Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.827 Multithreaded Parallelism: Languages and Compilers

Problem Set 2

In this problem set you will be making use of the pH compiler in order to run some simple pH code. Because the pH compiler produces standalone executables, however, you're going to need to provide a main function which will be executed when the program starts up. Here's a simple example:

File `fact.hs` (.hs is the standard Haskell suffix):

```
-- a Haskell factorial program

fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n - 1)

-- The main function designates what the program will output.
main = print (fact 5) >>
       print (fact 15) >>
       print (fact 27)
```

You can print arbitrary numbers of lines of output by separating them using the `(>>)` operator as shown above. Many of the problems will ask you to turn in working code which uses a standard main function so that we can automate testing; in the mean time, however, you are welcome to use any main function you like to test and debug your code. Note that you can only print values which are printable; in particular, functions can't be printed, and types declared with a `data` declaration will only be printable if you include a `deriving (Text)` declaration at the end of the data declaration. See the Haskell manual in your reading packet for more details, but note that the classes `Read` and `Show` are bundled together into `Text` in the pH compiler (due, ironically, to the compiler-dependent nature of the `deriving` construct).

Later on you may discover that you'd like to get at intermediate results in your code to ensure that they're correct. There's no officially-defined way to do this; however, every Haskell implementation provides a `Trace` construct, and pH is no exception:

```
trace :: String -> a -> a
```

When invoked, `trace` prints its first argument and returns its second argument. You can use `show` to convert printable objects into strings for `trace`. The dollar sign operator `$` represents right-associative function application (that's backwards from the usual application) and provides a handy way to insert traces unobtrusively:

```
fact 0 = 1
fact n = trace ("fact invoked at "++show n) $
 n * fact (n - 1)
```

In order to invoke the pH compiler, you'll need to use the makefile contained in /mit/6.827/ps-data/Makefile-pH. To do that, you first need to add lockers `6.827` and `gnu`. Thus, if the above example were in a file named fact.hs, we'd invoke the compiler as follows:

```
gmake -f /mit/6.827/ps-data/Makefile-pH fact
```

The 6.827 locker contains a script called `phc` which will run this command; alternatively, it should be easy to set up an alias for it in your Athena dotfiles. The above makefile will build a .c file and then compile that to a .o file, then produce an executable.

The pH compiler on Athena is only built for Sun workstations and can be slow to compile and run, so we suggest you do your work on Sun Ultras. If the slowness becomes a real problem, you have the option of using the Hugs and HBC Haskell compilers, which are installed in the 6.827 locker (and available from www.haskell.org). This will only work for the simple problems on this problem set which are just to get you familiar with programming in Haskell. Later problem sets will require pH mechanisms which you cannot get with Hugs or HBC.

Finally, if you edit your Haskell programs in Emacs, you may find the elisp files in /mit/6.827/emacs-files/ to be helpful. They define Haskell modes which help you format your programs by tabbing over the right amounts and matching parentheses for you. These files are also linked from the course home page under "Support Files." To use a module, add a line like this to your `.emacs` file:

```
;; Haskell
(load "/afs/athena/course/6/6.827/emacs-files/glasgow-haskell-mode.el")
```

Please Remember: 1) You can work in groups of up to three people; include the names of all group members on all problems. 2) Turn in the answer to each problem as a separate packet. 3) Comment your code carefully and include output from sample runs.

Problem 1 — Basic Hindley-Milner typechecking

This problem focuses on basic Hindley-Milner typechecking, without overloading. We begin with a few "finger exercises", where you're asked to find the types of some simple little programs. We then go on to try and demonstrate what Hindley-Milner typing *cannot* do. Finally, there are function types for which only a few possible functions can be defined. We show you a few such types, and ask you to come up with corresponding functions.

Some things to remember as you go along:

- $\rightarrow$ is right-associative; that means you read $a \rightarrow b \rightarrow c$ as $a \rightarrow (b \rightarrow c)$.
- Tuples typecheck analogously to $\rightarrow$.
- We read types in their most general form; thus when we write $a \rightarrow b$ we really mean $\forall a.\forall b.a \rightarrow b$.

- Watch out for non-generic type variables! If we write a type scheme with all its ∀'s in place, non-generic type variables are not quantified. This is why you're asked to indicate them specially in Part a.

