Name:
Student Number:

Please read the following guidelines carefully!

- Please write your name on the front and back of the exam.
- This examination has 4 questions. There are a total of 6 pages, DOUBLE-SIDED.
- You may always write helper functions unless explicitly asked not to. You may not use mutation or any Racket iterative constructs.
- Any question you leave blank, or where you clearly cross out your work and write “I don’t know,” is worth 10% of the marks.

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.

Good luck!
1. [5] Consider the Racket function `double-all`, which takes a list, and returns a new list consisting of the items in the original list, but with each item repeated twice. The copies of each item should appear in the same order as the original list.

```racket
> (double-all '(2 3 4 5))
'(2 2 3 3 4 4 5 5)
> ; edge cases
> (double-all '())
'(())
> (double-all '(2))
'(2 2)
```

Implement this function *without* using explicit recursion, but instead using higher-order list functions. (Solutions using explicit recursion will be awarded at most 3 marks.) You are encouraged to define your own helper functions. **The only list functions you may use are the ones found on the aid sheet.**

**Solution:**

```racket
(define (double-all lst)
  (foldl (lambda (item acc) (append acc (list item item)))
         '()
         lst))

; Or, using the fact that append can take an arbitrary number of lists
(define (double-all lst)
  (apply append
           (map (lambda (x) (list x x))
                lst)))
```
2. (a) [2] Consider the following Racket function call expression, which contains other function call subexpressions:

\( (a \ b \ (c \ b) \ (d \ (e \ f \ g) \ h \ (i)) \ (j \ k)) \)

Using what you know about evaluation order in Racket, state the order in which the function calls in these expressions are evaluated. You may list either the functions which are called (e.g., “c”), or the entire expression (e.g., “(c b)”) in your answer.

Solution: c, e, i, d, j, a

(b) [3] Consider the following function definition.

\( (\text{define } f \ (\lambda () 10)) \)

State the output of each of the following expressions in Racket. No explanation necessary.

> f

#<procedure:f>

> (f)

10

> ((f))

error: not a procedure

(c) [2] Define a closure.

Solution: A closure is a pointer to a function together with a map of name-value bindings of the free identifiers in the function body.

(d) [2] Explain how you know that \texttt{lambda} is not a function. What would go wrong if we tried to define our own “higher-order function” as follows?

\( (\text{define } (\text{my-lambda} \ \texttt{args} \ \texttt{body})
\quad (\lambda \ \texttt{args} \ \texttt{body})) \)

Solution: we’d get unbound identifier errors in both the parameters we try to define and the body of a function if it refers to those parameters. This is because eager evaluation (which Racket functions all follow) attempts to evaluate all of a function’s arguments, which will fail if there are unbound identifiers.
3. Consider the following Racket macro:

   (define-syntax mymac
     (syntax-rules ()
       [(mymac) 0]
       [(mymac <x> <y> ...) (<x> (mymac <y> ...))])))

(a) [4] Give an example of a correct use of the macro with at least two different arguments after the `mymac`. Show three things: (1) the macro expression `(mymac something ...)`, (2) what the expression expands into, and (3) what the expanded expression evaluates to.

   Solution: (many possible)
   
   `(mymac sqr (lambda (x) (+ x 2)))
   ; Expands into
   `(sqr ((lambda (x) (+ x 2)) 0))
   ; Evaluates to
   4

(b) [2] Name one similarity and one difference between Racket macros and functions.

   Solution:
   
   • Macros and functions both operate by substitution of expressions/values into other expressions.
   • For the macros we’ve seen in lecture, their name must appear as the first identifier inside an expression, just like a function. That is, we signify their use in an expression in the same way. (This isn’t true of all macros, but an acceptable answer given what you’ve seen in this course.)
   • Macros do not evaluate expressions before substitution; functions do.
   • Macros are expanded before the program is run; functions are only called during runtime.
   • Macros can contain literals in their expressions; functions can’t.
   • Macros can expand into multiple definitions/expressions; functions just return a single value.
   • Macros can expand into a definition that can be used in the rest of the program; `define` expressions in a function body are local to that body.

(c) [2] Consider this proposed shortcut for the class macro to automatically define a basic Point class:

   (define-syntax Point-class
     (syntax-rules ()
       [(Point-class)
        (define (Point x y)
          (lambda (msg)
            (cond [(equal? msg "x") x]
                  [(equal? msg "y") y]
                  [else "Attribute error!"])))])

   Why can’t we use this macro to do what we want?

   Solution: because Racket macros are hygienic (i.e., obey lexical scoping rules), any local identifiers inside the macro aren’t accessible from outside the macro. Even if we use the macro, there won’t be a `Point` function available for use. That is, the `define` won’t actually bind a name called `Point` which is accessible to code outside the macro.
4. (a) [3] Consider the simplified `class` macro provided on the aid sheet.

We want to extend this macro to `class-private`, which allows the user to follow the public attributes with some “private attributes”: values set by the constructor which are accessible to the methods, but not from outside the class.

Example of the required syntax:

```scheme
(class-private A (a) (private b c)
  [(f x) (+ x (* a b))])

> (define obj (A 10 20 30))
> ((obj "f") 3)
203
> (obj "b")
"Unrecognized message!"
```

Note that `private` is a literal keyword which must appear before the private attributes.

In the space below, define an extended class macro to enable this syntax. You should re-use most or all of the code we provided above. You may assume there is always at least one private attribute, and there aren’t any name conflicts. There can be any number of public methods and attributes, same as the regular class macro.

Please add comments to identify what code you’ve added/changed.

```scheme
(define-syntax class-private
  ; Need to make "private" a literal keyword
  (syntax-rules (private)
    [(class <Class> (<attr> ...) (private <pattr> ...))
      ; Need to add an extra block of private attributes
      (define (<Class> <attr> ... <pattr> ...) ; Only need to change the signature of the constructor
        (lambda (msg)
          (cond [(equal? msg (id->string <attr>)) <attr>]
                ... [(equal? msg (id->string <method>))
                  (lambda (<param> ...) <body>)]
                ...
                [else "Unrecognized message!"]))))))
```
(b) [3] Suppose we wanted to allow a user to mix the order of private and public attributes. We have partially defined a helper macro which can take a list of expressions, and extract the identifiers paired with the literal keyword `private`.

```scheme
(extract-private (a (private b) c (private d)) ())
; ==> (macro transformation)
(b d)
```

Notice that this macro takes a second argument, which we recommend you use as an accumulator. Complete the macro by filling in the recursive rules.

```scheme
(define-syntax extract-private
  (syntax-rules (private)
    [(extract-private () <acc>) <acc>]
    [(extract-private ((private <id>) <other-ids> ...) (<acc-ids> ...))
      (extract-private (<other-ids> ...) (<acc-ids> ... <id>))]
    [(extract-private (<non-private> <other-ids> ...) (<acc-ids> ...))
      (extract-private (<other-ids> ...) (<acc-ids> ...))]
    ])
```