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

Problem Set 1 Solutions

Problem 1 Getting Started with Haskell

This problem involved following directions and typing things. A suitable description of a solution can be seen in the description of Problem 1 on the problem set.

Problem 2 Numerical Integration

Part a:

Here is a possible implementation of `composite_strategy`:

```
simpson_i f a b = let
    h = (b-a)/2.0
  in
    ((h/3.0)*((f a)+(4.0*(f (a+h)))+f( (a+2.0*h))))

composite_strategy f a b n = let
    p = (b-a)/n
  in
    if n == 1 then
      (simpson_i f a b)
    else
      ((simpson_i f a (a+p)) + (composite_strategy f (a+p) b (n-1)))
```

Here is some sample output using various functions and values of `n`:

```
The integral of sin from 0.000000 to 3.141593 with n = 1 is 2.094395.
The integral of sin from 0.000000 to 3.141593 with n = 2 is 2.004560.
The integral of sin from 0.000000 to 3.141593 with n = 30 is 2.000000.
The integral of sin from 0.000000 to 3.141593 with n = 210 is 2.000000.
The integral of sin from 0.000000 to 3.141593 with n = 2310 is 2.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with n = 1 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with n = 2 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with n = 30 is 997001013000.000977.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with n = 210 is 997001013000.010620.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with n = 2310 is 997001012999.952393.
```

Part b:

The divide-and-conquer structure of the provided algorithm lends itself easily to parallelization. In the following solution, each recursive call launches a new thread. In addition, the three Simpson calculations per call to `adaptive_strategy` can run in parallel.

```
simpson_i f a b = let
    h = (b-a)/2.0
  in
    ((h/3.0)*((f a)+(4.0*(f (a+h)))+f( (a+2.0*h))))

adaptive_strategy f a b sigma = let
    old_approx = (simpson_i f a b)
    x = (b+a)/2.0
    new_approx = ((simpson_i f a x) + (simpson_i f x b))
  in
    if (abs (new_approx - old_approx)) < sigma then
      new_approx
    else
      let
        part_1 = (adaptive_strategy f a x sigma)
        part_2 = (adaptive_strategy f x b sigma)
      in
        (part_1 + part_2)
```

Here is some sample output using various functions and values of `sigma`:

```
The integral of sin from 0.000000 to 3.141593 with sigma = 1.000000 is 2.004560.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.166667 is 2.004560.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.033333 is 2.000269.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.004762 is 2.000269.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.000433 is 2.000017.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.000033 is 2.000001.
The integral of sin from 0.000000 to 3.141593 with sigma = 0.000002 is 2.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 1.000000 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.166667 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.033333 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.004762 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.000433 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.000033 is 997001013000.000000.
The integral of 4x^3 - 9x^2 + 2x + 13 from 0.000000 to 1000.000000 with sigma = 0.000002 is 997001013000.000000.
```

Part c:

The `composite_strategy` algorithm has the following complexity:

- the *work* $T_1(n)$ (i.e., the execution time on a single processor) satisfies the recursion relation: $T_1(n) = 2T_1(n/2) + \Theta(1)$. Hence, $T_1(n) = \Theta(n)$.
- the *critical path* $T_{\text{inf}}(n)$ (i.e., the execution time on an unlimited number of processors) satisfies the recursion relation: $T_{\text{inf}}(n) = T_{\text{inf}}(n/2) + \Theta(1)$. Hence, $T_{\text{inf}}(n) = \Theta(\ln n)$.
- the *degree of parallelism* is $\bar{P} = T_1(n)/T_{\text{inf}}(n) = \Theta(n/\log_2 n)$.

Note: $T_1(n)$ is what people mean by time complexity when they ignore parallelism issues. If you gave only this answer, you won't lose any point `:)` However, keep in mind that the analysis of an (implicitly) parallel algorithm requires more than just the usual sequential complexity.

The `adaptive_strategy` algorithm has the same asymptotic time bounds in terms of n, the number of subintervals used. n itself is determined by `sigma` and the particular function being integrated.

