MP 1 — Basic Haskell

CS 421
Revision 1.0

Assigned January 22, 2016
Due February 5, 2016

1 Objectives and Background

The objective for this MP is to practice four major techniques you will need in this course. These are recursion, infinite lists, algebraic data types, and higher order functions.

This assignment is not supposed to be (that) difficult, and you can find answers in many places. (Yes, you can use them, but don't cut and paste. Type them in yourself. This MP will be tested in the testing center!) But this assignment is long. Start early. **No extensions are given in this course.**

2 Getting Started

You will have received a git repository as a student in this course. Update your repository with `git pull` to find a directory called `mp1-haskell`. There you will find a file called `mp1.hs`.

The file will look something like this:

```
module Mp1 where

data Cons a = Cons a (Cons a)
            | Nil
  deriving (Show,Eq)

data Exp = IntExp Int
         | PlusExp [Exp]
         | MultExp [Exp]
```

The first line is a Haskell module declaration. It declares a new namespace called `Mp1` in which your functions will live. The three code lines after that are a type declaration. The `Cons` type declared here is supposed to be isomorphic to Haskell's built-in lists. The three lines after that declare the `Exp` data type, which we will use to write a very small interpreter.

There will be a bunch of other lines like these:

```
-- plus :: insert type signature here
plus = undefined
```

We have specified the types of all the functions in this handout, and then specified that the function is undefined. This is so we can run automated tests against your code without everything breaking if you happen to leave one out. Replace the first line with the type signature and the second line with your code.

For the `plus` example above, your code might look something like:

```
plus :: Num a => a -> a -> a
plus a b = a + b
```

One other note: unless we say otherwise, **you are always allowed to use helper functions.** You are also allowed to use Haskell built-in functions, unless, of course, we are asking you to implement one. (This should go without saying, but somebody always asks...)

3 Recursion Problems

3.1 Recursive List Functions

These functions are all built in to the prelude, so we will use different names than the standard ones.

Problem 1.

Write a function `mytake n xx` that returns the first n elements of xx. This is identical to Haskell's `take`.

```
*Mp1> :t mytake
mytake :: Int -> [Int] -> [Int]
*Mp1> mytake 5 [1,2,3]
[1,2,3]
*Mp1> mytake 5 [1..20]
[1,2,3,4,5]
*Mp1> mytake 0 [1..20]
[]
```

Problem 2.

Write a function `mydrop n xx` that returns all but the first n elements of list xx. This is the same as the HASKELL function `drop`.

```
*Mp1> :t mydrop
mydrop :: Int -> [Int] -> [Int]
*Mp1> mydrop 3 [1,2,3,4,5,6]
[4,5,6]
```

Problem 3.

Write a function `rev xx` which returns the reverse of list `xx`. The function must run in linear time for credit![1] If you use tail recursion you will be able to do it.

```
*Mp1> :t rev
rev :: [Int] -> [Int]
*Mp1> rev [1,2,3,4,5]
[5,4,3,2,1]
```

Problem 4.

Write the function `app xx yy` that takes two lists $[x_0, x_1, \ldots, x_n]$ and $[y_0, y_1, \ldots, y_m]$ and returns the list $[x_0, x_1, \ldots, x_n, y_0, y_1, \ldots, y_m]$.

```
*Mp1> :t app
app :: [Int] -> [Int] -> [Int]
*Mp1> app [1,2,3] [6,7,8]
[1,2,3,6,7,8]
```

3.2 Set Theory

We will use lists to represent sets. To reduce our complexity, assume that all lists are sorted. For all functions in this section, assume the input will be ordered (ascending), and all elements will be distinct. Your output should preserve this property.

Problem 5. Write a function `add x xx` that adds an element `x` to set `xx`.

The type signature is saying that `add` has type `a -> [a] -> [a]` — i.e., it takes an element and a list of the same type of element, and return a list of the same type of element. The `Ord a =>` part says that whatever that type `a` turns out to be, it must be orderable (i.e., $<$ and friends are defined for it.)

```
*Mp1> :t add
add :: Ord a => a -> [a] -> [a]
*Mp1> add 10 [3,8,23,66]
[3,8,10,23,66]
*Mp1> add 23 [3,8,23,66]
[3,8,23,66]
```

Problem 6. Write a recursive function `union xx yy` that returns the union of sets `xx` and `yy`. Yes, you are allowed to use `add`.

```
*Mp1> :t union
union :: Ord a => [a] -> [a] -> [a]
*Mp1> union [2,4,5,9] [4,6,20,33]
[2,4,5,6,9,20,33]
```

Problem 7. Write a recursive function `intersect xx yy` that returns the intersection of sets `xx` and `yy`.

```
*Mp1> :t intersect
intersect :: Ord a => [a] -> [a] -> [a]
*Mp1> intersect [2,4,5,9] [4,6,9, 20,33]
[4,9]
```

[1] If your solution uses ++ (list append), then it's probably $\mathcal{O}(n^2)$.

Problem 8.

