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

Problem Set 1 — Out: September 19, 2006
Due: October 3, 2006

This homework is expected to be done individually by all students. Hand-in will be done at the **end** of class on the day that it is due. If you cannot turn in your homework before that time, email the course staff (`6827-staff@lists.csail.mit.edu`) in advance. After solutions have been handed out, no homework will be accepted.

Programming problems must be stored in your folder `/mit/6.827/student_dirs/username` on athena (which is also `/afs/athena.mit.edu/course/6/6.827/student_dirs/username`). **Check that you have a folder and that you can write into your folder!**.

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 our installation on Athena. In order to run it you need to add the 6.827 and the ghc locker by typing:

```
add 6.827
add ghc
```

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 — Simple Functions (25 points)

In this problem, we examine two simple algorithms for numerical integration based on Simpson's rule.

Part a: (10 points)

Given a function f and an interval $[a, b]$, Simpson's rule says that the integral can be approximated as follows:

$$I = \frac{h}{3}\left[f(a) + 4f(a+h) + f(a+2h)\right]$$

where

$$h = \frac{b-a}{2}$$

For better accuracy, the interval of integration $[a, b]$ is divided into several smaller subintervals. Simpson's rule is applied to each of the subintervals and the results are added to give the total integral. If the subintervals of $[a, b]$ are all of the same size p, where $p = 2h$, then we have the *Composite Strategy* for integration.

Write a Haskell program containing a function

```
composite_strategy f a b n
```

where

f is the function to integrate

a, b are the endpoints of the integration interval

n is the number of subintervals such that $h = (b - a)/2n$

Integrate some simple functions and try different values for n.

Part b: (10 points)

If the subintervals of the integration are not all equal and can be changed as required, then we obtain an *Adaptive Strategy* for integration. A simple adaptive integration algorithm can be described as follows:

1. Approximate the integral using Simpson's rule over the entire interval $[a, b]$. Call this approximation *old_approx*.
2. Compute the midpoint x of the interval: $x = (b + a)/2$.
3. Approximate a new interval by applying Simpson's rule to the subintervals $[a, x]$ and $[x, b]$ and add the two results. Call this approximation *new_approx*.
4. If the absolute value of the difference between *new_approx* and *old_approx* is within some limit *sigma* then return *new_approx*.
5. Otherwise, apply the adaptive strategy recursively to each of the subintervals, add the two results, and return the sum as the answer.

Write a Haskell program that contains the function

```
adaptive_strategy f a b sigma
```

Make sure `adaptive_strategy` takes advantage of the parallelism available in the algorithm. Try integrating several functions while varying the *sigma* parameter.

Part c: (5 points)

How do the algorithms vary in complexity? Compare the accuracy of the answers. What factors affect the execution time and efficiency of the two strategies?

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. For this problem programming is **not** necessary, and a writeup will suffice.

```
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 set of squares `squares`, which contains exactly all square numbers.

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. We recommend that you take a look at the pH Book, Appendix A, before you move on with this problem set. 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.

- (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).
- (5 points) The function ONE? which tests whether the given number is 1. (Hint: use the data structure combinators. Don't try to construct a lambda term from whole cloth.)
- (5 points) The function PRED which subtracts 1 from the given number. You may decide what to do when 0 is passed as an argument to PRED. (Extra credit: can you come up with a term T for which (SUC T) reduces to $\underline{0}$? If so, give the term; if not, explain why.)

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

Problem 5 — 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.