task is to implement a function `optimize :: Expr -> Maybe Expr` that optimizes an expression according to the following rules:

- Multiplication between any expression e and 0 is simplified to 0.
- Addition between any expression e and 0 is simplified to e.
- Subtraction an expression e by 0 simplified to e.
- Division or Modulo by 0 returns `Nothing`.
- Any evaluations between constants are calculated directly.

Expected running results:

```
*Main> optimize $ Bin Add (Var "x") (Bin Sub (Val 2) (Val 1))
Just (Bin Add (Var "x") (Val 1))
*Main> optimize $ Bin Add (Val 3) (Bin Sub (Val 2) (Val 1))
Just (Val 4)
*Main> optimize $ Bin Add (Val 3) (Bin Mul (Var "x") (Val 0))
Just (Val 3)
*Main> optimize $ Bin Add (Var "x") (Val 0)
Just (Var "x")
*Main> optimize $ Bin Add (Var "x") (Val 1)
Just (Bin Add (Var "x") (Val 1))
*Main> optimize $ Bin Div (Val 3) (Val 0)
Nothing
```

2 Programming with Monads (45 pts)

2.1 State Monad (20 pts)

We have learnt two monads in the course: `IO` Monad and `Parser` Monad. In this section, we're going to practice the usage of `State` Monad. If you're familiar with imperative languages like C++, you may know how to use *global variable*:

```cpp
#include <iostream>

using namespace std;

int x = 0; // x as global variable, and the starter value is 0
```

```
int main()
{
    x = x + 2; // update the value of x by adding 2 onto x
    x = x * x; // update the value of x by computing the square of it
    int y = x; // assign the y with updated x, and y should be 4
    cout << y;

    return 0;
}
```

In Haskell, you can mimic this pattern using `State` Monad,

```haskell
import Control.Monad.Trans.State ( get, put, State, evalState )

type ReturnValue = Int
type GlobalX = Int

startX = 0

y :: State GlobalX ReturnValue  -- State Value
y = do x <- get                 -- 0
       put $ x + 2              -- 2
       x' <- get                -- 2
       put $ x' * x'            -- 4
       get                      -- 4

Main> evalState y startX
4
```

Let's explain this code:

- To use the `State` monad, we first import four essential components from the `State` module.
- The intuition of `State` monad is to suppose there's a global state in the code which can be fetched and modified.
- `get` and `put` are two essential functions to interact with the state, `get` fetches the value of the state and `put` modifies the value of the state.
- `State` is the type constructor that accepts two type arguments: one is the type of the state (in the example, our state is global x), and another is the type of return value of the computation (in the example, the type of the return value is the type of y).

- `evalState` is a function that triggers a computation to run in a state. `evalState :: State s a -> s -> a` accepts two arguments: (1) the value typed with `State s a` (2) the value of the initial state (in the example, it is 0 since the start value of global variable x is 0).

Problem 6. (10 pts) In our 1st assignment, we implement the `solveRPN` by using an `eval` function (You are allowed to import `(.|.)` and `(.&.)` in `Data.Bits` in this question). Check the solution given below:

```haskell
type Stack = [Int]

-- use a extra parameter as stack to store parsed and/or computed value.
eval :: [String] -> Stack -> Int
eval [] [] = 0
-- no unparsed string, return the head of the stack
eval [] (x:_) = x
-- unary operations
-- we pop 1 element of the stack; inc/dec it, and push it back
eval ("inc":xs) (s:ss) = eval xs (s+1:ss)
eval ("dec":xs) (s:ss) = eval xs (s-1:ss)
-- binary
-- we pop 2 elements of the stack; manipulate them
-- and push the computed value back.
eval ("+":xs) (s1:s2:ss) = eval xs (s1 + s2:ss)
eval ("-":xs) (s1:s2:ss) = eval xs (s2 - s1:ss)
eval ("*":xs) (s1:s2:ss) = eval xs (s1 * s2:ss)
eval ("/":xs) (s1:s2:ss) = eval xs (s2 `div` s1:ss)
-- logical operations
eval ("&":xs) (s1:s2:ss) = eval xs (s1 .&. s2:ss)
eval ("|":xs) (s1:s2:ss) = eval xs ((s1 .|. s2):ss)
-- stack operations
eval ("clear":xs) _ = eval xs []
eval ("dup":xs) (s:ss) = eval xs (s:s:ss)
-- reading input
eval (x:xs) ss = eval xs ((read x :: Int):ss)

solveRPN :: String -> Int
solveRPN xs = eval (words xs) []
```

The task of this question is to implement an alternative `evalL` function using State monad. `EvalState` is similar to `Stack` in the above solution given; `EvalValue` is similar to the output type `Int` of the `eval` function.

```haskell
type EvalState = [Int]
type EvalValue = Int

evalL :: [String] -> State EvalState EvalValue
evalL = undefined

solveRPN :: String -> Int
solveRPN xs = evalState (evalL . words $ xs) []
```

The test cases of `solveRPN` should be the same as the cases provided in the first assignment.

Problem 7. (10 pts) In this problem, we will introduce more details of the state monad and define a tailored version of it. We call this new datatype `Stack` (please do not confuse it with the `Stack` given in the solution of 1st assignment). The `Stack` has two essential operations: `pop` and `push`. It differs from the `State` in:

- The global state is fixed: `[Int]`.
- `get` and `put` make changes to the whole state, `pop` and `push` manipulate the stack in a more gentle way: `pop` pops one element from the state; `push` will push one element into the state.

