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>8</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>26</td>
<td></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

```
Solution
#<procedure:g>
```

(ii) (g 3)

```
Solution
30
```

(iii) ((g 3))

```
Solution
Error in application (not a procedure). (g 3) evaluates to 30, which is a number, and cannot be called.
```

(iv) (let ([x 100])
     (g 3))

```
Solution
30
```
(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.

**Solution**

This function is not tail-recursive. The first recursive call to `count-evens` is not in tail position (it’s nested inside the `(+ 1 ...)`).

(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.

**Solution**

Because Haskell uses non-strict evaluation, the accumulator parameter `x` doesn’t actually get evaluated until the base case (`n = 0`) is reached. What is accumulated through the recursive calls is the computation to calculate `x (0 + 4 + 4 + 4 + 4 + ... + 4)`, which takes up an amount of memory that’s linear in the depth of the recursion.

Note: `f` is actually tail-recursive, and Haskell does do tail-call optimization! This isn’t the issue with the memory usage of this function.

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

```haskell
#| (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))
```
(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.

Solution

; Using `if`
(define (sequence functions input)
  (if (null? functions)
      input
      (sequence (rest functions)
                 ((first functions) input)))))

; Using pattern-matching
(define/match (sequence functions input)
  [([list] _) input]
  [([cons first-f rest-f] _) (sequence rest-f (first-f input))])

(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).

Solution

(define (sequence functions input)
  (foldl (lambda (f acc) (f acc))
         input
         functions))

Please do not write below this line. There is extra space at the back of the test paper.
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.

**Solution**

```racket
; YOUR MACRO EXPRESSION GOES HERE.

(my-mac my-f (4))

(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.

**Solution**

```racket
(my-mac my-d (3 (/ 1 0) (/ 2 0))) ; This by itself illustrates short-circuiting.
; Even though argument subexpressions raise errors,
; the function `my-d` gets defined; none of the
; subexpressions get evaluated at this line.

(my-d odd?) ; If you wanted to be even more sure, calling `my-d`
; here confirms the short-circuiting behaviour.
; Because the first `cond` branch evaluated is the 3,
; and `(odd? 3)` is true, none of the other branches
; (containing (/ 1 0) and (/ 2 0)) get evaluated.
```
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
  ((x integer?)
   (y integer?))
  ; The syntax for methods is the same as on the aid sheet.
  ...
)
```

> (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.

(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.

**Solution**

```
(my-class-constraints Person
  ((name string?)
   (age (lambda (x) (and (integer? x) (>= x 0)))))
```
(b) [2 marks] Complete the macro pattern for my-class-constraints. Your pattern should match zero or more attributes; every attribute must be paired with an expression representing a predicate.

Solution

(define-syntax my-class-constraints
  (syntax-rules (method)
    ([my-class-constraints <Class>
      ; (non-function) attributes
      ((<attr> <pred>) ...)
      ; methods -- Don't change this part.
      (method (<name> <params> ...) <body>) ...)

(c) [2 marks] Write the macro template (i.e., what the macro expands into) to implement the required behaviour for my-class-constraints. Important: in the my-class macro found on the aid sheet, refer to the entire (let ([class_dict_ ...]) ...) nested under (define (<Class> <attr> ...) expression as LET-EXPR. You may not modify anything in LET-EXPR in your new template; instead, write "LET-EXPR" in your new template to refer to this part (so that you don’t need to rewrite the entire thing).

; 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.

Solution

(define (<Class> <attr> ...)
  (if (and (<pred> <attr>) ...)
    LET-EXPR
    (error "Contract violation in constructor")))
Use this page for rough work. If you want work on this page to be marked, please indicate this clearly at the location of the original question.