Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 1 — Out: September 9, 2015
Due: September 22, 2015 at 5 PM

This homework is expected to be done individually by all students.

Problem 1 — Getting Started with Haskell (10 points)

This problem is intended to make you comfortable programming in functional languages, namely Haskell. The choice of the programming environment is entirely up to you – you can take a look at :

http://www.haskell.org/implementations.html

and choose whatever you like. We recommend that you download and install The Glasgow Haskell Compiler (GHC) available from:

http://www.haskell.org/ghc/

If you encounter any problems, feel free to use the GHC installation on Athena. In order to run it you need to add the ghc locker by typing:

```
add ghc
```

You will also need to add `/mit/ghc/bin` to the front of your path. You can now run interactive GHC by typing `ghci`.

In order to make sure things work correctly, type the following excerpt and save it as `p1.hs` in your home directory:

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

plus a b = apply_n ((+) 1) b a
mult a b = apply_n ((+) a) b 0
expon a b = apply_n ((*) a) b 1
```

then run ghci and load the file by typing:

```
:l p1.hs
```

Once you've loaded a file you can reload the file after you've changed it on disk with the command:

```
:r
```

You can now call one of the functions you have just specified by typing:

```
expon 3 4
```

(You should obtain 81).

Now, back to the problem set.

Problem 2 — Newton's method (25 points)

Newton's method is a technique used to find successively better approximations to the roots of a real-valued function.

Given an initial approximation x_i to the root of a function, Newton's method computes a better approximation x_{i+1} according to the following formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Part a: (10 points)

In order to apply newton's method, we will first have to compute the derivative of a given function f. Your first task is to create a function

```
diff f dx x
```

that computes the derivative of f at x through finite differencing using the formula

$$f'(x) = \frac{f(x+dx) - f(x)}{dx}$$

Compute the derivative of a few simple functions by hand, and check the output of your diff function against those analytical results.

Part b: (15 points)

Now, you want to create a function

```
newton_iter f f' x k
```

that given a function f, its derivative f', and an initial estimate x uses k iterations of newton's method to compute a zero for the function.

Use the `newton_iter` function above together with the `diff` function to compute zeros for the following functions and initial estimates

- $sin(x)$, $x_0 = 0.5$, $d_x = 0.01$
- $x^3 - 328 * x^2 - 1999 * x - 1670$, $x_0 = 100$, $d_x = 0.01$

Compare the results of using `diff` with different values of dx with the result of using the analytically derived derivative.

Problem 3 — Functions (25 points)

In this problem we are going to generate a "Set of Integers" type. We will encode the set as a decider for membership in the set.

```
type IntSet = (Int -> Bool)

isMember :: IntSet -> Int -> Bool
isMember f x = f x
```

Part a: Simple Sets (2 points)

Define the Empty set (the set with no elements) and the set of integers (contains all elements).

```
emptySet :: IntSet
emptySet x = ...
allInts :: IntSet
allInts x = ...
```

Part b: Intervals (3 points)

Write the function:

```
-- interval x y contains all the integers in [x,y]
interval :: Int -> Int -> IntSet
interval lBound uBound   = ...
```

Part c: More Interesting Sets (5 points)

Generate a function that given a number `k` produces the set of numbers relatively prime to `k`. (Hint: use Euclid's algorithm)

Part d: Set Operators (10 points)

Now that you can generate some sets, you need to generate operators to combine sets.

Write the following functions:

```
-- Boolean Operators
setIntersection :: IntSet -> IntSet -> IntSet
setUnion        :: IntSet -> IntSet -> IntSet
setComplement   :: IntSet -> IntSet

