In this half of the problem set you will be getting more familiar with the use of monads. You will be taken from using a generic monad, to exploiting a functional monad, to using a special monad.

**Please Remember:** The problem set is to be handed in at the beginning of class on the day it is due. **No late work will be accepted** unless you specifically request and are granted an extension ahead of time. As always all programming problems should be saved in `/mit/6.827/student/_dirs/<your username>` in an appropriately named file `ps3_problemN.hs`. As always everyone is allowed to do their own work. **You MAY discuss with others. However everyone must turn in their own original work.**
Problem 1

Simple Monadic Functions (20 points)

In this problem we’ll focus on writing functions in the monadic style. You can find the skeleton in /mit/6.827/psets/ps3.

Part a: (5 points) Warmup: permM

We’ve provided you with a monadic factorial function (factM) which prints out the computation that it is doing.

Write the function permM which uses this function to calculate the number of permutations of size \( k \) of a set of distinct elements \( n \). Make sure that the resulting output matches the computation.

Part b: (15 points) Writing generic monadic functions

We discussed in lecture how map, fold, and filter each had monadic counterparts. In this problem you will be defining the functions my_mapM, my_foldM, and my_filterM. These operate as expected, threading the monad from the head of the list down to the tail.

- my_mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
- my_foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
- my_filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]

Problem 2

Writing a Parser (60 points)

In this problem we are going to write a parser for the following grammer.

//Boolean Expression
NAME = [a-z] | [a-z] NAME

E =
  0 | 1 // Constants
  | X   // Don’t care
  | NAME // variable
  | (E)  // Parenthesis
  | !E   // boolean NOT
  | E | E  // boolean OR
  | E & E // boolean AND
  | E ^ E // boolean XOR

//Assignment
A = [a-z]+ := E ;

//Program
P = A | A P
Note that spaces and newlines may appear between any two tokens.

This grammar is ambiguous $E_1 \mid E_2 \& E_3$ could be either $E_1 \mid (E_2 \& E_3)$ or $(E_1 \mid E_2) \& E_3$. For simplicity we are going to assume that all such cases are $E_1 \mid (E_2 \& E_3)$. This will make your task easier.

We will be translating this into the Abstract Syntax:

```haskell
data Expr = EConst Bool
          | EDontCare
          | EVar Name
          | ENot Expr
          | EOr Expr Expr
          | EAnd Expr Expr
          | EXor Expr Expr
          deriving(Eq,Show)

type Assignment (Name,Expr)

type Program = [Assignment]
```

To complete this problem we will be making heavy use of the Maybe monad as briefly pointed out in the lecture. We’ll be writing simple parsers and composing them using the following functions (and others if you care to write them):

- `(<+++>) :: (Parser a) -> (Parser a) -> (Parser a)`
  
  $px <+++> py$ is a parser which first attempts to parse with $px$ and then attempt to parse via $py$.

- `mapP :: (a -> b) -> (Parser a) -> (Parser b)`
  
  $mapP f px$ creates a new parser which parses the string with $px$ and applies function $f$ to the resulting token.

- `joinP :: (a -> b -> c) -> (Parser a) -> (Parser b) -> (Parser c)`
  
  $joinP f pa pb$ sequences parsers $pa$ and $pb$ into a parser which combines the tokens via the function $f$.

Parsers are of the form of a a function from a string to possibly a tuple of the returned token, and the remaining string. The type of Parser is:

```haskell
type (Parser token) = String -> Maybe (token, String)
```

Some simple parsers have been given to you which will parse constant strings, and variables names. You should feel free to write any parser you want. It may not always be the most natural to use
Part a: (5 points) parseConsts

Finish the definition of parseConsts which parses constant Values and don’t cares ("0","1","X"). Use the do syntax.

Part b: (5 points) Write parser parseVar which parses a variable.

Part c: (5 points) dropWS

Write a parser function dropWS which given a parser \( p \) returns a parser which drops all whitespace and then does \( p \).

Part d: (5 points) Write parser parseOp which parses one of the binary operators, and returns a token of type \( Expr \to Expr \to Expr \).

Part e: (5 points) Write parser function parenP which given a parser, parses a token wrapped by parenthesis (remember to allow white spaces).

Part f: (25 points) Write parseExpr which parses an expression. While doing this you will probably want to write the following parsers

- \texttt{parseNot} which parses the negated Expression \((!E)\)
- \texttt{parseSExpr} which parses all expressions except for those with top-level binary operators.
- \texttt{parseExpr'} which parses an operator followed by an expression and returns an appropriate token of type \( Expr \to Expr \).
- \texttt{parseNothing} which parses the empty string and return ()

Part g: (5 points)

Write the parsers \texttt{parseAssignment} and \texttt{parseProgram}.

