Last Name: _____ **First Name:** _____

Student #: |_|_|_|_|_|_|_|_|_| **Signature:** _____

UNIVERSITY OF TORONTO MISSISSAUGA
DECEMBER 2019 FINAL EXAMINATION
CSC324H5F
Principles of Programming Languages
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 8 questions on 14 pages, including this page, and including aid sheets 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 one of the last pages of the exam and indicate clearly the part of your work that should be marked.

Good Luck!

MARKING GUIDE

# 1:	_____/ 8
# 2:	_____/ 9
# 3:	_____/ 6
# 4:	_____/ 6
# 5:	_____/ 7
# 6:	_____/ 6
# 7:	_____/ 4
# 8:	_____/ 4
TOTAL:	_____/50

Racket

```
; Function definition and application
(lambda (x y) (* x y))
(lambda () 16)

(+ 3 4 5)                   ; 12
(equal? 3 (- 4 25))         ; #f
((lambda (x) (+ 3 x)) 10)   ; 13

; Name bindings
(define x 10)
(define (f z) (first (rest z)))
(define f2 (lambda (z) (first (rest z))))
(let* ([y (+ 10 20)]
       [z (+ 10 y)])
  (* y z))

; Syntactic forms
(and #f (/ 1 0))      ; #f
(or #t (/ 1 0))       ; #t
(if #t 42 (/ 1 0))    ; 42
(cond [#f (/ 1 0)]
      [#t 42]
      [(/ 1 0) 25]
      [else 1])       ; 42

; Pattern matching
(define/match (comment x)
  [(7) "Lucky"]
  [(13) "Unlucky"]
  [(_) "Other"])

(define/match (f lst)
  [((list)) 100]
  [((cons x xs)) (+ x (length xs))])

; Lists
(cons 2 (cons 3 null))      ; '(2 3)
(list 1 2 20)               ; '(1 2 20)
(range 0 5)                 ; '(0 1 2 3 4)
(first (list 1 2 3))        ; 1
(rest (list 1 2 3))         ; '(2 3)
(null? null)                ; #t
(length (list 1 20 5))      ; 3
(append (list 1 2 3)
        (list 4 5 6))       ; '(1 2 3 4 5 6)
(member 2 (list 1 2 3))     ; '(2 3)
(member 4 (list 1 2 3))     ; #f
(list-ref (list 2 40 1) 1)  ; 40
(take (list 1 2 3 4) 2)     ; '(1 2)
(drop (list 1 2 3 4) 2)     ; '(3 4)

(map (lambda (x) (* 3 x))
     (list 1 2 3))          ; '(3 6 9)
(filter (lambda (x) (< 3 x))
        (list 10 -4 15))    ; '(10 15)
(foldl + 15 (list 1 2 3)) ; (3 + (2 + (1 + 15)))
(foldr + 15 (list 1 2 3)) ; (1 + (2 + (3 + 15)))
(apply - (list 16 3))     ; 13
```

Haskell

```
-- Function definition and application
\x -> x + x
\x y -> x * y

max 3 4   -- 4
3 + 4     -- 7
(==) 3 4  -- False

-- Name bindings
x = 10
f z = z + 10
f2 = \z -> z + 10
let y = 20 + 10
    z = y + 10
in  y * z

-- If (note that && and || are functions)
f x =
  if x == 10
  then 16
  else -20




-- Pattern matching
comment 7 = "Lucky"
comment 13 = "Unlucky"
comment _ = "Other"


f [] = 100
f (x:xs) = x + length xs


-- Lists
2:3:[]                      -- [2,3]
[1,2,20]                    -- [1,2,20]
[0..4]                      -- [0,1,2,3,4]
head [1,2,3]                -- 1
tail [1,2,3]                -- [2,3]
null []                     -- True
length [1,2,20]             -- 3
[1,2] ++ [4,6]              -- [1,2,4,6]

elem 2 [1,2,3]              -- True
elem 4 [1,2,3]              -- False
[20,40,1] !! 1              -- 40
take 2 [1,2,3,4]            -- [1,2]
drop 2 [1,2,3,4]            -- [3,4]

map (\x -> 3 * x) [1,2,3]        -- [3,6,9]

filter (\x -> 3 < x) [10,-4,15]  -- [10,15]

foldl (+) 15 [1,2,3]  -- (((15 + 1) + 2) + 3)
foldr (+) 15 [1,2,3]  -- (1 + (2 + (3 + 15)))
```


CONT'D...

Streams, Continuations, and Choice

```
; Streams
(define s-null 's-null)
(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))
(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))

; The ambiguous choice operator
(define (-< . lst)
  (shift k (s-append-map k lst)))
(define (s-append-map k lst)
  (if (empty? lst)
      s-null
      (s-append
        (k (first lst))
        (thunk (s-append-map k (rest lst))))))
(define (s-append s t)
  (cond
    [(s-null? s) (t)]
    [(pair? s)
     (s-cons (s-first s)
             (s-append (s-rest s) t))]
    [else (s-cons s (t))]))

(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>)
     (thunk (reset (singleton <expr>)))]))
(define (singleton x) (make-stream x))

; Self-updating stream
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)  ; <g> is a thunk that
                  ; evaluates to a stream
     (let* ([stream (<g>)]) ; evaluate <g>
       (if (s-null? stream)
           'DONE
           (begin
               (set! <g> (cdr stream))
               (s-first stream))))]))

; Backtracking
(define (fail) (shift k s-null))
```

Haskell Types

```
-- Type annotations
True :: Bool
(&&) :: Bool -> Bool -> Bool
head :: [a] -> a
map :: (a -> b) -> [a] -> [b]
(==) :: Eq a => a -> a -> Bool
1 :: Num a => a
(+) :: Num a => a -> a -> a

-- Type declarations, type synonyms
data Point = Point Int Int
data Tree a = Empty | Tree a (Tree a) (Tree a)
type String = [Char]

-- Typeclass instantiation
instance Show Point where
  show (Point x y) =
    "(" ++ (show x) ++ ", " ++ (show y) ++ ")"

-- Functors and monads
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a

-- Modeling failures
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

-- Modeling mutation
data State s a = State (s -> (a, s))

get :: State s s
get = State (\state -> (state, state))

put :: s -> State s ()
put x = State (\_ -> ((), x))

runState :: State s a -> s -> (a, s)
runState (State f) init = f init

-- NOTE: Maybe, (Either a), (State s), and IO
-- are all instances of Functor and Monad.
```