cs150: Exam 1

Due: Wednesday, 25 February at 3:30pm (in class)
or by 6:30pm (under door of Olsson 219)

UVA ID (e.g., wrw6y) :

Directions

Work alone. You may not discuss these problems or anything related to the material covered by this exam with anyone except for the course staff between receiving this exam and turning it in.

Open resources. You may use any books you want, lecture notes, slides, your notes, and problem sets. You may ***not*** use DrScheme, but it is not necessary to do so. You may also use external non-human sources including books and web sites. If you use anything other than the course books, slides, and notes, cite what you used. You may not obtain any help from other humans other than the course staff.

Answer well. Answer all questions 1-9 (question 0 is your name, which hopefully everyone will receive full credit for), and optionally answer questions 10-11.

You may either: (1) print out this exam and write your answers on it or (2) write your answers directly into the provided Word template and print the result out. Whichever one you choose, you must turn in your answers printed on paper and they must be clear enough for us to read and understand. You should not need more space than is provided to write good answers, but if you want more space you may attach extra sheets. If you do, make sure they are clearly marked.

The questions are not necessarily in order of increasing difficulty, so if you get stuck on one question you should continue on to the next question. There is no time limit on this exam, but it should not take a well-prepared student more than a few hours to complete. It may take you longer, though, so please do not delay starting the exam. There is no valid excuse (other than a medical or personal emergency) for running out of time on this exam.

Full credit depends on the clarity and elegance of your answer, not just correctness. Your answers should be as short and simple as possible, but not simpler. Your programs will be judged for correctness, clarity and elegance, but you will not lose points for trivial errors (such as missing a closing parenthesis).

Your Scores

0	1	2	3	4	5	6	7	8	9	Total
10	10	10	10	10	10	10	10	10	10	100

(Your scores are recorded on the second page so that they are not visible to other students when tests are distributed or passed back.)

1. The **and**-expression is a special form for logical conjunction. An **and**-expression has any number of operand expressions. We will consider a restricted form of the **and**-expression in which every argument must be either #t or #f. For example, all of the expressions below are valid **and**-expressions (question 2 will describe the evaluation rule for **and**-expressions):

```
> (and)
#t
> (and #t #t)
#t
> (and #t #t #f #t #t)
#f
```

Define a BNF grammar rule for the *AndExpression*.

AndExpression ::=

2. For simplicity, the rest of this question assumes a limited version of the **and**-expression that only takes two operands: `(and` $expr_1$ $expr_2$`)`**.** However, the operands to the **and**-expression in this question can be any Scheme expressions (not just #t and #f). This simplified **and**-expression behaves identically to the standard and-expression when applied to two operands, but is not defined for other than two operands. The evaluation rule for an **and**-expression is:

> To evaluate an and-expression, evaluate the first subexpression. If it evaluates to a false value, the value of the and-expression is false. Otherwise, the value of the and-expression is the value of the second subexpression.

Professor Wrongo doesn't like unnecessary special forms and suggest that the **and**-expression special form can be replaced with this procedure:

```
(define (and e1 e2) (if e1 e2 #f))
```

Provide a convincing argument that the `and` procedure above is not equivalent to the **and**-expression special form. (Hint: describe inputs where they mean different things.)

For convenience, here is the find-maximum procedure from Chapter 4:

```
(define (find-maximum f low high)
    (if (= low high)
        (f low)
        (max (f low)
             (find-maximum f (+ low 1) high)))))
```

3. (This is a slight rewording of Exercise 4.8 in the book.) The `find-maximum` procedure we defined in Chapter 4 evaluates to the maximum value of the input function in the range, but does not provide the input value that produces that maximum output value. Define a procedure, `find-maximizing-input` that takes the same inputs as `find-maximum`, but outputs the input value in the range that produces the maximum output value.

For example:

```
> (find-maximizing-input (lambda (x) x) 3 150)
150
> (find-maximizing-input (lambda (x) (- (* 12 x) (* x x))) 0 50)
6
```

For maximum credit, your answer should have running time in $\Theta(n)$ where n is the difference between the values of the `high` and `low` inputs. (But you will receive most of the credit even if your solution is less efficient.) You may assume `low` <= `high` and that `f` runs in $\Theta(1)$.

```
(define (find-maximizing-input f low high)
```

4. Define a `make-incrementer` procedure that takes one input, the increment number, as input and produces as output a procedure. The output procedure is a procedure that takes one number as input, and produces as output the value of that number increased by the increment number.

For example:

```
> (make-incrementer 1)
#<procedure>
> ((make-incrementer 2) 148)
150
>((make-incrementer 3) ((make-incrementer 7) 1))
11
```

```
(define (make-incrementer n)
```

5. Define a procedure `find-worst` that takes two inputs: a non-empty list and a comparison function. As output it produces the element in the list which is the worst according to the comparison procedure. (No points off if you mistakenly write `find-best`.)