Part h: (5 points)

Write down 2 pros or cons to the monadic style over the regular (non-monadic) style for this problem.

Problem 3 Evaluation (15 points)

In the previous problem we parsed sets of assignments. In this problem we’re going to evaluate these values. Given a program, and a particular variable, you will asked to return the evaluation
of that variable. To save ourselves from having to recompute the result we are going to use a
HashTable (with keys of type String and values of type Expr) to do the evaluation.

HashTables (as found in Data.HashTable) are embedded in the IO monad. This means that your
evaluation function must be monadic.

You will have to write the monadic function evalExpr :: Program -> Name -> IO Expr. When
you reach a variable, you check the hashTable to see if it already exists, and if so you plug in the
value.

Notice that we have a 3-valued system, with X meaning either 0 or 1. When doing a reduction
with X return the most restrictive value (e.g. X \& 1 = X and X\&0 = 0).

When you add a variable x to the hashTable you should output “added x”. Similarly when you
read a variable x from the hashTable you should print “read x”.

We will guarantee that we will define any variable used in the program and that there are no
recursive definitions (Which means you can only get 0,1, or X as a final result).

Problem 4 Introduction to BSV (10 points)

For the next few problems we are going to be using Bluespec SystemVerilog. Though it’s primary
use is for hardware design we will be expressing more software-like designs. Be aware though, that
this is effectively uncharted waters, and issues are more likely to arise. You can find the language

Setting up the tool: For this you will need to login into a linux athena machine. go to add 6.827
and source /mit/6.827/setup.csh.

Check that the tool is working by typing bsc -v -help. bsc should run, dump out some flags and
tell you how many days of the license are left. If you do not get a definite number, something is
wrong.

Part a: A Simple Example (15 points)

Let’s verify that we can compile a simple example. Place the following testbench in the file GCD.bsv.

```hs
interface GCD;
    method Action start(int x, int y);
    method int result();
endinterface

module mkGCD(GCD);
    Reg#(int) a <- mkReg(0);
    Reg#(int) b <- mkReg(0);

    rule swap((a <= b) && (b != 0));
        a <= b;
        b <= a;
endmodule
```
rule sub((a > b) && (b != 0));
  a <= a - b;
endrule

// I'm a method with a guard of y == 0
method Action start(int x, int y) if (b == 0);
  a <= x;
  b <= y;
endmethod

method int result() if (b == 0);
  return(a);
endmethod
endmodule

module mkTestBench(Empty);
  Reg#(int) state <- mkReg(0);
  GCD gcd <- mkGCD();

  rule start(state == 0);
    let x = 1424;
    let y = 12236;
    $display("Starting GCD to %d %d",x,y);
    gcd.start(x,y);
    state <= 1;
  endrule

  rule check(state == 1);
    let x = gcd.result();
    $display("GCD Result: %d",x);
    state <= 2;
  endrule

  rule terminate(state == 2);
    $display("ending run");
    $finish;
  endrule
endmodule

Compile this with the commands:
bsc -v -u -sim -g mkTestBench test.bsv

bsc -v -sim -e mkTestBench *.ba

./a.out

Compile the program and run it. It should not terminate. There's a mistake in the description. Find it. What is the mistake? Fix it. Verify that the GCD does terminate.

Problem 5 A Simple BSV Problem (15 points)

Now that you're a little more comfortable with using the compiler, let's start with a simple problem. In this Example was a simple problem. We will be making an multiplier with an adder and shifters. Using a similiar inteface to the the GCD from the previous problem.

interface Multiplier;
    method Action start_mult(int x, int y);
    method int result();
endinterface

Fill in the definition of the two methods and the rules you need. Be sure to add state.

module mkMult(Multiplier);
    method Action start_mult(int x, int y);
    method int result();
endmodule

module mkTestBench(Empty);
    Reg#(int) state <- mkReg(0);
    Mult mult <- mkTestBench();

    rule start(state == 0);
        let x = 144;
        let y = 1236;
        $display("Starting %d x %d",x,y);
        mult.start_mult(x,y);
        state <= 1;
    endrule

    rule check(state == 1);
        let x = mult.result();
$display("Result: %d",x);
state <= 2;
endrule

rule terminate(state == 2);
$display("ending run");
$finish;
endrule
endmodule

---

Problem 6  Handling Mutable Lists (25 points)

In this problem we are going to generate a store that will hold mutable lists. We’ll do this by using a static “memory” which we’ll call a cell table.

Starting with this module, write a ListWrapper module which will insert items into lists, make new lists, and joins two lists together (mutating the lists). Make sure that atomicity is preserved.

interface List;
    method Actionvalue#(Addr) newList();
    method Actionvalue#(Addr) insert(Addr oldlist, Value value);
    method Actionvalue#(Addr) join(Addr x, Addr y);
endinterface