Write a function `powerset` that returns the powerset (set of all possible subsets) of its input. For this problem, we do **not** want you to use helper functions, but you may use the functions you wrote above. Use a list comprehension to solve this one. The output must be sorted according to lexicographic ordering.

```
*Mp1> :t powerset
powerset :: Ord a => [a] -> [[a]]
*Mp1> powerset [1,2,3]
[[],[1],[1,2],[1,2,3],[1,3],[2],[2,3],[3]]
```

3.3 Mapping and Folding

Use recursion for the problems in this section: no higher order functions.

Problem 9.

Write a function `inclist` that takes a list $[x_0, x_1, \ldots, x_n]$ and returns the list $[x_0+1, x_1+1, \ldots, x_n+1]$

```
*Mp1> :t inclist
inclist :: (Num a) => [a] -> [a]
*Mp1> inclist [5,4,3,2]
[6,5,4,3]
```

Problem 10.

Write a function `sumlist` that takes a list $[x_0, x_1, \ldots, x_n]$ and returns the sum $\Sigma_{i=0}^{n} x_i$. No using the `sum` builtin!

```
*Mp1> :t sumlist
sumlist :: (Num t) => [t] -> t
*Mp1> sumlist [1..10]
55
```

Problem 11.

Write a function `myzip` that takes two lists $[x_0, x_1, \ldots, x_n]$ and $[y_0, y_1, \ldots, y_m]$ and returns the list $[(x_0, y_0), (x_1, y_1), \ldots, (x_p, y_p)]$, where $p = min(n, m)$.

```
*Mp1> :t myzip
myzip :: [t] -> [t1] -> [(t, t1)]
*Mp1> myzip [1,2,3] [9]
[(1,9)]
*Mp1> myzip [] ["a","b","c"]
[]
*Mp1> myzip ["a","b","c"] [1..10]
[("a",1),("b",2),("c",3)]
```

Problem 12.

Using your `myzip` function, write a function `addpairs` that takes two lists $[x_0, x_1, \ldots, x_n]$ and $[y_0, y_1, \ldots, y_m]$ and returns the list $[x_0+y_0, x_1+y_1, \ldots, x_p+y_p]$, where $p = min(n, m)$.

```
*Mp1> :t addpairs
addpairs :: (Num a) => [a] -> [a] -> [a]
*Mp1> addpairs [1,2,3,4] [40,50,60,90]
[41,52,63,94]
```

4 Infinite Lists

Problem 13.

Write the list `ones` which returns the infinite list of the number 1.

```
*Mp1> :t ones
ones :: [Integer]
*Mp1> take 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

Problem 14. Write the list `nats` which returns the infinite list of the natural numbers.

```
*Mp1> :t nats
nats :: [Integer]
*Mp1> take 10 nats
[1,2,3,4,5,6,7,8,9,10]
```

Problem 15.
Write a list `fib` which is the infinite list of Fibonacci numbers. It must run in linear time. Use `addpairs`.

```
*Mp1> :t fib
fib :: [Integer]
*Mp1> take 10 fib
[1,1,2,3,5,8,13,21,34,55]
```

4.1 Algebraic Data Type Problems

Problem 16. Write a function `list2cons` that converts a HASKELL list into a `Cons`.

```
*Mp1> :t list2cons
list2cons :: [a] -> Cons a
*Mp1> list2cons [2,3,4]
Cons 2 (Cons 3 (Cons 4 Nil))
```

Problem 17. Write a function `cons2list` that converts a `Cons` to a HASKELL list.

```
*Mp1> :t cons2list
cons2list :: Cons a -> [a]
*Mp1> cons2list (Cons 2 (Cons 3 (Cons 4 Nil)))
[2,3,4]
```

Problem 18. Write a function `eval` that takes an `Exp` and returns an integer corresponding to the result of the computation being represented. This can be done very compactly if you make use of higher order functions, but you may use recursion if you prefer.

```
*Mp1> :t eval
eval :: Exp -> Int
*Mp1> eval $ PlusExp [IntExp 10, IntExp 32]
42
*Mp1> eval $ PlusExp [MultExp [IntExp 10, IntExp 4], PlusExp [IntExp 99, IntExp 23]]
162
```

5 Higher Order Functions

For these problems you will rewrite some of your recursive functions using higher order functions. You may **not** use the recursive versions in your solutions.

Problem 19.
Write a function `inclist'` that takes a list $[x_0, x_1, \ldots, x_n]$ and returns the list $[x_0 + 1, x_1 + 1, \ldots, x_n + 1]$ This time, use higher order functions.

```
*Mp1> :t inclist'
inclist' :: (Num a) => [a] -> [a]
*Mp1> inclist' [5,4,3,2]
[6,5,4,3]
```

Problem 20.
Write a function `sumlist'` that takes a list $[x_0, x_1, \ldots, x_n]$ and returns the sum $\Sigma_{i=0}^{n} x_i$. This time, use higher order functions.

```
*Mp1> :t sumlist'
sumlist' :: (Num t) => [t] -> t
*Mp1> sumlist' [1..10]
55
```

Problem 21. Write the `list2cons'` function, this time using higher order functions.

```
*Mp1> :t list2cons'
list2cons' :: [a] -> Cons a
*Mp1> list2cons' [2,3,4]
Cons 2 (Cons 3 (Cons 4 Nil))
```