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

Problem Set 2

Out: October 3, 2006
Due: October 19, 2006

In this problem set you will have to write several Haskell programs. If necessary, you may consult Problem Set 1 on how to use the Haskell interpreter `ghci` on Athena. Many of the problems will ask you to turn in working code that uses a standard main function so that we can automate testing; in the meantime, you are welcome to use any main function you like to test and debug your code. Note that you can only print values that are printable; in particular, functions cannot be printed, and types declared with a `data` declaration will only be printable if you include a `deriving Show` declaration at the end of the data declaration.

You may discover that you like to get at intermediate results in your code to ensure that they are correct. There is no officially-defined way to do this; however, every Haskell implementation provides a `trace` construct, and `hugs` is no exception:

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

(In order to use this construct, you'll need a `import Trace` declaration at the top of your program). 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)
```

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 that help you format your programs by tabbing over the right amounts and matching parentheses for you. To use one of these modes, 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: The problem set is to be handed in at the beggining of class on the day it is due. **No late work will be accepted** unless you specifically request 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 `ps2_problemN.hs`. As always everyone is allowed to do their own work. **You MAY discuss with other people Problems 3-5. However everyone must turn in their own original work.**

Problem 1 — Typechecking (25 points)

This problem focuses on typechecking in both out simple type system as well as the basic Hindley-Milner typesystems. In this problem we will ignore overloading. Remember $\rightarrow$ is right-associative; that means you read $a \rightarrow b \rightarrow c$ as $a \rightarrow (b \rightarrow c)$.

Part a: (7 points) Simple Types

Give the simple Hindley Milner types (i.e. no polymorphism) for the following expression. Each top level expression is worth 1 point. Assume that all arithmetic operations take arguments of type *Int* and that all comparisons return results of type *Bool*.

1.
```
det a b c = (b * b) - 4 * a * c
```

2.
```
step (a,b) = (b,b+1)
```

3.
```
sum f n =
    if (n<0) then  0
             else  sum f (n-1) + f n

  sumSum f n = sum (sum f) n
```

4.
```
repeat n f x =
    if (n==0) then  x
              else  repeat (n-1) f (f x)

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

5.
```
loopy x = loopy x
```

Part b: (10 points) Simple Types and Hindley-Milner

Here are some more term. For each term:

- Determine if it types in our simple type system
- State whether it types in the HM type-system (i.e. with polymorphism)
- If so, give the type, else give the principal type. Otherwise state why it was not typable.

1.
```
f x = if x then x+3 else x*2
```

2.
```
foo f =
    let func x = f x x
    in func
```

3.
```
r g x y = if (g x) then  g y
                   else  1+(g y)
```

4.
```
s g =
    let h x = g (g x)
    in  h (h True, h False)
```

5.
```
let id x = x
   in
     if (id x) then (id 0)
               else (id 1)
```

Part c: (8 points) Hindley-Milner

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

```
ident x = x
```

Try to come up with functions that have the following types. Be careful not to give functions whose types are too general! Use Haskell syntax. A few questions may have several possible answers; you only need to give one. (1 point for each question)

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 (10 points)

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: (8 points)

Give Haskell types for the following functions. (1 point each, except `repeat17` and `r` that count as 2 points each)

```
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: (2 points)

Here are some simple terms that 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 (20 points)

Higher order functions are one of the key features of Haskell, 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.

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: (5 points)

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: (5 points)

Write a function `squaresum` to compute the sum of the squares of the integers from 1 to n. For example, `squaresum 3 = 14`. *Note:* In Haskell, `[n..m]` evaluates to the list containing all integers

from n to m, in consecutive order.

Part c: (5 points)

Write a function `capitalize` that 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: (5 points)

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 (30 points)

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 is boilerplate code in `/mit/6.827/ps-data/ps2-4.hs`.

Part a: (8 points)

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: (14 points)

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: (8 points)

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 — Text Justification (30 points)

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. We provide boilerplate code 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: (8 points)

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 that 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`. For example,

```
splitWords " beyond the" = ["beyond","the"]
splitWords "kings of "   = ["kings","of"]
```

Part b: (7 points)

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

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

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. `splitLine` returns a pair consisting of 1) a `Line` that contains as many words as possible, and 2) the list of remaining words.

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: (5 points)

To put it all together, write the functions

```
fill :: Int -> String -> [Line]
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: (10 points)

Write a function `justifier :: Int -> String -> String`. `justifier` takes a line width and a string; it splits the input string into words, arranges them into justified lines, and returns the concatenation of all the justified lines into a single string. Lines are separated in the final string by newline (`'\n'`) characters. You are free to choose where to add spaces in a line.

Note: This problem is adapted from Simon Thompson, "Haskell: The Craft of Functional Programming". We use the greedy filling algorithm here, that 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.