It is difficult to compare the accuracy of the answers because of the difference in the way one specifies the desired accuracy (`n` versus `sigma`). Intuitively, the adaptive strategy should probably give better results for the same number of subintervals. This is because it tends to subdivide more finely those intervals that will probably benefit most from subdivision. For the examples we used, both algorithms converged on reasonably accurate answers fairly quickly. A discussion on the number of evaluations of the function made by various values of `sigma` is also interesting.

The execution time of `composite_strategy` is affected by both `n` and the particular function being integrated. Likewise, the execution time of `adaptive_strategy` is affected by both `sigma` and the function. `adaptive_strategy` is affected by the function not just via the function's evaluation time, but also through its behavior. This is because the function's shape determines how quickly `old_approx` and `new_approx` converge to within `sigma` of one another.

Since there is non-unity average parallelism for both algorithms, their efficiencies should be affected (positively) by number of processors. Thread creation and management overhead also contribute to execution time, which is why the `sigma` constant is important. The base case in `composite_strategy` should be made large enough to prevent such overhead from causing a degradation. `adaptive_strategy` could likely also benefit from such an optimization.

Problem 3 — Functions (25 points)

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 = False

allInts :: IntSet
allInts x = True
```

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 = (\x -> x >= lBound && x <= uBound)
```

Part c: More Interesting Sets (5 points)

```
squares :: IntSet
squares n = let
               ls = dropWhile (\x -> x*x < n) [1..]
```

```
            n' = head ls
        in
          n == (n'*n')
```

Part d: Set Operators (10 points)

```
setIntersection :: IntSet -> IntSet -> IntSet
setIntersection f g x = f x && g x

setUnion        :: IntSet -> IntSet -> IntSet
setUnion        f g x = fx || g x

setComplement   :: IntSet -> IntSet
setComplement   f x = not (f x)

addToSet      :: Int -> IntSet -> IntSet
addToSet n s x =
                if (n == x) then
                    True
                else
                    s x

deleteFromSet :: Int -> IntSet -> IntSet
deleteFromSet n s x = if (n == x) then
                          False
                      else
                          s x
```

Part e: Equality (5 points)

Two IntSets A and B are equal iff $\forall x \in \mathbb{Z}.x \in A \iff x \in B$. With our definition of of IntSets we could only do this by check for each Int value. Intuitively, this is an infinite task as there are an infinite number of integers.

However, Int is not Integer and is actually bound to 32 bits (or 64 bits on some platforms). Therefore we can in fact determine that if two sets are equal (though it's extremely expensive).

Had we used a finite list representation instead of our functional representation, determining equality would have been simpler as all elements of the set are explicitly enumerable. However, this drastically increases the complexity of membership of certain sets. For instance the set of even numbers can be tested for very quickly as a function, but as a list involves a slew of results.

If we extended the Set to work on arbitrary sized integers (the Integer type), then all the problems with infinity would exist. In general we would not be able to decide equality with either the function or list representation as both may imply an infinite amount of work.

Problem 4 Combinators

Generating a new lambda combinator out of the ether is difficult and takes a lot of creativity. However, once one has created a reasonable set of simple combinators (such as the set that we had given you), often one can write interesting new combinators using the old ones. Simple reductions on these expressions can help keep the sizes down as we build up a librarry.

- boolean AND:

  Once you have the equivalent of an if expression, one can describe AND easily as:

  ```
  AND = \x -> \y -> if x then y else False
  ```

  Translating this to lambda calculus:

  $$\begin{aligned} \text{AND} &= \lambda x.\lambda y.\text{COND } x\ y\ \text{FALSE} \\ &= \lambda x.\lambda y.(\lambda a.\lambda b.\lambda c.a\ b\ c)\ x\ y\ \text{FALSE} \\ &= \lambda x.\lambda y.\ x\ y\ \text{FALSE} \end{aligned}$$

- boolean OR:

  Similarily we can think of OR as:

  ```
  OR = \x -> \y -> if x then True else y
  ```

  Translating:

  $$\begin{aligned} \text{OR} &= \lambda x.\lambda y.\text{COND } x\ \text{TRUE}\ y \\ &= \lambda x.\lambda y.(\lambda a.\lambda b.\lambda c.a\ b\ c)\ x\ \text{TRUE}\ y \\ &= \lambda x.\lambda y.\ x\ \text{TRUE}\ y \end{aligned}$$

