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 9 pages, DOUBLE-SIDED.
- You may generally use helper functions, explicit recursion, higher-order functions, and pattern-matching, 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>7</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>27</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. [7 marks] Programming language concepts.

(a) [2 marks] Consider the following Racket expression (assume all identifiers are defined and all parenthesized expressions are function calls):

\[(a \ (b \ c) \ d \ ((e) \ (f \ g)) \ (h))\]

Using what you know about evaluation order in Racket, list the function call expressions in the order they would be evaluated. Write the entire function call expression, not just the name of the function being called. For example, \((b \ c)\) is a function call expression, and that exact expression should appear somewhere in your list. No explanation is necessary.

**Solution**
\[(b \ c), \ (e), \ (f \ g), \ ((e) \ (f \ g)), \ (h), \ (a \ (b \ c) \ d \ ((e) \ (f \ g)) \ (h))\]

(b) [1 mark] Recall that in Racket, \texttt{and} is not a function. Why not?

**Solution**
\texttt{and} short-circuits: it does not evaluate all of its arguments if it encounters a \texttt{#f} value. This means \texttt{and} expressions can't eager evaluate all of the arguments, and but all function calls in Racket are eagerly evaluated.

(c) [1 mark] Define the term closure.

**Solution**
A closure is a data structure storing a reference to a function body (and its parameters), and an environment that binds the free variables in that function body.
(d) **[3 marks]** Here is an implementation of the Haskell function `filter`.

```haskell
filter _ [] = []
filter pred (x:xs) = 
  if pred x 
  then 
    x : (filter pred xs) 
  else 
    filter pred xs
```

Show how to implement `filter` in a tail-recursive way (hint: have `filter` call a tail-recursive helper function that takes an additional accumulator parameter).

**Solution**

```haskell
filter pred lst = filterTail pred lst []

filterTail _ [] acc = acc
filterTail pred (x:xs) acc = 
  if pred x 
  then 
    filterTail pred xs (acc ++ [x])  -- Note the accumulator order here!
  else 
    filterTail pred xs acc
```
2. [5 marks] **Functional programming.** Consider the Racket function \texttt{argmin}, which takes two arguments:

- A unary function \( f \) that returns an integer.
- A non-empty list \( \texttt{lst} \), where every element is a valid input to \( f \).

\texttt{argmin} returns the element of \( \texttt{lst} \) that minimizes the value of \( f \), using the element that appears latest in the list if there is a tie.

\[
> \text{(argmin string-length (list "hi" "goodbye" "" "a" "hahaha"))}
\]

"" ; The empty string, which has the minimum length.

Implement this function in Racket using a call to \texttt{foldl}. You may assume that the input list is non-empty.

**Solution**

```racket
(define (argmin f lst)
  (foldl (lambda (x acc) (if (<= (f x) (f acc))
                             x
                             acc))
         ; Initial accumulator value is the first list element.
         (first lst)
         ; We only call foldl on the rest of the list.
         (rest lst)))
```


(a) [3 marks] Consider the following macro:

```
(define-syntax my-macro
  (syntax-rules ()
    [(my-macro <f> (<a> <b>) ...) (map <f> (list <a> ... <b> ...)))]))
```

(i) Write an expression using this macro that evaluates to the list ’(10 20 30 40). No explanation necessary.

Solution

```
(my-macro (lambda (x) (+ 10 x)) (0 20) (10 30))
```

(ii) Write an expression using this macro that would result in a runtime error, and briefly explain.

Solution

```
(my-macro first (0 0))
```

This expands into (map first (list 0 0)), which raises a runtime error since first expects a list, not a number.

Note: other possibilities include having a single <f>/<a>/<b> itself raise a runtime error (e.g., by using (/ 1 0)), or have <f> not be a unary function.

(b) [1 mark] Consider the my-class macro on the aid sheet, and the sample class Point defined below it. Suppose we evaluate the definition (define p (Point 3 4)), which binds p to a function. How many arguments does p take?

Solution

1 (the message being sent, i.e., the attribute to lookup)

(c) [2 marks] In the my-class macro, suppose we replace (fix-first me (hash-ref class_dict_msg)) with (hash-ref class_dict_msg), but leave the Point class definition the same. Explain, using English and a code example, what difference we would observe when using Point objects.

Solution

In order to call methods, we would need to pass in a Point object explicitly for self. For example:
(define p (Point 3 4))
((p 'size) p) ; We need to pass in p for self explicitly.

(d) [1 mark] David says, “Each use of my-class binds a global identifier class_dict_. To prevent a duplicate definition error, we can only use my-class once per Racket file.” Is David correct? Explain why or why not.

Solution
David is not correct. Racket macros do not allow identifiers written literally in a template to have scope beyond the macro body (achieved by renaming these identifiers for each macro expansion). This means that multiple uses of my-class will not have conflicting class_dict_ bindings.
4. [8 marks] Abstract Syntax Trees. Here is a Haskell definition of small arithmetic expression language.

```haskell
data Binding = Binding String Expr

data Expr = IntLiteral Integer -- An integer literal
          | Identifier String -- An identifier
          | Plus [Expr] -- The sum of an arbitrary number of expressions
          | Let [Binding] Expr -- A list of name bindings, and the body of the let.
```

```
-- Example expressions
Plus [IntLiteral 10, Identifier "x", Identifier "y"] -- (+ 10 x y)
Let [Binding "x" (IntLiteral 10),
     Binding "y" (Plus [IntLiteral 3, Identifier "x"])] -- (let* ([x 10]
     ] -- [y (+ 3 x)])
Plus [Identifier "x", Identifier "y"] -- (+ x y)
```

The semantics of this expression language are the same as Racket. For example, the last example above evaluates to 23 (since x is 10 and y is 13). The order of bindings matters: switching the bindings of x and y would not be a valid expression. Finally, the only valid values are integers (there are no other “primitive” data types).

In this question, you’ll implement a small interpreter for this language. As you saw on a previous exercise, we’ll use a `Map String Integer` (imported from `Data.Map`) data structure to store name-value bindings. The `Map` functions you may use are found on the aid sheet. Your interpreter will consist of a main function with the following type signature:

```
interpret :: Map String Integer -> Expr -> Integer
```

You may assume that there are no unbound identifiers, so don’t worry about doing any error-checking for this question.

(a) [4 marks] Complete the following three cases of the `interpret` function.

**Solution**

```
interpret env (IntLiteral num) = num
interpret env (Identifier ident) = env ! ident
interpret env (Plus arguments) = foldl (+) 0 $ map (\x -> interpret env x) arguments
```

(b) [4 marks] Implement the final case of the `interpret` function, handling `Let` expressions. A `Let` expression with an empty list of name bindings is valid.
Solution

```haskell
interpret env (Let bindings body) =
  -- Note: we pass in `env` for the initial value so that bindings can be
  -- accumulated across multiple nested Lets (like in Racket).
  let newEnv = foldl updateEnv env bindings
  in
    interpret newEnv body

updateEnv :: Map String Integer -> Binding -> Map String Integer
updateEnv acc (Binding name expr) = insert name (interpret acc expr) acc
```
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.