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
```

Answer: Since Haskell uses lazy evaluation, the expression in the assignment `acc' = f acc x` is not evaluated until we reach the end of the list. In the meantime, the unevaluated expressions take up unnecessarily large amount of storage space.

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. (Tail recursion is incorrect)
- **0.5 points** Answer only discusses the accumulator update.
- **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 lazy evaluation in general, but not specific to this example
- **2 points** Clear answer that discusses lazy evaluation, and the evaluation of accumulator update.

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)))
> (f 1)
#<procedure>
> ((f 1))
Error, because 1 is not a function.
> ((f (lambda (x) 3)))
Error, because we are calling (lambda (x) 3) with the wrong number of arguments.
```

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. It is true the (f 1) returns a procedure, but no error is produced.
- **1 points** for a solution that indicates that (f 1) should be a procedure.

For `((f 1))`

- **0 points** for blank or incorrect answer
- **0.5 points** for a solution that states there is an error, *or* student writes 1.
- **1 points** for a solution that indicates that there is an error because 1 is not a function.

For `((f (lambda (x) 3)))`

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

Part (c) [2 marks]

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

Answer:

```
Prelude> g x y = if x then x else y
Prelude> g True
A procedure, which Haskell cannot display
Prelude> ((g False False))
False
```

Marking:

No part marks for either part of this question.

For part 1, we accepted any of "procedure", `#<procedure>`, or "Error because Haskell can't display a procedure". For the second expression, saying that there is an error is worth 0 points.

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**.

Answer:

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

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) [2 marks]

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

Answer:

```
(let* ((n 100)
       (f (lambda () (n)))
       (g (lambda (n) (f))))
    (g 10))
```

Marking:

- **0 points** for blank or incorrect answer
- **0.5 points** for an example that does not illustrate dynamic scoping, but is syntactically valid.
- **1.5 points** for an example that illustrates dynamic scoping, but has some syntactic incorrectness. We are ignoring non-matching trailing brackets.

- **2 points** for an example that illustrates dynamic scoping and is syntactically valid. We are ignoring non-matching trailing brackets.

A number of people provided examples of variable shadowing, which is different from lexical vs dynamic scoping.

The example should have a function call where the environment *at the time the function is created* has different variable bindings compared to the environment *at the time the function is called*.

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.

Answer:

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

==> (list 'a 1)

(my-macro (4))

==> (list 'a 4)
```

Marking: 1 point for each correct answer. If student adds extra brackets, take off half point for each occurence. (e.g. `(list 'a (4))` instead of `(list 'a 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)
...))
```

Answer: The `(method)` means that the identifier `method` is a keyword, and not a macro variable. If this code is replaced with `()` then calls like this to the `my-class` macro would *succeed*, even though they shouldn't:

```
(my-class (Foo x y)
  (foo (get-x self) (self 'x)))
```

Marking:

- **0 points** for an incorrect or blank answer
- **0.5 points** for an answer that is on the right track
- **1 points** for an answer that is ambiguous, with no/poor example.
- **1.5 points** for either an answer that is clear that missing a correct example, or an answer that is unclear but the example is excellent.
- **2 points** for a clear answer with a clear example.

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.

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

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

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 missing/adding unnecessary brackets.

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.

Answer:

```
(define (split lst)
    (split-helper lst '() '()))

(define (split-helper lst a1 a2)
    (if (<= (length lst) 1)
        (list (reverse (append lst a1))
              (reverse a2))
        (split-helper (rest (rest lst))
                      (cons (first lst) a1)
                      (cons (second lst) a1))))
```

Non-tail recursive solution (max 3 points):

```
(define/match (split lst)
   [((list)) '(() ())]
   [((cons a (list)))
    (list (list a) '())]
   [((cons a (cons b rest)))
    (let* ([result (split rest)]
           [result1 (first result)]
           [result2 (second result)])
        (list (cons a result1)
              (cons b result2)))])
    (list (list a) '())]
```

Marking:

- 0.5 points for correctly handling the base case where the list is empty
- 0.5 points for correctly handling the base case where the list has a single element
- 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 lists

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.