CSC 324H5 F 2019 Midterm
Duration — 50 minutes
Aids allowed: none

⓪	⓪	⓪	⓪	⓪	⓪	⓪	⓪	⓪	⓪
①	①	①	①	①	①	①	①	①	①
②	②	②	②	②	②	②	②	②	②
③	③	③	③	③	③	③	③	③	③
④	④	④	④	④	④	④	④	④	④
⑤	⑤	⑤	⑤	⑤	⑤	⑤	⑤	⑤	⑤
⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥	⑥
⑦	⑦	⑦	⑦	⑦	⑦	⑦	⑦	⑦	⑦
⑧	⑧	⑧	⑧	⑧	⑧	⑧	⑧	⑧	⑧
⑨	⑨	⑨	⑨	⑨	⑨	⑨	⑨	⑨	⑨

Student Number:

UTORid: ___|___|___|___|___|___|___|___

Last Name: ________________________ **First Name:** ________________________

Lecture Section: L0101 Test Version: A Instructor: Lisa Zhang

*Do **not** turn this page until you have received the signal to start.*
(Please fill out the identification section above, and read the instructions below.)
Good Luck!

This test consists of 5 questions on 8 pages (including this page). *When you receive the signal to start, please make sure that your copy is complete.*
Comments are not required except where indicated, although they may help us mark your answers. They may also get you part marks if you can't figure out how to write the code.
If you use any space for rough work, indicate clearly what you want marked.

1: ______/ 7

2: ______/ 4

3: ______/ 4

4: ______/ 5

5: ______/ 5

TOTAL: ______/25

Total Pages = 8

Question 1. [7 marks]

Part (a) [2 marks]

Why is the following Haskell implementation of `foldl` inefficient?

```
foldl _ acc [] = acc
foldl f acc (x:xs) =
  let acc’ = f acc x
  in
      foldl f acc’ xs
```

Part (b) [3 marks]

What do the following Racket expressions evaluate to? If there is an error, explain why.

```
> (define (f x) (lambda () (x)))
> (f 1)
```

```
> ((f 1))
```

```
> ((f (lambda (x) 3)))
```

Part (c) [2 marks]

What do the following Haskell expressions evaluate to? If there is an error, explain why.

```
Prelude> g x y = if x then x else y
Prelude> g True
```

```
Prelude> ((g False False))
```

Question 2. [4 marks]

Part (a) [2 marks]

Consider the calculator grammar from Exercise 4:

```
<expr> ::= NUM                  ;; integer in base 10
         | ID                   ;; variable names (excluding +, -, *, /, =, >, <)
         | (<op> <expr> <expr>)
         | (if (<comp> <expr> <expr>) <expr> <expr>)
         | (let* ((ID <expr>) ...) <expr>)
         | (lambda (ID ...) <expr>)
         | (<expr> <expr> ...)
<comp> ::= = | > | <
<op>   ::= + | - | * | /
```

Cross out any of the four expressions below that are **not syntactically valid in this grammar**.

- `(lambda (a b) (lambda (a b) (a b)))`
- `y`
- `(let* ((x (> 1 1)) (if x 1 0)))`
- `(= 1 1)`

Part (b) [2 marks]

Provide an example expression in the calculator grammar whose behaviour would differ depending on whether we used lexical or dynamic scoping.

Question 3. [4 marks]

Part (a) [2 marks]

Consider the following macro:

```
(define-syntax my-macro
  (syntax-rules ()
    [(my-macro (<x>) <y> ...) (list 'a <x>)]
    [(my-macro (<x>) ...)     (list 'b <x> ...)]
    [(my-macro (<x> <y> ...)) (list 'c <y> ...)]))
```

Perform macro expansion on the following two expressions. Write “ERROR” without further explanation if there is an error.

```
(my-macro (1) (2) (3))
```

```
(my-macro (4))
```

Part (b) [2 marks]

Consider the `my-class` macro in the aid sheet. What does the `(method)` in the second line do? Give an example call to `my-class` that would result in an undesirable behaviour if we replaced `(method)` with `()`.

```
(define-syntax my-class
  (syntax-rules (method)
...))
```

Question 4. [5 marks]

Recall that we can use `let*` in Racket for local variable binding. However, any code written in terms of `let*` can be rewritten using nested `lambda` definitions.

```
(let* ((a 3)
       (b 4))
    (+ a b))
```

```
((lambda (a)
  ((lambda (b) (+ a b))
   4)
 3)
```

Complete the following implementation of a macro `my-let*`. The arguments of `my-let*` follow the Racket syntax for `let*`; the macro rewrites the expression in terms of the equivalent `lambda` definitions.

```
(define-syntax my-let*
  (syntax-rules ()
   [(my-let* () <body>)
                                          ]

    [(my-let* ((<name> <expr>) <rest> ...) <body>)
     ((lambda 
        (my-let*                                        ))
       ))]))
```

Question 5. [5 MARKS]

Write a function `split` in Racket that splits a list into two, where the list elements alternate. For example:

```
> (split '(1 2 3 4 5))
'((1 3 5) (2 4))
> (split '())
'(() ())
```

For full marks, use tail recursion. A non-tail recursive solution can earn up to 3 points. You may write as many helper functions as you need, and use any of the list functions in the aid sheet.

[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.]