Check the below example how to use the `pop` and `push`.

```haskell
e1 :: Stack Int
e1 = do pop

e2 :: Stack Int
e2 = do x <- pop
       push (x + 1)
       pop

e3 :: Stack Int
e3 = do pop
        pop
        pop

> evalStack e1 [1,2,3] -- returns 1
> evalStack e2 [1,2,3] -- returns 2
> evalStack e3 [1,2,3] -- returns 3
```

We give the `Stack` monad definition below, please try to understand the details and implement two functions `pop` and `push`. In the last, let's use our `Stack` to re-implement our `evalL` function.

```haskell
newtype Stack a = Stack {runStack :: [Int] -> ([Int], a)}

instance Functor Stack where
    fmap = liftM

instance Applicative Stack where
    pure x = Stack $ \s -> (s, x)
    (<*>) = ap

instance Monad Stack where
    return = pure
    m >>= k = Stack $ \s -> case runStack m s of
        (s', x) -> runStack (k x) s'

push :: Int -> Stack Int
push n = undefined

pop :: Stack Int
pop = undefined

evalStack :: Stack Int -> [Int] -> Int
evalStack m s = snd (runStack m s)

evalL' :: [String] -> Stack Int
evalL' = undefined
```

2.2 Maybe Monad and List Monad (25 pts)

Monad has two essential operations: `return` and `>>=` (bind).

2.2.1 Maybe Monad

For the maybe monad, the `return` function wraps the element into a `Just` and `>>=` returns `Nothing` if `m` is `Nothing`, otherwise it takes the data (`x`) out and apply `g` to the `x`.

```haskell
return :: a -> Maybe a
return x  = Just x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(>>=) m g = case m of
              Nothing -> Nothing
              Just x  -> g x
```

We often use `Maybe` to define a safer method, for example, the `log` function will cause runtime error if we apply it to a negative integer.

```haskell
log :: Floating a => a -> a

> log 1000
6.907755278982137
> log (-1000)
''ERROR'' -- runtime error
```

We then define a safer one: `safeLog` by using `Maybe`,

```haskell
safeLog :: (Ord a, Floating a) => a -> Maybe a
safeLog x
    | x > 0      = Just (log x)
    | otherwise = Nothing
```

We apply the same idea to `sqrt` to define `safeSqrt`,

```haskell
safeSqrt :: (Ord a, Floating a) => a -> Maybe a
safeSqrt x
    | x >= 0      = Just (sqrt x)
    | otherwise   = Nothing
```

Previously we can use a composition operation `(.)` to compose two functions,

```haskell
logSqrt = log . sqrt
```

It becomes cumbersome to define a safer `logSqrt` since the `sqrt` return a `Maybe a` instead of `a`, we might write the code

```haskell
safeLogSqrt :: (Ord a, Floating a) => a -> Maybe a
safeLogSqrt x = case safeSqrt x of
                  Nothing -> Nothing
                  Just a -> safeLog a
```

Problem 8. (5 pts) Rewrite `safeLogSqrt` using `return` and/or `>>=` in maybe monad (use `safeLog` and `safeSqrt` provided). (do-notation is not allowed)

2.2.2 List Monad

For the list monad, the `return` function wraps the element into a list and `>>=` will map the `f` to the `xs` and `concat` the resulting list.

```haskell
return :: a -> [a]
return x = [x]

(>>=) :: [a] -> (a -> [b]) -> [b]
xs >>= f = concat (map f xs)
```

We can check the example which defines a sqaure function over lists by using do-notation.

```haskell
squares lst = do
      x <- lst
      return (x*x)
```

This code desugars into

```haskell
squares lst = lst >>= (\x -> return (x*x))
```

And then evaluates into (note here `return` constructs a list with sigle element, that's why we need to use `concat` function to flatten this list).

```haskell
squares lst = concat (map (\x -> [x*x]))
```

Problem 9. (5 pts) Since list monad is similiar to the usage of list comprehension, implement a `zipL` function using list monad. The `zipL` function should work the same with `zip` in Haskell's library. Please directly use `return` and/or `>>=` instead of do-notation.

2.2.3 Composing List and Maybe Monad (Advanced)

Problem 10. (15 pts) Implement the Monad Instance of the 2 forms of composition of `[]` and `Maybe`. And write the reasoning process of three monad laws of it in the comment (taught in the Lecture 10).

```
newtype LM a = LM { getLM :: [Maybe a] }

instance Monad LM where
  -- return :: a -> LM a
  return a = undefined
  -- (>>=) :: LM a -> (a -> LM b) -> LM b
  (>>=) = undefined

newtype ML a = ML { getML :: Maybe [a] }

instance Monad ML where
  -- return :: a -> ML a
  return a = undefined
  -- (>>=) :: ML a -> (a -> ML b) -> ML b
  (>>=) = undefined
```

The `Functor` instances and the `Applicative` instances have been given in the template. Changing the definition of them is allowed if you know what you are doing, but generally discouraged.

There would be no sample input and output for this question. Any definition that satisfies the three monad laws is accepted:

```
x :: a
m :: M a

f :: a -> M b
g :: b -> M c

-- Left Identity of return
return x >>= f = f x

-- Right Identity of return
m >>= return = m
```

```haskell
-- Associativity of (>>=)
m >>= (\x -> f x >>= g) = (m >>= f) >>= g
```

Hints and Clarification

- You should probably be very familiar with the monadic behavior of `[]` (List) and `Maybe` before solving this problem.
- Again, with a correct implementation, all the methods should be fairly simple, so the focus of this question is on the thought process instead of the implementation.
- Thinking about why the following naive definition is wrong might be a good start.

```haskell
instance Monad LM where
  return a = LM []
  m >>= f = LM []

instance Monad ML where
  return a = ML Nothing
  m >>= f = ML Nothing
```

- Strictly speaking, failing to satisfy one of the three rules will get you 0 score for the corresponding monad instance, because both naive implementations above only violate one rule.

- You are allowed to import combinators from other modules in this problem, and the consiceness will be assessed.
