Last Name: ____________ **First Name:** ____________

Student #: ____________ **Signature:** ____________

UNIVERSITY OF TORONTO MISSISSAUGA
DECEMBER 2018 FINAL EXAMINATION
CSC324H5F
Principles of Programming Languages
Daniel Zingaro, Lisa Zhang
Duration - 2 hours
Aids: none

The University of Toronto Mississauga and you, as a student, share a commitment to academic integrity. You are reminded that you may be charged with an academic offence for possessing any unauthorized aids during the writing of an exam. Clear, sealable, plastic bags have been provided for all electronic devices with storage, including but not limited to: cell phones, smart devices, tablets, laptops, calculators, and MP3 players. Please turn off all devices, seal them in the bag provided, and place the bag under your desk for the duration of the examination. You will not be able to touch the bag or its contents until the exam is over.

If, during an exam, any of these items are found on your person or in the area of your desk other than in the clear, sealable, plastic bag, you may be charged with an academic offence. A typical penalty for an academic offence may cause you to fail the course.

Please note, once this exam has begun, you **CANNOT** *re-write it.*

You must earn 40% or above on the exam to pass the course; else, your final course mark will be set no higher than 47%.

This final examination consists of 7 questions on 14 pages (including this page), plus an additional aid sheet at the back of the exam. You may detach the aid sheet if you wish, but please do so without removing any other page from the exam. When you receive the signal to start, please make sure that your copy of the examination is complete.

If you need more space for one of your solutions, use the last pages of the exam and indicate clearly the part of your work that should be marked.

Marking Guide

1: ______/10

2: ______/ 5

3: ______/ 6

4: ______/10

5: ______/ 6

6: ______/ 5

7: ______/ 8

Good Luck! TOTAL: ______/50

Question 1. [10 MARKS]

Circle either "True" or "False" for each of the below statements.

1. True False Racket is a dynamically typed language.
2. True False Imperative programming is more declarative than logic programming because we declare more variables.
3. True False Types with multiple value constructors are called "polymorphic".
4. True False The higher-order function `foldl` is more efficiently implemented in Haskell than in Racket.
5. True False In a tail-recursive function, the accumulator should be either a number or a list.
6. True False The higher-order function `foldl` can be written in terms of `map`.
7. True False In Racket, the expressions `(+ 3 1)` and `((+ 3 1))` evaluate to the same value.
8. True False In Haskell, the expressions `(3 + 1)` and `((3 + 1))` evaluate to the same value.
9. True False In the Racket expression `(+ 3 1)`, the continuation of `+` is `_`
10. True False In Racket, to append the value `5` to the end of the list `(list 1 2 3 4)`, we write `(append (list 1 2 3 4) 5)`.

Question 2. [5 marks]

Part (a) [3 marks]

Recall that we can use `cond` in Racket to write conditional statements. However, any code written in terms of `cond` can be rewritten using nested calls to `if`.

```
(define (f x)
  (cond [(< x 3) (+ x 3)]
        [(> x 5) (* x 2)]
        [else    (+ x 1)]))
```

```
(define (f x)
  (if (< x 3)
      (+ x 3)
      (if (> x 5)
          (* x 2)
          (+ x 1))))
```

Complete the following implementation of a macro `my-cond` by filling in one name per box. The arguments of `my-cond` follow the Racket syntax for `cond`; the macro rewrites the expression in terms of equivalent calls to `if`. You may assume that the final condition of `cond` will be an `else`.

```
(define-syntax my-cond
  (syntax-rules ( __________ )
   [(my-cond [else <expr>])
     __________ ]

    [(my-cond [<cond> <expr>] <rest> ...)
     (if __________
         __________
         ( __________ __________ ))]))
```

Part (b) [2 marks]

Why should `my-cond` be defined as a **macro**, rather than a **function**? (Be concise; only the first 10 words of your answer will be graded.)

Question 3. [6 MARKS]

For both parts of this question, please use recursion **directly**; i.e. do not use higher-order list functions such as `map` and `foldl`.

Part (a) [2 MARKS]

Write the Racket function `list-max` that takes a nonempty list of integers, and returns the biggest value in the list. For example:

```
> (list-max '(2 8 1))
8
> (list-max '(1))
1
```

```
(define (list-max lst)
```

Part (b) [4 MARKS]

Write the Racket function `biggest` that takes an argument `lst`, where `lst` is a list and each of its elements is a nonempty list of integers. The function `biggest` returns the sublist in `lst` containing the biggest element. If there are multiple sublists that contain the biggest element, return any one of them.

```
> (biggest '((2 3 6) (8 1)))
'(8 1)
> (biggest '((2 3 6) (8 1) (8 4 3))
'(8 1)   ; or '(8 4 3), either return value is acceptable
```

You may use the helper function `list-max` from part (a), and define any other helper functions you like. However, do not use higher-order list functions such as `map` and `foldl`.

```
(define (biggest lst)
```

Question 4. [10 MARKS]

We would like to define a Haskell function `andmap` that takes a function `f` and a list `lst`, and returns whether `f x` evaluates to true for every `x` in `lst`. Here are some examples:

```
Prelude> andmap (> 3) [4, 5]
True
Prelude> andmap (> 3) []
True
Prelude> andmap (\x -> x) [False, True, False]
False
```

Part (a) [2 MARKS]

The function `andmap` is polymorphic. Complete the type signature of `andmap`.

```
andmap ::
```

Part (b) [4 MARKS]

Write the definition of `andmap` using a single call to `foldl`.