Part a:

Give the Hindley-Milner types for the following functions. Assume for the moment that all arithmetic operations take arguments of type *Int* and that all comparisons return results of type *Bool*. In the last part, give types for `result` and `n_again` as well—but distinguish the generic and non-generic type variables:

```
det a b c = (b * b) - 4 * a * c

step (a,b) = (b,b+1)

loopy x = loopy x

repeat n f x =
  if (n==0) then  x
            else  repeat (n-1) f (f x)

sum f n =
  if (n<0) then  0
           else  sum f (n-1) + f n

sumSum f n = sum (sum f) n

decrement n =
  let (result, n_again) = repeat n step (-1,0)
  in  result
```

Part b:

Here are some simple terms which do not typecheck. Explain why.

```
f x = if x then x+3 else x*2

r g x y = if (g x) then  g y
                   else  2+(g y)

s g =
  let h x = g (g x)
  in  h (h 3, h 4)
```

```
fix f =
  let func x = f (x x)
  in func func
```

Part c:

Given a function type, there are often only a few functions we can define which have that type. For example, we can only write one function which has the type $a \rightarrow a$:

```
ident x = x
```

Notice that there is another function, `loopy`, which actually has a *more general* type (which you found in the last exercise):

```
loopy x = loopy x
```

Try to come up with functions which have the following types. Be careful not to give functions whose types are too general! A few of them have several possible answers; you only need to give one.

1. $a \rightarrow Int$
2. $(a, b) \rightarrow (b, a)$
3. $a \rightarrow b \rightarrow a$
4. $a \rightarrow b \rightarrow b$
5. $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
6. $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$
7. $(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$
8. $(a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (a, b) \rightarrow (c, d)$

Problem 2 Typechecking using the class system

Now that you understand Hindley-Milner typechecking, it's time to add overloading. Some of the "finger exercises" use the same code; your answers will be different in the presence of overloading, however.

You should assume the following declarations (which are a subset of Haskell's functionality). Actual code for the operations has been omitted for brevity (and because they're all primitives anyhow).

```
class Eq a where
  (==) :: a -> a -> Bool

class (Eq a) => Num a where
  (+), (-), (*) :: a -> a -> a

class Bounded a where
  minBound, maxBound :: a

instance Eq Bool
  ...

instance Eq Int
  ...

instance Eq Float
  ...

instance Num Int
  ...

instance Num Float
  ...

instance Bounded Bool
  ...

instance Bounded Int
  ...
```

Note that in Haskell numeric constants are overloaded; for the purposes of this exercise, assume whole number constants such as `5` can have any numeric type (so `5 :: (Num a) => a`), and floating-point constants have type `Float`. (The real situation is a bit more complicated in both cases.) Make the contexts you write as small as possible; the context `(Eq a, Num a)` is equivalent to the smaller `(Num a)`. Finally, don't eliminate a context like `(Num a, Bounded a)` by rewriting `a` as `Int`; you should assume other members of these classes exist (they do).

Part a:

Give Haskell types for the following functions.

```
det a b c = (b * b) - 4 * a * c

repeat17 n f =
  if (n==0) then  17
```

```
            else  f (repeat (n-1) f)

spin x y = if x==y then 5
   else 2.7

alleq a b c = if a==b then a==c
      else False

similar a b c = if a==b then a==c
else c

r g x y = if (g x) then  x==3
                   else  y*2==minBound
```

Part b:

Here are some simple terms which do not typecheck. Explain why.

```
d a b = (a + minBound) * 0.5

f x = if minBound==maxBound then x else x+3
```

Problem 3 Programming with Maps and Folds

Higher order functions are one of the key features of Haskell and pH, and they permit writing very concise programs. In this problem, you are to write all your solutions using a combination of the functions `map`, `foldl`, and `foldr`, plus some of your own functions. This style of programming may be foreign to some of you, so don't be afraid to ask questions!

There is boilerplate code for problem 3 in `/mit/6.827/ps-data/ps2-3.hs`. You should turn in your code using the boilerplate (and it should run without error when you do so). Naturally, you may use other `main` functions as you go in order to debug your work.

