UNIVERSITY OF TORONTO
Faculty of Arts and Science

Final Examination

CSC324H1F

December 10, 2014 (**3 hours**)

Examination Aids: Cheat sheet on back, detachable!

Name:

Student Number:

Please read the following guidelines carefully!

- This examination has **10** questions. There are a total of **16 pages, DOUBLE-SIDED**.
- You may use helper functions unless explicitly told not to.
- Any question you leave blank or clearly cross out your work and write "I don't know" is worth **10% of the marks**.
- You must earn a grade of **at least 40% to pass this course**.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total
Grade											
Out Of	10	8	10	6	13	9	6	7	7	6	82

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

It's been a real pleasure teaching you this term.

Good luck!

1. (a) **[3]** Carefully explain the difference between the three Racket values below.

```
(define first-one 1)
(define (second-one) 1)
(define third-one (1))
```

(b) **[2]** Consider the following nested function call expression in Racket. Write down the order in which the function call expressions will be evaluated, using what you know about eager evaluation in Racket.

```
(a b (c (d f) (e)) (b c))
```

(c) **[2]** Define a **closure**. Then, give an example using closures to illustrate the difference between **lexical scope** and **dynamic scope**.

(d) **[3]** State the output(s) for the following code snippets (each run separately). No explanation required.

```
(define x 1)
(define (f x) (+ x 10))

(let* ([x 3]
       [f (lambda (y) (+ x y))])
      (f 5))

(f 5)
```

Outputs (2 of them!):

```
(define a 10)
(define b a)

(let ([b 3])
  ((lambda (c) (+ a b c)) 12))
```

Output:

2. (a) [**4**] Implement the **Racket** function `num-common`, which takes two lists of *distinct* elements, and outputs the number of elements that appear in both lists. Sample usage:

```
(num-common '(1 5 29 0 "Hi") '("Hi" 3 2 1 -10))    ; returns 2
```

You may use either explicit recursion or higher-order list functions. You may not use any of Racket's set functions (e.g., `intersect`), nor `remove-duplicates`.

```
(define (num-common lst1 lst2)
```

(b) [**4**] In **Haskell**, implement `map` in terms of `foldl`, whose definition is given below. You may not use any helper functions for this question, nor explicit recursion or pattern matching. **Hint:** use the cons `(:)` operator.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ i [] = i
foldl f i (x:xs) = foldl f (f i x) xs

map :: (a -> b) -> [a] -> [b]

map f lst =
```

3. (a) **[2]** Explain the difference between **static typing** (used in Haskell) and **dynamic typing** (used in Racket).

(b) **[3]** For each of the following Haskell values, fill in the type that would be inferred by Haskell.

```
x ::

x = if 3 > 4 then [] else [True, False, not True]


f ::

f w y z = if y then (z w) else w
```

(c) **[2]** Define **currying**, and use this idea to explain what the *type* of the expression `(f "Hi")` is (the `f` refers to the one from part (b)).

(d) **[3]** We have seen in Haskell how infinite lists can sometimes, but not always, lead to non-terminating programs. For each of the three situations below, give an example of a *unary* function that takes a list, and fits the description, and **explain why it does**. You may use either built-in functions or define your own.

(1) The function always terminates when given an infinite list.

(2) The function sometimes terminates when given an infinite list.

(3) The function never terminates when given an infinite list.

4. (a) **[2]** Define **generic polymorphism**, and explain how Haskell supports this form of polymorphism. **Give an example in your answer.**

   (b) **[2]** Define **ad hoc polymorphism**, and explain how one non-Haskell language of your choice (like Java) supports this form of polymorphism. **Give an example in your answer.**

   (c) **[2]** Consider the type signature of the plus operator: `(+) :: Num a => a -> a -> a`. Explain the meaning of "`Num a =>`", and be sure to mention how this restricts how `(+)` can be used.

5. Recall from lecture that we saw how to use functions to model mutable state. In this question, you'll do something similar with a familiar problem: parsers. A *parser* is a function that takes in a string and outputs a tuple `(data, rest)`, where `data` is some data that was parsed from the front of the string, and `rest` is the rest of the string remaining to be parsed. Here's a simple example below:

```
type BadParser a = [Char] -> (a, [Char])

-- Return the first character of the string, and then the rest
parseFirst :: BadParser Char
parseFirst (c:rest) = (c, rest)
```

(a) **[1]** What happens when we call `parseFirst ""`?

(b) **[3]** The specified type of `BadParser` is too restrictive; let's fix that by using the `Maybe` type to encode the possibility of failure in the types.

```
data Maybe a = Just a | Nothing
type Parser a = [Char] -> (Maybe a, [Char])
```

Rewrite the definition of `parseFirst` using this new `Parser a` type to avoid the problem from part (a). For this and the next two parts: **in the case of a failure, `rest` should always be the original string.**

```
parseFirst :: Parser Char
```

(c) **[5]** Implement the operator `(+++) :: Parser a -> Parser [a] -> Parser [a]`, which takes two parsers `f` and `g`, and returns a new parser that has the following behaviour:

- Try to apply `f` to a string, and then `g` to the resulting string after applying `f`.
- If both parsers succeed (return `Just` something), then use the cons operator `(:)` to prepend the result of the first parser to the result of the second.
- If either parser fails (i.e., returns `Nothing` as its data), then return `Nothing`.
- It should "short-circuit", so that if the first parser fails, the second parser is never applied.

Sample usage:

```
parseOne :: Parser Int
parseOne "" = (Nothing, "")
parseOne (x:xs) = if x == '1' then (Just 1, xs) else (Nothing, x:xs)

parseTwoList :: Parser [Int]
parseTwoList "" = (Nothing, "")
parseTwoList (x:[]) = (Nothing, x:[])
parseTwoList (_:_:xs) = (Just [5,7], xs)

>> (parseOne +++ parseTwoList) "1abcd"
(Just [1,5,7], "cd")
```

(d) [**4**] Implement the parsing combinator `repeatN :: Int -> Parser a -> Parser [a]`, which takes a non-negative integer $n \geq 0$ and a parser, and tries to apply that parser n times to a string, returning a list of all the data parsed. Return `Nothing` as the data if any of the n parsings fail; return `Just []` as the data if $n = 0$.

You must use the `+++` operator for full credit; if you do not, you will receive a maximum of *two* marks. (This restriction really is meant to simplify your code!)

Sample usage:

```
myParser :: Parser Int
myParser "" = (Nothing, "")
myParser ('a':rest) = (Nothing, 'a':rest)
myParser (_:rest) = (Just 1, rest)

> repeatN 3 myParser "ronald"
(Just [1, 1, 1], "ald")
> repeatN 4 myParser "ronald"
(Nothing, "ronald")
```

6. Now let's switch over to Prolog. In this question, you are required to write Prolog facts and queries to represent each of the following English statements. You are responsible for defining your own predicates; read over *all* the questions carefully, then list your predicates in the space below, with a brief description of the meaning of each one. Then, use your predicates to translate the English statements into Prolog. Note that you are allowed to write more than one line for each question!

   Predicates:

   (a) **[2]** `archer`, `lana`, and `cyril` are spies.

   (b) **[2]** Every spy is an enemy of `barry`.

   (c) **[2]** Every friend of `barry` who is *not* a spy is an enemy of `archer`. (Use `!` and `fail` here; do not use the `\+` operator.)

   (d) **[3]** "The enemy of my enemy is my friend." (If two people share a common enemy, they are friends.) Note: for simplicity, you can be your own friend; no need to check for distinctness here.

7. **[6]** In Prolog, we can represent boolean formulas using the following terms:

   - `myTrue`, `myFalse`: the boolean true and false values.
   - `and(F,G)`, `or(F,G)`: the standard boolean operations, where `F` and `G` are boolean formulas.

   Of course, these may be nested, so the following is a valid Prolog term:

```
and(or(myTrue, and(myFalse, myTrue)), myTrue)
```

   Your goal is to **define a predicate** `eval(F, v)` where `F` is is a valid boolean formula (as above), and `v` is either `myTrue` or `myFalse`. You may assume the input to `eval` is always valid; no error checking is required.

   Sample usage:

```
?- eval(and(or(myTrue, and(myFalse, myTrue)), myTrue), X).
X = myTrue.
```

8. Sometimes in Prolog, we care only about whether there is *some* value that makes the query return true, but we do not care how many values there are. For example:

```
f(a).
f(b).
anyf :- f(X).
```

   (a) **[2]** Unfortunately, this rule does not give us exactly what we want. Carefully explain the output of the following query:

   ```
   ?- anyf.
   true ;
   true.
   ```

   (b) **[2]** Change the definition so that only one answer (either `true` or `false`) is ever printed. *Explain why your change(s) work.*

Next, recall from lecture the following *bad* implementation of a `max(X,Y,Z)` predicate, which succeeds if and only if `Z` is equal to the larger of `X` and `Y` (which you can assume are integers).

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

   (c) **[1]** Give one example of a successful use of this predicate (i.e., a query that does the right thing), in which the cut is used to make this predicate more *efficient*.

   (d) **[2]** Give one example of an unsuccessful query on this predicate, and *explain* what goes wrong.

9. (a) [**4**] Write a Prolog predicate `remove(X, L, N)` that succeeds if and only if `X` is an item in list `L`, and `N` is a list containing the same elements as `L`, except with the first occurrence of `X` removed.

   Sample usage:

```
?- remove(hi, [hello, bye, hi, good, hi], [hello, bye, good, hi]).
true.
```

   (b) [**3**] Using `remove`, write a Prolog predicate `perm(L, M)`, which succeeds if and only if `L` and `M` are lists containing the exact same elements (i.e., if `M` is a permutation of `L`). Your predicate must work correctly when given two instantiated lists, but is allowed to fail when given one or more variables in place of lists.

   Note that you can get full marks for this question even if you don't do part (a)!

10. (a) **[2]** Even though Racket uses eager evaluation, we saw in lecture that we could *simulate* lazy evaluation by wrapping arguments in lambda expressions:

```
(define (my-or x y) (or (x) (y)))
```

State what is output/happens when we execute the following function calls. No explanation necessary.

```
> (my-or #t (/ 1 0))
```

Output:

```
> (my-or (lambda () #t) (lambda () (/ 1 0)))
```

Output:

(b) **[4]** We now implement a simpler version of what I asked on the midterm. Write a macro `lazy-call` which allows a "lazy" calling behaviour without wrapping the argument expressions. Note that your macro should work on *any* "lazy" function defined in the same way as `my-or`, and your macro should work on functions that take in any number of parameters (including no parameters). Sample usage:

```
> (define (my-f x y z) (+ (x) (z)))
> (lazy-call (my-f 3 (/ 1 0) 4))
7
```

Marking notes: you'll get 1 mark for correctly handling a function with no arguments, and 3 marks for correctly handling a function with 1 or more arguments.

Use this page for rough work.

Use this page for rough work.

Use this page for rough work.