CMPT 383
Quiz #8
December 1, 2005

1. Imagine a language of expressions for representing integers defined by the syntax rules:
   (i) `Zero` is an expression
   (ii) If `e` is an expression, then so are `succ(e)` and `pred(e)`.
   (iii) If e_1 and e_2 are expression, then so is `add(`e_1`,`e_2`)`.

   An evaluator reduces expressions in this language by applying the following rules repeatedly until no longer possible:

   `succ(pred(e)) = e`
   `pred(succ(e)) = e`
   `add(zero,`e_2`) =` e_2
   `add(succ(`e_1`),`e_2`) = succ(add(`e_1`,`e_2`))`
   `add(pred(`e_1`),`e_2`) = pred(add(`e_1`,`e_2`))`

   Simplify the expression `add(succ(pred(zero)),zero)` using:

   a) Innermost reduction sequence.
   `add(succ(pred(zero)),zero)`
   { by succ(pred(e)) = e }
   `add(zero,zero)`
   { by add(zero,e) = e }
   `zero`

   b) Outermost reduction sequence.
   `add(succ(pred(zero)),zero)`
   { by add(succ(e_1),e_2) = succ(add(e_1,e_2)) }
   `succ(add(pred(zero),zero))`
   { by add(pred(e_1),e_2) = pred(add(e_1,e_2)) }
   `succ(pred(add(zero,zero)))`
   { by succ(pred(e)) = e }
   `add(zero,zero)`
   { add(zero,e_2) = e_2 }
   `zero`

   c) Can the expression be simplified using another reduction sequence?
   `add(succ(pred(zero)),zero)`
   { by add(succ(e_1),e_2) = succ(add(e_1,e_2)) }
   `succ(add(pred(zero),zero))`
   { by add(pred(e_1),e_2) = pred(add(e_1,e_2)) }
   `succ(pred(add(zero,zero)))`
   { by add(zero,e_2) = e_2 }
   `succ(pred(zero))`
   { by succ(pred(e)) = e }
   `zero`

2. Which of the following equations are *true* for all `xs` and which are *false*?

a) `[]:xs = xs`	**`False`**	g) `[[]] ++ xs = [xs]`	**`False`**
b) `[]:xs =[[],xs]`	**`False`**	h) `[[]] ++ xs = [[],xs]`	**`False`**
c) `xs:[] = xs`	**`False`**	i) `[[]] ++ [xs] = [[],xs]`	**`True`**
d) `xs:[] = [xs]`	**`True`**	j) `[xs] ++ [] = [xs]`	**`True`**
e) `xs:xs = [xs,xs]`	**`False`**	k) `[xs] ++ [xs] = [xs,xs]`	**`True`**
f) `[[]] ++ xs = xs`	**`False`**		

3. Define a data type `Direction` whose values describe the four major points of the compass, and define a function `reverse` for reversing direction.

```
data Direction = North | South | East | West
       deriving (Enum)

reverse :: Direction -> Direction
reverse d = toEnum( mod (fromEnum d + 2) 4 )
```

or

```
data Direction = North | South | East | West
       deriving (Eq)

reverse :: Direction -> Direction
reverse d
       | d == North = South
       | d == South = North
       | d == East = West
       | d == West = East
```

4. Evaluate `map (map square) [[1,2],[3,4,5]]`

```
[[1,4],[9,16,25]]
```

5. The Fibonacci numbers f_0, f_1, … are defined by the rule that $f_0=0$, $f_1=1$ and $f_{n+2}=f_n+f_{n+1}$ for all $n>=0$. Give a definition of the function `fib` that takes an integer `n` and returns `fn`.

```
fib :: Integer → Integer
fib 0 = 0
fib 1 = 1
fib (n+2) = fib n + fib (n+1)
```

6. Using pattern matching, define a function `rev2` that reverses all lists of length `2`, but leaves others unchanged. Ensure that the patterns are exhaustive and disjoint.

```
rev2 :: [a] -> [a]
rev2 [] = []
rev2 (x:[]) = [x]
rev2 (x:y:[]) = [y,x]
rev2 (x:y:z:xs) = (x:y:z:xs)
```

7. Define a function `nextlet` that takes a letter of the alphabet and returns the letter coming after it. Assume that letter `'A'` follows `'Z'`.

```
nextlet :: Char → Char
nextlet c
       | c == 'Z'    = 'A'
       | c == 'z'    = 'a'
       | otherwise   = chr(ord c + 1)
```

8. Explain informally, but clearly and fully, what the function `assert_length`, defined below, does.

```
assert_length           :: Integer → [a] → [a]
assert_length 0 xs      = []
```

```
assert_length (n+1) xs  = (head xs) : assert_length n (tail xs)
```

The function `assert_length` **takes a nonnegative integer** *n* **and a list** *xs* **as arguments. This function selects an initial segment (the fist** *n* **elements) of the given list (***xs***). An error is produced if the list (***xs***) is smaller than the number or elements required (***n***).**