Part a:

Write a function `remdups` to remove adjacent duplicate elements from a list. For example,

```
remdups [1,2,2,3,3,3,1,1] = [1,2,3,1]
```

Use `foldl` or `foldr` to define `remdups`.

Part b:

Write a function `squaresum` to compute the sum of the squares of the integers from 1 to n. For example,

```
squaresum 3 = 14
```

Part c:

Write a function `capitalize` which capitalizes the first character of every word in a string. Remember a `String` in Haskell and pH is simply a type synonym for `[Char]`. Assume the strings you will be given consist of letters, spaces, and punctuation marks. Note that if you `import Char` at the top of your program you can use the Haskell functions `isUpper`, `isLower`, and `toUpper`.

```
capitalize "hello, there" = "Hello, There"
```

Part d:

The mathematical constant e is defined by:

$$e = \sum_{n \geq 0} \frac{1}{n!}$$

Write down an expression that can be used to evaluate e to some reasonable accuracy.

Note: Parts of this problem can be found in Richard Bird and Philip Wadler, "Introduction to Functional Programming".

Problem 4 Polynomials

In this problem, we'll be looking at operations on polynomials of one variable. A polynomial will be represented as a list of tuples such that each tuple represents a term. The first element of each tuple is the coefficient of the term and the second element is the exponent. For example, the polynomial

$$1 - 6x^5 + 4x^9$$

is represented with the list:

```
[(1,0),(-6,5),(4,9)]
```

Notice that the elements of the list are sorted in order of increasing exponent. Throughout this problem, your functions should maintain this invariant. Use the following *type synonym* to simplify your code:

```
type Poly = [(Int,Int)]
```

There's boilerplate in `/mit/6.827/ps-data/ps2-4.hs`.

Part a:

Implement a function `addPoly` that sums two polynomials. Here's a template for `addPoly`:

```
addPoly :: Poly -> Poly -> Poly
addPoly p1 p2 = <your code here>
```

The type inference algorithm can deduce the type of `addPoly` without the type declaration. Still, adding explicit type signatures is a sound software-engineering technique.

Part b:

Implement the function `mulPoly` that multiplies two polynomials. Make sure to remove terms containing zero coefficients and make sure to maintain the sorted order invariant.

Part c:

Implement a function `evalPoly :: Poly -> Int -> Int` that evaluates a polynomial at a particular value. You'll probably want to use the ^ exponentiation operator.

Problem 5 List Comprehensions

Part a:

To get you started with list comprehensions, we'll work on the example in Section 6.4.2 of the *pH* book. This section presents an interesting application of list comprehensions as a database query language, similar to SQL (Structured Query Language).

Write a query that finds the names of all strongmen who toppled someone of the other side.

Write a function `predecessor` using list comprehensions that, given a strongman's codename, returns the codename of the strongman he toppled.

Use `predecessor` to write `predecessors`: a function that, given a strongman's codename, returns a list of all the strongmen that came before him.

Part b:

The classic Eight Queens chess puzzle is the focus of this part of the problem. Given a chessboard and eight queens, the goal is to place the queens on the board so that no two queens are in check. Since queens can move arbitrarily along rows, columns, and diagonals, this implies that no two queens can share a row, column, or diagonal. The following is a valid solution to the Eight Queens problem:

```
  +-------------------------------+
8 |   |   |   |   |   | Q |   |   |
  |-------------------------------|
7 |   |   |   |   |   |   |   | Q |
  |-------------------------------|
6 |   | Q |   |   |   |   |   |   |
  |-------------------------------|
5 |   |   |   | Q |   |   |   |   |
  |-------------------------------|
4 | Q |   |   |   |   |   |   |   |
  |-------------------------------|
3 |   |   |   |   |   |   | Q |   |
  |-------------------------------|
2 |   |   |   |   | Q |   |   |   |
  |-------------------------------|
1 |   |   | Q |   |   |   |   |   |
  +-------------------------------+
    1   2   3   4   5   6   7   8
```

Your goal is to design a function `queens` that takes a single argument n which is both the size of the board and the number of queens to place on it. For the Eight Queens case, your function should be invoked as `queens 8`. Your function is to return a list of chess boards showing all the legal queen positions, and it should make use of list comprehensions as much as possible.