For example:

```
> (find-worst (list 1 5 0) <)
5
```

```
(define (find-worst lst cf)
```

6. Answer each of the following questions about function growth. Give an argument that explains each answer. For example, to prove that n is in O($n/2$) you might demonstrate the constants $n_0 = 1$ and $c = 2$. If there is any ambiguity, use the definitions from Chapter 7 of the course book.

Is n in O($2n+5$) ? Why or why not?

Is n^2 in O($2n+5$) ? Why or why not?

Is $4n^2$ in $\Omega(n)$? Why or why not?

Is $4n^2$ in $\Omega(n^3)$? Why or why not?

Is $n\log n$ in $\Theta(n^2)$? Why or why not?

Another cryptographic invention of Alan Turing's at Bletchley Park was a process then known as *banburismus* (more commonly now called *sequential analysis*) developed to determine if two intercepted Enigma messages had been encrypted using the same key.

The goal of the banburismus technique is to determine when two intercepted Enigma messages were encrypted using the same or similar initial machine settings. The key insight is identical to that behind the double delta technique used by Colossus: since the letter distribution in a natural language (in this cases German) is not even, it is much more likely that the same letters will occur at a given position than would occur by random chance (which is approximately the case if the Enigma machines were not configured with similar wheel settings).

So, to determine if two messages were sent by Enigma machines with the same wheel settings we need to count the number of occurrences in the ciphertext where the two messages have the same letter at the same position. If that number significantly exceeds the number predicted by random chance, then it is likely the messages were encoded using the same wheel settings.

7. Define a procedure `count-matches` that takes as input two lists of characters (representing two intercepted ciphertexts) and outputs a number that is the number of positions where the characters in the two lists match (use `eq?` to test two characters for equality).

For example,

```
> (count-matches (list #\A #\B #\C) (list #\B #\B #\C)
2
> (count-matches (list #\A #\B #\C) (list #\B))
0
>(count-matches (list #\A #\B #\C) (list #\A #\B #\C #\D))
3
```

```
(define (count-matches lst1 lst2)
```

Since the Enigma wheels rotate, it was also possible to use this technique to find messages sent using similar initial configurations by trying different alignments of the two messages. For example, suppose the intercepted messages are:

```
Message 1: GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
Message 2: YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

The Bletchley Park analysts would try aligning the messages at different starting points, looking for ways of aligning them that have a high number of matches. For example:

Align 0:
```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

Align +1
```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
 YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
...
```
Align +9
```
GXCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
         YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```

With the Align +9, there are 9 matches which is promising (that is, it would be very unlikely to occur by chance, so the wheel settings are probably similar).

8. Define a procedure `find-best-alignment` that takes as input two lists, representing two intercepted ciphertexts, and outputs the number of matching letters in the best possible alignment. You may assume that `count-matches` is correctly defined.

A good answer will find the best positive alignment (only considering moving the second message to the right, as in the example above). An excellent answer will consider both positive and negative alignments (moving the second message to the left instead).
For example:

```
> (find-best-alignment (list #\A #\B #\C) (list #\B #\B #\C))
2
> (find-best-alignment (list #\A #\B #\C #\D) (list #\B #\C #\D))
3
> (find-best-alignment (list #\A #\B #\C #\D)
                       (list #\F #\A #\B #\C #\D))
4
```

This is the correct result for "excellent" answers that consider negative alignments. The answer using just positive alignments is `0`.

Hint: note that the letters with no corresponding letter in the other message don't matter. So, we could view Align +1 above as
```
XCYBGDSLVWBDJLKWIPEHVYGQZWDTHRQXIKEESQSSPZXARIXEABQIRUCKHGWUEBPF
YNSCFCCPVIPEMSGIZWFLHESCIYSPVRXMCFQAXVXDVUQILBJUABNLKMKDJMENUNQ
```
without the leading `G` in message one. Hint 2: this is different than rotate-wheel in PS4 in that you probably do *not* want to move the first character to the end of the list. Hint 3: helper functions are perfectly legal.

(Answer space is on the next page)

8 (continued). Define your `find-best-alignment` procedure here:

```
(define (find-best-alignment msg1 msg2)
```

9. Analyze the running time of your `find-best-alignment` procedure. Your analysis should include a description of the running time using Θ notation.

10. Extra Credit 1: 2 points. Either indicate that you went to at least one of the Shuttle Rescue demonstrations (the CS 340 bit with the Lego robots) for at least 20 minutes ***or*** read http://shuttle.cs.virginia.edu:8080/specs.html and use techniques from PS2 and PS3 to solve the following problem: You are given a non-empty list of routes from the start to the location of the Escape Pod. Each route is a list commands such as "go forward *X* centimeters" or "rotate right *Y* degrees". It takes 1 unit of battery power to turn 10 degrees and 1 unit of battery power to go forward 5 centimeters. Write a function that returns the cheapest route to the escape pod.

```
(define (find-best-route routes)
```

11. (0 points.) Do you feel your performance on this exam will fairly reflect your understanding of the course material so far? If not, explain why. You will not lose any points for your answer.