Answer all questions in place. Please do not attach extra sheets. If you have anything to tell us or your answer is overflowing, please use the "notes" spaces given next to each problem.

Your Name:				_	
Your Student	Id: _				

P1 (4 points) For the scala function below, write down all the correct facts from the ones given besides the code.

```
def foo(x: Int): Int = {
   if ( x > 20 ) {
      return 1 + foo(x-1)
   }
   if ( x < 100) {
      return 0
   }
   return foo(x-1)
}</pre>
```

- (A) The function is *not* tail recursive.
- (B) Scala will complain at compile time that the function's return type is mismatched.
- (C) It compiles successfully without an error but throws an exception at runtime for certain values of input x.
- (D) It compiles successfully without an error but fails to terminate for some values of input x.
- (E) It compiles and runs successfully for all integer values of input x.

Write down all of the options from (A) -(D) that are correct. No justifications needed.

```
Answers A, E
```

P2 (8 points) Below is a a recursive function to compute the count of odd numbers in a list. Note that in scala 1.head returns the very first element of a list and 1.tail returns a list that includes all but the very first element from the list l.

```
def countNumOdd(1: List[Int]): Int = {
   if (l.length == 0) 0
   else if (l.head % 2 == 1) 1 + countNumOdd(1.tail)
   else countNumOdd(1.tail)
}
```

Fill up the blanks to complete the tail recursive version of the countNumOdd function.

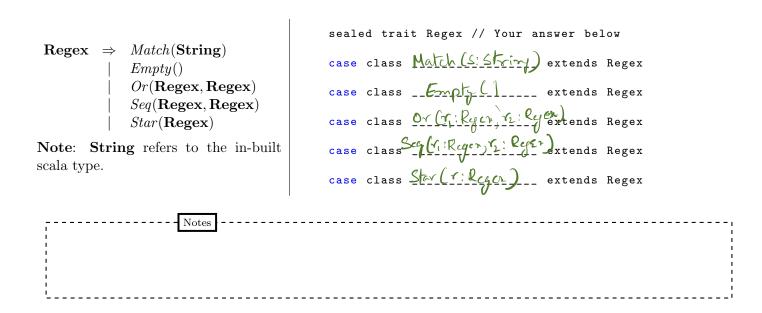
```
def countNumOddTail(1: List[Int], acc: Int = _O____): Int = {

if (1.length == 0) _____OCC ____//Answer here

else if (1.head % 2 == 1) CountNumOddTail(Lial, act)/Answer here

else _Count NumOddTail(Lial, acc) ____//Answer here
}
```

P3 (3 points) Complete the missing portions in the scala definition corresponding to the grammar for regular expressions given below. Please keep the names of terminals and non-terminals unchanged.



P4 (7 points) The evaluation for let bindings we discussed in class/notes is called "eager" evaluation because we *eagerly* evaluate let bindings whether they are needed or not. For example, the following let expression never uses the value of x, but eager evaluation nonetheless computes fact(50):

```
let x = fact(50) in
200
```

An alternative to eager evaluation is "lazy" evaluation where we evaluate a let binding if and only if it is necessary. A simple way to carry out lazy evaluation is by changing the type of the environment env from Map[String, Value] to Map[String, Expr]. In other words, a let variable is now bound to its defining expression e (instead of the result of evaluating e) in the environment env. We define the following semantic rule for the let expression under our version of lazy evaluation:

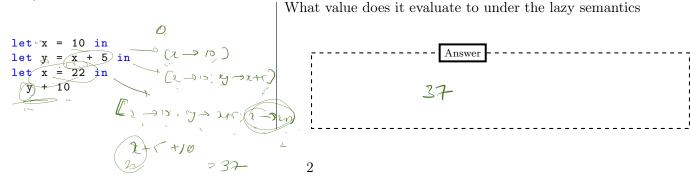
$$\frac{1}{\mathsf{eval}(\mathsf{Let}(id, e_1, e_2), env) = \mathsf{eval}(e_2, env \circ \{id \to e_1\})}(\mathsf{lazy-let-ok})$$

Indeed, we need to change the variable/identifier evaluation rule to make the lazy evaluation work:

$$\frac{id \in dom(env)}{\mathsf{eval}(\mathsf{Ident}(id), env) = \mathsf{eval}(env(id), env)} (\mathsf{lazy\text{-}var\text{-}ok})$$

The error rule for variables remains the same as discussed in class notes: if  $id \notin dom(env)$  then eval(Ident(id), env) evaluates to error. The other rules remain the same as well (refer to class notes since this is just a sample exam).

(a, 3 points) Consider the following program:



(a, 4 points) Consider four lettuce programs as shown below. Which among the 4 evaluate differently under the new rules compared to the standard old rules?

let 
$$x = 10$$
 in
let  $y = x + 5$  in
let  $x = 22$  in
 $y + 10$ 

let  $z = 1$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z + 5$  in
let  $z = 2$  in
 $z =$ 

P5 (4 points) Fill in the missing parts of the following two rules for and expression such that it evaluates as expected.

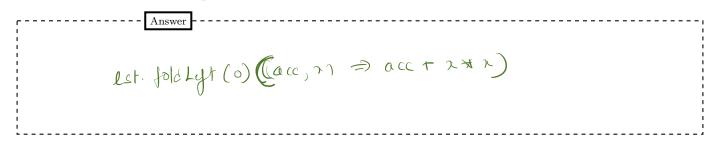
(a, 2 points) Complete the "and-ok-long" rule for the non-error case when evaluating  $e_2$  is necessary:

(a, 2 points) Complete the "and-ok-short" rule for the non-error case when evaluating  $e_2$  is not necessary (short circuiting):

$$\frac{\operatorname{eval}(e_1,env) = v_1, \ v_1 \neq \operatorname{error}, \ \ \underline{\vee}_{\mathsf{L}} = \$$

P6 (4 points) For each of the following problems requires solution in Scala using a combination of map, filter, foldLeft, foldRight, and/or zip. No other API functions should be used.

(a, 2 points) Given 1st of type List[Int], compute sum of square of each element in the list. Example: 1st = List(1, 2, 3, 5), output; 39)



(a, 2 points) Given two lists of equal size, 1st1 and 1st2, both of type List[Int], compute the list of pairs  $(x_i, y_i)$  such that  $x_i$  and  $y_i$  are the *i*'th elements of 1st1 and 1st2 (respectively), and  $x_i > y_i$ . Order of elements in the output list corresponds to that in the input list.

Example: lst1 = List(1, 5, 2, 3, 8, 6), lst2 = List(3, 4, 4, 7, 2, 1), output: List((5,4), (8,2), (6,1))