Question 1. [7 marks]

Part (a) [2 marks]

In Racket, can `if` expressions of the form `(if <cond> <then> <else>)` be implemented as a function? Why or why not?

Answer: No. In an if expression, only one of the `<then>` and `<else>` expressions are evaluted. However, since functions in Racket is eagerly evaluated,

Marking:

Please note that for this question, it is not enough to have the right idea in your head. We are only grading based on what you have written on the test sheet.

- **0 points** Answer is blank, incorrect, or not understandable by the grader.
- **0.5 points** The student probably has the right answer, but the grader is not certain due to the writing/legibility of the solution.
- **1 points** The student probably has the right answer, but the grader is not certain due to the writing/legibility of the solution.
- **1.5 points** Answer discusses eager evaluation in general ¡OR¿ answer mixes up macros and functions.
- **2 points** Clear answer that discusses eager evaluation of both ¡then¿ and ¡else¿ expr.

Part (b) [3 marks]

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

Answer:

```
> (define (f x) (lambda (x) x))
> (f 1)
#<procedure>
> (f 1 2)
Error because f only takes one argument
> ((f 1) 2)
2
```

Marking:

For `(f 1)`

- **0 points** for blank or incorrect answer
- **0.5 points** for a solution that states there should be an error because (f 1) is a procedure.
- **1 points** for a correct answer.

For `(f 1 2)`

- **0 points** for blank or incorrect answer
- **0.5 points** for a solution that states there should be an error, but for the wrong reason.
- **1 points** for a correct answer.

For `((f 1) 2)`

- **0 points** for blank or incorrect answer
- **0.5 points** if the student wrote "Error" and student thought (f 1) returns 1.
- **1 points** for a correct answer.

Part (c) [2 marks]

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

Answer:

```
Prelude> g x y = x
Prelude> g 2
A procedure, which Haskell cannot display
Prelude> g 4 (1 / 0)
4
```

Marking:

For `g 2`

- **0 points** for blank or incorrect answer
- **0.5 points** if the solution states currying doesn't happen unless we store the procedure.
- **1 points** for a correct answer. It is ok to write "procedure" or "error because Haskell can't print procedures"

There are no part marks for `g 4 (1 / 0)`

Question 2. [3 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**.

Answer:

- ~~(lambda (x) (= x (+ 3 4)))~~
- ~~()~~
- `(if (= (lambda (x) x) (lambda (y) y)) 3 4)`
- `(3 4 5)`

Marking: 0.5 points for each correct answer

Note that we are looking at syntactic correctness only. The first two of these expression can be generated by the grammar. The latter two cannot be generated by the grammer.

Part (b) [1 mark]

What is a closure?

Answer: A closure is a data structure that stores the function definition (paramaters and body) and the environment at the time the function is defined.

Marking:

- 0.5 points for storing the information about a function
- 0.5 points for storing about the environment *at the time the function is defined*

Question 3. [5 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.

Answer:

```
(my-macro (1))

==> (list 'a 1)

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

==> (list 'c (2) 3)
```

Marking: 1 point for each correct answer. If student adds extra brackets, take off half point for each occurence. (e.g. `(list 'a (1))` instead of `(list 'a 1)`)

Part (b) [3 marks]

Create a class `Pet` with the attribute `name` and `age`, and a method `birthday` that returns a new `Pet` with the same name, and with the `age` incremented by one.

Answer:

```
(my-class (Pet name age)
  (method (birthday self age) (Pet (self 'name) (+ 1 (self 'age)))))
```

Marking:

- **1 point** for using the correct syntax to define the class name, attributes, method name, and method arguments. Take half point off for each syntactical issue.
- **1 point** for correctly using the `self` procedure. Half point for minor syntactical issues.
- **1 point** for correctly using the `Pet` constructor. Half point for minor syntactical issues.

Question 4. [5 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`. 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`.

Answer:

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

$\boxed{\text{¡expr¿}}$]

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

(if $\boxed{< cond >}$

$\boxed{< expr >}$

($\boxed{my - cond}$ $\boxed{< rest > ...}$))]))

Marking:

- 1 point per box. Please note that the `...` is required!
- If the student gets the right answer, but not in the exact right box, that's ok.
- 0.5 points for adding unnecessary brackets.

Question 5. [5 marks]

Write a function `merge` in Racket that merges two sorted lists. For example:

```
> (merge '(1 4 5) '(2 4))
'(1 2 4 4 5)
> (merge '(1 4 5) '())
'(1 4 5)
> (merge '() '(5))
'(5)
```

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.

```
(define (merge l1 l2)
  (merge-helper l1 l2 '()))
(define (merge-helper l1 l2 acc)
 (cond
   [(empty? l1)  (append (reverse acc) l2)]
   [(empty? l2)  (append (reverse acc) l1)]
   [(< (first l1) (first l2))
    (merge-helper (rest l1) l2 (cons (first l1) acc))]
   [else
    (merge-helper l1 (rest l2) (cons (first l2) acc))]))
```

Non-tail recursive solution (max 3 points):

```
(define/match (merge l1 l2)
 [((list) l2) l2]
 [(l1 (list)) l1]
 [((cons e1 rest1) (cons e2 rest2))
  (if (< e1 e2)
      (cons e1 (merge rest1 l2))
      (cons e2 (merge l1 rest2)))])
```

Marking:

- 0.5 points for correctly handling the base case where both lists are empty
- 0.5 points for correctly handling the base case where one list is empty
- 1.0 points for having the correct recursive call structure and values
- 1.0 points for having the correct racket syntax
- 1.0 points for using tail recursion
- 1.0 points for having the correct ordering of the list elements

Note that functions like `drop` and `take` are recursive. If you are calling functions like these in your recursive calls, then the resulting function is *not* tail-recursive.