- boolean NOT Here again it's natural to think of NOT as:

  ```
  NOT = \x -> if x then FALSE else TRUE
  ```

  Translating:

  $$\begin{aligned} \text{NOT} &= \lambda x.\text{COND } x\ \text{FALSE TRUE} \\ &= \lambda x.(\lambda a.\lambda b.\lambda c.a\ b\ c)\ x\ \text{FALSE TRUE} \\ &= \lambda x.\ x\ \text{FALSE TRUE} \end{aligned}$$

- EXP

  Looking at PLUS and MUL, we see that they're described as:

  $$\begin{aligned} \text{PLUS} &= \lambda m.\lambda n.m\text{SUC ZERO} \\ \text{MUL} &= \lambda m.\lambda n.m(\text{PLUS } n)\ \text{ZERO} \end{aligned}$$

Here, we start with m "op" 0 and then apply a function which takes (m "op" n) to (m "op" $n+1$). We can do almost the same thing with EXP. For EXP we repeat the "multiply by m" function n times on the initial value 1.

which is

$$\text{EXP} \quad = \quad \lambda m.\lambda n.n\,(\lambda x.\text{MUL}\,m\,x)\text{ONE}$$

To solve for EXP without MUL it's easier if we consider one more combinator, the NTH combinator $nth = \lambda n.\lambda f.\lambda x.nfx$ which operates as:

```
nth 0 f x = x
nth n f x = f (nth (n-1) f x)
```

EXP could also be seen as being a kind of NTH-like function which given m, n, f, and x returns the result of applying f m^n times to x. If we borrow from the first EXP implementation we can describe what's going on with the recusion

```
EXP_NTH m n f x = (NTH m) EXP_NTH m (n-1) f x
```

but all this is doing is repeating (NTH m) n times before applying it to f and x. This is exactly what NTH is for:

```
EXP_NTH m n f x = (NTH n) (NTH m) f x
```

This translates to:

$$\begin{aligned}\text{EXP'} \quad &= \quad \lambda m.\lambda n.\lambda f.\lambda x.(\text{NTH}\,n)(\text{NTH}\,m)\,f\,x \\ &= \quad \lambda m.\lambda n.\lambda f.\lambda x.(n)\,(m)\,f\,x \\ &= \quad \lambda m.\lambda n.n\,m\end{aligned}$$

- ONE?

  To solve for one we borrow from the definition of ZERO? = $(\lambda n.n(\lambda x.FALSE)TRUE)$, which applies a function n times to some initial values in such a way that we extract out if it was applied zero times or more.

  To encode this we will use a shifting solution:

```
SHIFTLEFT (x,y) = (y,False)
SHIFTEDPAIR n   = NTH n SHIFTLEFT (False, True)
ONE? n          = FST (SHIFTEDPAIR n)
```

$$\begin{aligned}
\text{SHIFTLEFT} &= (\lambda p.\text{PAIR (SND } p\text{) FALSE)} \\
\text{SHIFTEDPAIR} &= \lambda n.\text{NTH } n \text{ SHIFTLEFT (PAIR FALSE TRUE)} \\
\text{ONE?} &= \lambda n.\text{FST (SHIFTEDPAIR } n\text{)} \\
&= \lambda n.\text{FST ((}\lambda n.\text{NTH } n \text{ SHIFTLEFT (PAIR FALSE TRUE)) } n\text{)} \\
&= \lambda n.\text{FST (NTH } n \text{ SHIFTLEFT (PAIR FALSE TRUE))} \\
&= \lambda n.\text{FST (}n \text{ SHIFTLEFT (PAIR FALSE TRUE))} \\
&= \lambda n.\text{FST (}n\ (\lambda p.\text{PAIR (SND } p\text{) FALSE) (PAIR FALSE TRUE))}
\end{aligned}$$