-- Set generation
addToSet        :: Int   -> IntSet -> IntSet
deleteFromSet   :: Int   -> IntSet -> IntSet
```

Part e: Equality (5 points)

How would we define the equality operator on IntSets? Would we be able to do better if we had used a list of Ints instead of a function to represent our set? What would we have had to give up to do that?

Problem 4 — Using λ combinators (25 points)

The next two problems on this problem set focus on the pure λ-calculus. The idea is to become comfortable with the reduction rules used, and with the important differences between some of the reduction strategies which can be used when applying those rules.

In this problem, we shall write a few combinators in the pure λ-calculus to get familiar with the rules of λ-calculus. Here are the definitions of some useful combinators.

$$
\begin{array}{lcl}
\text{TRUE} & = & \lambda x.\lambda y.x \\
\text{FALSE} & = & \lambda x.\lambda y.y \\
\text{COND} & = & \lambda x.\lambda y.\lambda z.x\ y\ z \\
\text{FST} & = & \lambda f.f\ \text{TRUE} \\
\text{SND} & = & \lambda f.f\ \text{FALSE} \\
\text{PAIR} & = & \lambda x.\lambda y.\lambda f.f\ x\ y \\
\underline{n} & = & \lambda f.\lambda x.(f^n\ x) \\
\text{SUC} & = & \lambda n.\lambda a.\lambda b.a\ (n\ a\ b) \\
\text{PLUS} & = & \lambda m.\lambda n.m\ \text{SUC}\ n \\
\text{MUL} & = & \lambda m.\lambda n.m\ (\text{PLUS}\ n)\ \underline{0}
\end{array}
$$

Now, write the λ-terms corresponding to the following functions *in normal form.*

- (3 points) The boolean AND function.
- (3 points) The boolean OR function.
- (3 points) The boolean NOT function.
- (6 points) The exponentiation function (EXP). You should write two expressions, one using MUL and the other without MUL (note: don't eliminate MUL by substituting the body of the MUL combinator into your first definition—MUL only "stands for" its definition in the first place, so you've done nothing).

- (10 points) The function EQ which produces TRUE if its two parameters are equal and false if they are not (Hint: use the data structure combinators. Don't try to construct a lambda term from whole cloth.)

In addition to the given combinators, you are free to define any others which you think would be useful.

Problem 5 Alternative ways of doing recursion (15 points)

Define a function `FIB` in λ calculus that computes the N^{th} Fibonacci number. Create one version that uses the Y combinator and another version that uses self application in terms of itself (i.e. to compute the N^{th} Fibonacci number you would call `FIB FIB N`).

Problem 6 Normal Order NF Interpreter for the λ calculus (50 points)

In lecture, we discussed interpreters for the λ calculus, and gave two examples: call-by-name, written *cn(E)*, and call-by-value, written *cv(E)*. We consider both of these interpreters to terminate when they return an answer in *Weak Head Normal Form*. In this problem, we're going to look at similar interpreters which yield answers in β *normal form*—that is, an expression which cannot possibly be β-reduced anymore.

Part a: Step-wise Reduction (4 points)

Consider the following term:

$$(\lambda x.\lambda y.x)(\lambda z.(\lambda x.\lambda y.x)z((\lambda x.zx)(\lambda x.zx)))$$

Provide the **first 2 reduction steps each** for *normal order* and *applicative order* strategies.

Remember, in *normal order* we pursue a leftmost redex strategy (choose the leftmost redex). In *applicative order* we pursue a leftmost innermost strategy (choose the leftmost redex, or the innermost such redex if the leftmost redex *contains* a redex).

Part b: A Normal Order Interpreter (10 points)

In the style presented in class, write an normal order interpreter. Remember we're evaluating to normal form not weak head normal form.

Now we're going to code this interpreter up in Haskell.

Part c: Renaming Function in Haskell (10 points)

An expression will be of the form:

```
data Expr =
      Var Name          -- a variable
```

```
    | App Expr Expr    -- application
    | Lambda Name Expr -- lambda abstraction
  deriving
    (Eq,Show) -- use default compiler generated Eq and Show instances

type Name = String  -- a variable name
```

As a first step, write the function `replaceVar :: (Name, Expr) -> Expr -> Expr` which given a variable name x and a corresponding expression e, and an expression in which to do the replacement E, replaces all free instances of the variable x with the given expression e.

Part d: Doing a Single Step (15 points)

Now let's write a function to do a single step of the reduction.

Your normal order reduction will have the form: `normNF_OneStep :: ([Name],Expr) -> Maybe ([Name],Expr)`. The Maybe type is defined in the prelude as:

```
data (Maybe a) =
     Nothing
   | Just a
```

`normNF_OneStep` takes a list of fresh names, and a lambda expression. If there is a redex reduction available, it will pick the correct normal order redex and reduces it (possibly using the given fresh names for renaming). If a reduction was performed resulting in `expr'` and reducing the names list to `names'` the function returns `Just (names' expr')`. Otherwise it will return the value `Nothing`.

Part e: Repetition (3 points)

Now write a function: `normNF_n :: Int -> ([Name],Expr) -> ([Name],Expr)` which given an interger n, and an expression does n redex reductions (or as many as were possible) and returns the result (and the unused names).

Part f: Generating New Names (4 points)

Now we need a way to generate fresh variables names for an expression. Writing generating a list of variable names is simple leveraging the infinite list of positive integers `[1..]`. Use this to generate an infinite list of fresh names called `freshNames`.

Remember, we want fresh variables and it is possible that our "fresh" names aren't really fresh.

What we need to do is make sure the ones we choose are not already used in the Expr.

Write a function `usedNames :: Expr -> [Name]` which given an expression returns all the names used in it.

Part g: Finishing Up (4 points)

Then using these functions write `normNF :: Int -> Expr -> Expr` which given an integer n and an expression does n reductions to it and returns the result.

6.820 Fundamentals of Program Analysis

Fall 2015