Part (c) [2 MARKS]

What is the type of the variable `mystery` defined below?

```
apply x f = f x
mystery = andmap (apply Nothing)
```

Part (d) [2 MARKS]

Consider a total function `f` in Haskell with the following type.

```
f :: a -> (b -> c) -> [b] -> b -> c
```

Provide an implementation of `f`. Your implementation should **not** use the Haskell value `undefined` or raise an exception.

Question 5. [6 MARKS]

Part (a) [3 MARKS]

Each of the below expressions is executed, in sequence, in a Racket shell. Fill in the output of each expression. If an error occurs, simply write "ERROR" without further explanation.

```
> (define cont null)
> (let/cc c 3)
```


```
> (+ 3 (let/cc c
    (begin (set! cont c)
           1)))
```


```
> (c 5)
```


```
> (cont 5)
```


```
> (+ 1 (cont 5))
```


```
> (cont (cont 5))
```


Part (b) [3 MARKS]

Define an error-raising continuation called `raise`, which takes a string argument. When `raise` is called, it should cause the program to halt and return the string. You may use mutation.

```
> (define denom 0)
> (+ 5 (if (equal? denom 0)
           (raise "The denominator is zero")
           (/ 3 denom)))
"The denominator is zero"
> (* 2 (raise "error"))
"error"
```

Question 6. [5 MARKS]

Consider a list of unique integers in Racket; for example, the list `'(4 6 10)`. We would like to explore subsets of these integers that do not contain integers appearing directly next to each other. For example, if a subset contains `4`, then the subset should not also contain `6`.

Write a function `solution` that uses logic programming to return the first such subset, and makes the remaining possible subsets available through calls to `next`. The order that solutions are produced does not matter; just produce them all.

```
> (solution '(4 6 10))
'()
> (next)
'(4)
> (next)
'(6)
> (next)
'(10)
> (next)
'(4 10)
> (next)
'done
```

```
(define (solution lst)
```

Question 7. [8 MARKS]

Consider the following type declarations:

```
-- type declarations
data Person         = Person String Float      -- name, salary
data Robot          = Robot Int                -- identifier
data Organization p = Individual p             -- organization of one
                    | Team p [Organization p]  -- team leader, and list of sub-orgs

-- example:
owner   = Person "Janet" 100000
cto     = Person "Larry"  90000
cfo     = Person "Mike"   90000
intern  = Person "Sam"    40000
company = Team owner [Team cto [Individual intern],
                      Individual cfo]

-- robot organization:
robot1   = Robot 1
robotOrg = Individual robot1
```

Part (a) [2 MARKS]

We discussed how value constructors are functions in Haskell. What are the type signatures of each value constructor created above? If the name on the left of `::` is not a value constructor, write "Not a constructor".

`Person ::`

`Organization ::`

`Individual ::`

`Team ::`

Part (b) [4 MARKS]

Recall that a **functor** is a type class that supports mapping. Write code so that the type constructor `Organization` is an instance of the type class `Functor`, where a given function would be mapped over every `p` in the `Organization`.

```
instance Functor Organization where
```

Part (c) [1 MARK]

Consider the function `applyAnnualRaise`, which increases a `Person`'s salary by a fixed 3%. Use a call to `fmap` to apply this function to everyone in the organization `company` (defined at the beginning of this question). Save the result in the variable called `companyWithRaise`.

```
applyAnnualRaise :: Person -> Person
applyAnnualRaise (Person name salary) = Person name (salary * 1.03)


companyWithRaise =
```

Part (d) [1 MARK]

Consider the function `robotize`, which replaces a `Person` with a `Robot`. Use a call to `fmap` to turn `company` (defined at the beginning of this question) into an organization with the same structure, but populated entirely by robots. Save the result in the variable called `robotCompany`.

```
robotize :: Person -> Robot
robotize p = Robot 0


robotCompany =
```

[Use the space below for rough work. This page will ***not*** *be marked, unless you clearly indicate the part of your work that you want us to mark.]*

*[Use the space below for rough work. This page will **not** be marked, unless you clearly indicate the part of your work that you want us to mark.]*

*[Use the space below for rough work. This page will **not** be marked, unless you clearly indicate the part of your work that you want us to mark.]*