- PRED

  Predecessor is very similiar to the ONE? function. We use a shifting buffer format to do calculate our predecessor. Because we need to ignore a function call, we do this by precomputing the **next** predecessor to the side and shifting the current one (which was the old next predecessor) to the spot we'll check

```
SHIFTLEFT (x,y) = (y,y+1)
SHIFTEDPAIR n   = NTH n SHIFTLEFT (-1, 0)
PRED n          = FST n (SHIFTEDPAIR n)
```

$$\begin{aligned}
\text{SHIFTLEFT} &= \lambda p.\text{PAIR (SND } p\text{) (SUC (SND } p\text{))} \\
\text{SHIFTEDPAIR} &= \lambda n.\text{NTH } n \text{ SHIFTLEFT (PAIR NEGONE ZERO)} \\
\text{PRED} &= \lambda n.\text{FST (SHIFTEDPAIR } n\text{)} \\
&= \lambda n.\text{FST ((}\lambda n.\text{NTH } n \text{ SHIFTLEFT (PAIR NEGONE ZERO)) } n\text{)} \\
&= \lambda n.\text{FST (}n \text{ SHIFTLEFT (PAIR NEGONE ZERO))} \\
&= \lambda n.\text{FST (}n\ (\lambda p.\text{PAIR (SND } p\text{) (SUC (SND } p\text{))) (PAIR NEGONE ZERO))}
\end{aligned}$$

  Here NEGONE can be any arbitrary value.

- Extra Credit

  There is no combinator that could be used on its own to represent "negative one" (that is, a combinator M1 for which SUCC M1 was equivalent to ZERO). We can show this formally. Consider what happens if we reduce the application of the successor function to M1:

$$\begin{aligned}
& (\lambda n.\ \lambda a.\ \lambda b.\ a\ (n\ a\ b))\ \text{M1} \\
\longrightarrow\ & \lambda a.\ \lambda b.\ a\ (\text{M1}\ a\ b)
\end{aligned}$$

  Now we are left with an expression which, we claim, is not equivalent to ZERO. If this expression is applied to a function which ignores its first argument, then the definition of M1 is irrelevant to the result:

$$\begin{aligned}
& (\lambda a.\ \lambda b.\ a\ (\text{M1}\ a\ b))\ (\lambda x.\ 5)\ 3 \\
\longrightarrow\ & (\lambda x.\ 5)\ (\text{M1}\ (\lambda x.\ 5)\ 3) \\
\longrightarrow\ & 5
\end{aligned}
\qquad\qquad
\begin{aligned}
& \mathit{ZERO}\ (\lambda x.\ 5)\ 3 \\
\longrightarrow\ & (\lambda f.\ \lambda x.\ x)\ (\lambda x.\ 5)\ 3 \\
\longrightarrow\ & 3
\end{aligned}$$

Problem 5 Normal Order NF Interpreter for the λ calculus

Part a: Step-wise Reduction (4 points)

Although you were only required to perform the first **two** reductions, all reductions have been provided.

- Applicative Order Reduction

  Start with the innermost application, and proceed outwards. Subscripts on parentheses mark how far "in" they are. The next expression to reduce is underlined.

  $$
  \begin{aligned}
  &({}_1\lambda x.\lambda y.x)_1({}_1\lambda z.({}_2\underline{({}_3({}_4\lambda x.\lambda y.x)_4 z)_3}({}_3({}_4\lambda x.({}_5 z\ x)_5)_4({}_4\lambda x.({}_5 z\ x)_5)_4)_3)_2)_1 = \\
  &({}_1\lambda x.\lambda y.x)_1({}_1\lambda z.({}_2({}_3\lambda y.z)_3\underline{({}_3({}_4\lambda x.({}_5 z\ x)_5)_4({}_4\lambda x.({}_5 z\ x)_5)_4)_3)_2})_1 = \\
  &({}_1\lambda x.\lambda y.x)_1({}_1\lambda z.\underline{({}_2({}_3\lambda y.z)_3({}_4 z({}_5\lambda x.({}_6 z\ x)_6)_5)_4)_3)_2)_1} = \\
  &\underline{({}_1\lambda x.\lambda y.x)_1({}_1\lambda z.z)_1} = \\
  &({}_1\lambda y({}_2\lambda z.z)_2)_1 = \\
  &(\lambda y.\lambda z.z)
  \end{aligned}
  $$

- Normal Order Reduction

  Subscripts on parentheses mark how far "in" they are. The next expression to reduce is underlined.