To represent a chess board, use a list of `Int`'s where each entry in the list corresponds to the row position of a queen. The board pictured above can be represented as: `[4,6,1,5,2,8,3,7]`. The fourth entry in this list, for example, is `5` since a queen is placed in the fifth row of the fourth column in this configuration.

You are also to design a function `displayBoard` which takes a board configuration and returns a "printable" version as a `String`, following the format given above. You needn't worry about 0x0 boards!

In addition to your code, your write-up for this problem will include sample configurations generated by your `displayBoard` routine as well as the total count of solutions for the Eight Queens problem. The boilerplate code in `/mit/6.827/ps-data/ps2-5.hs` will do this for you.

Hint: There are 92 legal queen configurations for a board of size 8. The program should not take more than a minute or so to run.

Problem 6 Text Justification

Editors (like emacs) and word-processors implement two important functions for making rag-tag lines of text look like neat paragraphs: filling and justification. A filling function takes a piece of text like:

```
In the chronicles of the ancient
```

```
        dynasty of the Sassanidae,
who reigned          for         about
               four hundred years, from Persia to the borders
of China, beyond the great river      Ganges itself, we read the praises
of       one of the kings of this race, who      was said to be the best
monarch of his time.
```

and transforms it into

```
In the chronicles of the ancient dynasty of the Sassanidae, who
reigned for about four hundred years, from Persia to the borders of
China, beyond the great river Ganges itself, we read the praises of
one of the kings of this race, who was said to be the best monarch of
his time.
```

A justification function adds spaces between the words to align the right-hand sides of all lines, except the last.

```
In the  chronicles  of the   ancient dynasty  of the  Sassanidae,  who
reigned for about  four hundred years, from Persia  to the  borders of
China,  beyond the great  river Ganges itself, we  read the praises of
one of the kings of this race, who was said  to be the best monarch of
his time.
```

We define the input to this problem as a single string at the top-level of the Haskell program (boilerplate to be found in `/mit/6.827/ps-data/ps2-6.hs`):

```
myText = "... the ancient \n     dynasty of the Sassanidae, ..."
```

The first step in processing the text is to split an input string into words while discarding white space. Words can then be arranged into lines of a desired width, and these lines can then be justified to align their right-hand sides.

Part a:

We define a word as a sequence of characters that does not contain spaces, tabs, or newlines. Haskell provides a function `isSpace` in the `Char` library which indicates whether a given character is whitespace.

Write a function `splitWord :: String -> (Word,String)` that returns the first word in a string and the remainder of the string. If the string begins with a whitespace character, the first word is the empty string. For example,

```
splitWord " beyond the" = ("", " beyond the")
splitWord "kings of "   = ("kings"," of ")
```

Given the type synonym

```
type Word = String
```

write a function `splitWords :: String -> [Word]` that splits a string into words, using `splitWord`.

Part b:

Now we need to break a list of words into lines. We define

```
type Line = [Word]
```

and your job is to write a function `splitLine :: Int -> [Word] -> (Line,[Word])`. The first argument to `splitLine` is the length of the line to be formed. Assume that this length is at least as long as the longest word in the text. The second argument is the list of words we derived from the input string.

To conclude this part, write `splitLines :: Int -> [Word] -> [Line]`, a function that returns a list of "filled" lines given a line width parameter and a list of words.

Part c:

To put it all together, write the functions

```
fill :: Int -> String -> [Lines]
joinLines :: [Line] -> String
```

`fill` takes a line width and a string and returns a list of filled lines. `joinLines` takes the filled lines and puts them together into a single string. Lines are separated in the string by newline (`'\n'`) characters.

Part d:

Modify `joinLines` to justify lines by adding the appropriate number of interword spaces. You are free to choose where to add spaces in the line. Name the resulting functions `justify` and `justifyLines`:

```
justify :: Int -> String -> [Lines]
justifyLines :: [Line] -> String
```

Note: This problem is adapted from Simon Thompson, "Haskell: The Craft of Functional Programming". We use the greedy filling algorithm here, which minimizes the shortfall on each line; better systems try to minimize the squared shortfall on each line to give a more uniform margin. This is why Meta-Q in emacs often reformats a properly-filled paragraph, for example. Good "listy" algorithms for optimal filling have been derived in several papers.