UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm CSC324H1
Duration: 50 minutes Instructor(s): David Liu. No Aids Allowed

Name:

Student Number:

Please read the following guidelines carefully.

- Please print your name and student number on the front of the exam.
- This examination has 4 questions. There are a total of 8 pages, DOUBLE-SIDED.
- You may generally use helper functions, explicit recursion, higher-order functions, pattern-matching, and do notation, unless the question specifies otherwise.
- Documentation is not required unless asked for.
- Answer questions clearly and completely. Provide justification unless explicitly asked not to.

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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Grade</th>
<th>Out of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Q2</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Q4</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>
1. **[8 marks] Short answer.**

(a) **[4 marks]** You are given the following Racket definitions.

```
(define (f x)
  (lambda (y) (* x y)))
(define g (f 10))
```

For each of the following Racket code snippets, state what value would be output, or briefly describe what error would be raised, when the snippet is evaluated.

(i) `g`

(ii) `(g 3)`

(iii) `((g 3))`

(iv) `(let ([x 100])
       (g 3))`
(b) [2 marks] Consider the following Racket function.

```racket
(define (count-evens numbers)
  (if (null? numbers)
    0
    (if (even? (first numbers))
      (+ 1 (count-evens (rest numbers)))
      (count-evens (rest numbers)))))
```

Is this function tail-recursive? Explain your answer.

(c) [2 marks] Consider the following Haskell function.

```haskell
f 0 x = x
f 1 x = 0
f n x = f (n - 2) (x + 4)
```

When we evaluate `f 1000000 0` in the interpreter (ghci), a very large amount of memory is used. Explain.

2. [8 marks] Functional programming. Consider the following description of a function `sequence`.

```plaintext
#| (sequence functions input)
  | Given a list of unary functions [f1, f2, f3, ... f-k] and input x,
  | returns the value of (f-k (f-{-k-1} ... (f2 (f1 x)) ... )).
  | Returns `input` itself if the list of functions is empty.
|#
; Example:
(sequence (list (lambda (x) (+ x 1)) (lambda (x) (* x 3)) (lambda (x) (- 100 x)))
  4)
; Equals 85: (- 100 (* (+ 4 1) 3))
```
Midterm, CSC324H1

(a) [4 marks] Implement sequence in Racket or Haskell using explicit recursion. (Don’t do both; only the first implementation will be graded.) Do not define any helper functions, and do not use any list functions that aren’t found on the aid sheet.

(b) [4 marks] Implement sequence in Racket or Haskell without explicit recursion, and instead using one or more higher-order list functions (e.g., map, filter, foldl).
3. **[4 marks] Short answer (macros).** Consider the following Racket macro.

```racket
(define-syntax my-mac
  (syntax-rules ()
    [(my-mac <a> (<b> ...))
      (define (<a> f)
        (cond
          [(f <b>) <b> ...]
          [else (error "None")]]))])
```

(a) **[2 marks]** In the space below, give an example use of `my-mac` so that below it, the expression `(my-f even?)` evaluates to 4.

; YOUR MACRO EXPRESSION GOES HERE.

```racket
(my-f even?) ; After evaluating your macro expression, this line should evaluate to 4.
```

(b) **[2 marks]** We have seen in the course that macros can be used to avoid the eager evaluation semantics of function calls. Write a Racket code snippet that illustrates *short-circuiting behaviour* of `my-mac`. Also, *briefly explain* why your code illustrates that behaviour.
4. [6 marks] **Class macro.** The macro `my-class-constraints` behaves similarly to `my-class` (on the aid sheet), except it supports runtime checks on values passed to the constructor, raising an error if a check is violated.

```scheme
(my-class-constraints Point
  ; A point has two attributes, x and y, that must both be integers.
  ; Note that 'integer?' is a built-in predicate.
  ((x integer?)
    (y integer?))

  ; The syntax for methods is the same as on the aid sheet.
  ...
)
```

```scheme
> (define p1 (Point 2 3)) ; p1 behaves exactly the same as in the original macro.
> (define p2 (Point "hello" 3)) ; Calling 'Point' here raises an error.
Error: Contract violation in constructor
```

(a) [2 marks] Give an example use of `my-class-constraints` to create a class `Person` that has *no methods* and *two attributes*, `name` and `age`. This class enforces the following constraints when its constructor is called:

- A person’s name is a string (use `string?`).
- A person’s age is a non-negative integer.
(b) [2 marks] Complete the macro pattern for \texttt{my-class-constraints}. Your pattern should match zero or more attributes; \textit{every} attribute must be paired with an expression representing a predicate.

\begin{verbatim}
(define-syntax my-class-constraints
  (syntax-rules (method)
    [(my-class-constraints <Class>
      ; (non-function) attributes
      ; YOUR CHANGES GO HERE!

    ; methods -- Don't change this part.
    (method (<name> <params> ...) <body>) ...)
\end{verbatim}

(c) [2 marks] Write the macro template (i.e., what the macro expands into) to implement the required behaviour for \texttt{my-class-constraints}.

\textbf{Important}: in the \texttt{my-class} macro found on the aid sheet, refer to the entire \texttt{(let ([class dict ...]) ...)} nested under \texttt{(define (<Class> <attr> ...))} expression as \texttt{LET-EXPR}. You may not modify anything in \texttt{LET-EXPR} in your new template; instead, write \texttt{"LET-EXPR"} in your new template to refer to this part (so that you don’t need to rewrite the entire thing).

\begin{verbatim}
; Write your template here.
; Your solution should be quite short. Write "LET-EXPR" to re-use most of the
; original macro template.
; HINT: `and` and `or` take an arbitrary number of arguments.
\end{verbatim}