  $$
  \begin{aligned}
  &\underline{({}_1\lambda x.\lambda y.x)_1({}_1\lambda z.({}_2({}_3({}_4\lambda x.\lambda y.x)_4 z)_3({}_3({}_4\lambda x.({}_5 x\ x)_5)_4({}_4\lambda x.({}_5 x\ x)_5)_4)_3)_2)_1} = \\
  &({}_1\lambda y.\lambda z.({}_2\underline{({}_3({}_4\lambda x.\lambda y.x)_4 z)_3}({}_3({}_4\lambda x.({}_5 x\ x)_5)_4({}_4\lambda x.({}_5 x\ x)_5)_4)_3)_2)_1 = \\
  &({}_1\lambda y.\lambda z.({}_2\underline{({}_3({}_4\lambda y.z)_4({}_4({}_5\lambda x.({}_6 x\ x)_6)_5({}_5\lambda x.({}_6 x\ x)_6)_5)_4)_3)_2})_1 = \\
  &({}_1\lambda y.({}_2\lambda z.({}_3 z)_3)_2)_1 = \\
  &(\lambda y.\lambda z.z)
  \end{aligned}
  $$

Part b: A Normal Order Interpreter (10 points)

This was harder than the previous interpreters. The problem is that you're proceeding from outside in: you want to evaluate applications only as far as it is absolutely necessary to perform the top-most β-reduction, but *then* you need to ensure that the resulting expression is in normal form. The interpreter is given in two pieces: *lo*, the top-level interpreter, takes an expression and reduces it to normal form. $lo[\![(E_1\ E_2)]\!]$ uses the auxiliary interpreter *ll*: $ll[\![E_1]\!]$ performs leftmost, outermost reductions until E_1 is reduced to a lambda expression (or no further reductions are possible). *As soon as* E_1 is reduced to a lambda expression $\lambda x.E_3$, *ll* returns and *lo* performs the top-most application (which is now the leftmost, outermost reduction).

$$
\begin{aligned}
lo[\![x]\!] &= x \\
lo[\![\lambda x.\ E]\!] &= \lambda x.\ lo[\![E]\!] \\
lo[\![(E_1\ E_2)]\!] &= \textbf{let}\ \mathrm{f} = ll[\![E_1]\!]
\end{aligned}
$$

$$
\begin{array}{l}
\quad\textbf{in} \\
\quad\quad\textbf{case}\ \mathrm{f}\ \textbf{of} \\
\quad\quad\quad \lambda x.E_3 \texttt{ -> } lo[\![E_3[E_2/x]]\!] \\
\quad\quad\quad \_ \quad\quad \texttt{ -> } (f\ lo[\![E_2]\!])
\end{array}
$$

where ll is defined as follows:

$$
\begin{array}{lcl}
ll[\![x]\!] & = & x \\
ll[\![\lambda x.\ E]\!] & = & \lambda x.E \\
ll[\![(E_1\ E_2)]\!] & = & \textbf{let}\ \mathrm{f} = ll[\![E_1]\!] \\
 & & \textbf{in} \\
 & & \quad\textbf{case}\ \mathrm{f}\ \textbf{of} \\
 & & \quad\quad \lambda x.\ E_3 \texttt{ -> } ll[\![E_3[E_2/x]]\!] \\
 & & \quad\quad \_ \quad\quad \texttt{ -> } (f\ lo[\![E_2]\!])
\end{array}
$$

Solutions based on a one-part interpreter have the fatal flaw that they attempt to reduce the function part of an application *all the way to Normal Form* before doing the top-most β-reduction. Not only is this not outermost reduction, it actually fails in cases like the following:

$$((\lambda x.\ x)\ (\lambda f.\ f\ (\lambda x.\ x)\ \Omega))\ (\lambda x.\ \lambda y.\ x)$$

where Ω is an expression which will reduce forever, such as $(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x))$.

When we do our implementation we only do one reduction at a time. As such we implicitly sidestep this problem (though we do waste some time, because we walk down the tree multiple times).

Part c: Renaming Function in Haskell (10 points)

The `replaceVar` function is straight forward. We just walk down the tree and replace all free instances of the given variable. Notice that we stop once we reach a lambda abstraction using the same name.

```
replaceVar :: (Name, Expr) -> Expr -> Expr
replaceVar (n,e) e'@(Var x)       = if n == x then e
                                              else e'
replaceVar (n,e) e'@(App e1 e2)   = App (replaceVar (n,e) e1)
                                        (replaceVar (n,e) e2)
replaceVar (n,e) e'@(Lambda x e2) | n == x    = e'
                                  | otherwise = Lambda x (replaceVar (n,e) e2)
```

Part d: Doing a Single Step (15 points)

Before we do the first step, it's useful to define alpha renaming and beta substitution.

```
alphaRename :: (Name, Name) -> Expr -> Expr
```

```
alphaRename (n,n') = replaceVar (n, Var n')

betaSub :: (Name, Expr) -> [Name] -> Expr -> ([Name], Expr)
betaSub (n,e) ns v@(Var x) | n == x    = (ns,e)
                           | otherwise = (ns,v)
betaSub (n,e) ns (App e1 e2) = (ns'', App e1' e2')
  where
    (ns' , e1') =  (betaSub (n,e) ns  e1)
    (ns'', e2') =  (betaSub (n,e) ns' e2)
betaSub (n,e) (n':ns) (Lambda x eb) | n == x = (n':ns, Lambda x eb)
betaSub (n,e) (n':ns) (Lambda x eb) | n /= x =
    let
      (nv, ns',eb') = if elem x (usedNames e) then
                          (n', ns, alphaRename (x,n') eb)
                      else
                          (x, n':ns, eb)
      (ns'', eb'') = betaSub (n,e) ns' eb'
    in
      (ns'', Lambda nv eb'')
```

Notice the if in the final condition is not necessary: it simply prevents us from rename unnecessarily.

Now that we've done that, a Single step can be defined as:

```
normNF_OneStep :: ([Name], Expr) -> (Maybe ([Name], Expr))
normNF_OneStep (ns, Var x) = Nothing
normNF_OneStep (ns, Lambda n e) =
    case normNF_OneStep (ns,e) of
      Nothing          -> Nothing
      (Just (ns', e')) -> Just (ns', Lambda n e')
normNF_OneStep (ns, App (Lambda n ebody) earg) =
    Just (betaSub (n,earg) ns ebody)
normNF_OneStep (ns, App ef earg) =
    case normNF_OneStep (ns, ef) of
      Just (ns', ef') -> Just (ns', App ef' earg)
      Nothing         -> case normNF_OneStep (ns, earg) of
                           Just (ns',earg') -> Just (ns', App ef earg')
                           Nothing          -> Nothing
```

Note that we specifically have to deal with the `App (Lambda _ _) _` case seperately so the top level redexs are done first.

Part e: Repetition (3 points)

The easiest way to repeat the function is to make a temporary function which has the type matching $\alpha \to \alpha$ and repeating it n times as we've done before. We does this using the `fromMaybe :: a -> Maybe a -> a` which demaybies a value and provides a default in the case of Nothing.

```
normNF_n :: Int -> ([Name], Expr) -> ([Name], Expr)
normNF_n n t@(ns, e) =
   let
     oneStep tup = fromMaybe tup (normNF_OneStep tup)
     rep 0 f x   = x
     rep n f x   = rep (n-1) f (f x)
   in
     rep n oneStep t
```

Part f: Generating New Names (4 points)

```
freshNames :: [Name]
freshNames = map (\x -> ``x'' ++ show n) [0..]

usedNames :: Expr -> [Name]
usedNames = nup (usednames' e)
            where
              usedNames' (Var x)     xs = [x]
              usedNames' (App e1 e2) xs = usedNames' e1 ++ usedNames' e2
              usedNames' (Lambda x e)   = usedNames e ++ [x]
```

Here in usedNames, it would have also been okay to have used `usedNames e \\ [x]` as `x` is clearly only local here, but the operapproximation is correct.

Part g: Finishing Up (4 points)

Once we've generated a fresh name generator, all we have to do is splice the names in and take them out when we return.

```
normNF :: Int -> Expr -> Expr
normNF n e =
  let
    new_names = freshNames \\ (usedNames e)
    (_,e') = normNF_n n (new_names, e)
  in
    e'
```