Name:

Student Number:

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 8 pages, DOUBLE-SIDED.
- You may always write helper functions unless asked not to.
- 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>8</td>
<td></td>
</tr>
<tr>
<td>Q2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Q3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>27</strong></td>
<td></td>
</tr>
</tbody>
</table>
1. [8 marks] Programming language concepts.

(a) [2 marks] Here is a grammar describing a simple arithmetic language. \(<\text{expr}>\) represents a valid expression in the language.

\[
\begin{align*}
\text{<expr>} &= \text{<times-expr>} \\
&\quad | \text{<plus-expr>} \\
&\quad | \text{NUMBER} \\
\text{<times-expr>} &= (\ast \text{<plus-expr>} \text{<plus-expr>}) \\
\text{<plus-expr>} &= (+ \text{NUMBER NUMBER})
\end{align*}
\]

Which of the following are valid expressions in this language? Circle all that apply. No explanation necessary.

\((\ast 20 50)\)

\((\ast 20 50)\)

\((\ast (+ 30 40) (+ 60 3))\)

\(17\)

**Solution**

\((\ast 20 50), (\ast (+ 30 40) (+ 60 3)), \) and 17 are valid.

(b) [2 marks] Circle all the Haskell functions below that are tail-recursive. No explanation is necessary.

\[
\begin{align*}
f x &= \text{if } x == 0 \text{ then } 10 \text{ else } f (x - 1) \\
&\quad \text{if } x == 0 \text{ then } 10 \text{ else } f (x - 1) \\
&\quad \text{if } \text{mod } x 2 == 0 \text{ then } 1 + f (x - 1) \text{ else } f (x - 1) \\
\end{align*}
\]

\[
\begin{align*}
f 0 &= 10 \\
f x &= \text{if } \text{mod } x 2 == 0 \text{ then } f (x - 2) \text{ else } f (x - 1) \\
&\quad \text{if } \text{mod } x 2 == 0 \text{ then } f (x - 2) \text{ else } f (x - 1)
\end{align*}
\]

**Solution**

Top-left and bottom-right are tail-recursive.

(c) [2 marks] Beneath each Racket code snippet, write down the value obtained when you evaluate it. If there would be an error, briefly explain the error.
(define x 10)
(define f1 (lambda (n) (* x n)))

(let* ([x 2]
        [n 3])
  (f1 1))

Solution
10

(define f2 (lambda (n) (* y n)))

(let* ([y 2]
        [n 3])
  (f 1))

Solution
There's an error: y is unbound in the closure of f2.

(d) [2 marks] Suppose we define the following functions in Racket and Haskell:

; Racket version
(define (apply-if f condition x)
  (if condition
      (f x)
      (f 0)))

-- Haskell version
applyIf f condition x =
  if condition
  then
  f x
  else
  f 0

Explain why these functions are not equivalent in Racket and Haskell. Include a concrete code example in your explanation.

Solution
Racket evaluates function call arguments eagerly (before calling the function), while Haskell doesn’t. If condition is false, x doesn’t need to be evaluated in Haskell, but it still does in Racket. So (apply-if identity #f (error "e")) would raise an error in Racket, but applyIf id False (error "e") would evaluate to 0 in Haskell.
2. [6 marks] Functional programming. Consider the Racket function \texttt{indexes-where}, which takes exactly two arguments:

- A unary predicate (returns a boolean) \texttt{pred}.
- A list \texttt{lst}, where every element is a valid input to \texttt{pred}.

\texttt{indexes-where} returns a list containing the indexes of the elements of \texttt{lst} that satisfy \texttt{pred}.

\begin{verbatim}
> (indexes-where (lambda (s) (> (string-length s) 3)) (list "hi" "goodbye" "" "a" "hahaha")) '(1 4)
\end{verbatim}

Implement this function in Racket. The only list functions you may use are the ones found on the aid sheet. You may use any combination of pattern-matching, explicit recursion, and higher-order list functions.

\textbf{Hint}: define a helper function with an extra parameter that keeps track of the “current” index.

\textbf{Solution}

; Version 1, using explicit recursion
(define (indexes-where pred lst)
  (indexes-where-helper pred lst 0))

(define (indexes-where-helper pred lst i)
  (if (null? lst)
      null
      (let* ([x (first lst)]
             [rest-indexes (indexes-where-helper pred (rest lst) (+ i 1))])
        (if (pred x)
            (cons i rest-indexes)
            rest-indexes))))

; Version 2, using \texttt{foldl}
; It's a bit harder to use \texttt{foldl}, because we want to accumulate both
; the list of indexes \texttt{*and*} the current index, \texttt{*and*} we want
; the "update" function to have \texttt{pred} in scope.
(define (indexes-where1 pred lst)
  (first (foldl (indexes-where-helper1 pred) (list null 0) lst)))

(define (indexes-where-helper1 pred)
  ; A higher-order function that takes a predicate and returns the
  ; actual update function.
  (lambda (x acc)
    (let* ([indexes (first acc)]
           [i (second acc)])
      (if (pred x)
          (list (append indexes (list i)) (+ i 1))
          (list indexes (+ i 1))))))

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

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

Write an expression using this macro that produces the list ’(10 20) when evaluated.

Solution

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

Write an expression using this macro that would result in a runtime error when evaluated. Briefly explain what error would occur.

Solution

```
(my-macro first 1)
```

This expands into (list (first 1)), which raises a runtime error since first expects a list, not a number.

(b) [2 marks] 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)).

(i) In Racket, p is technically a function. How many arguments does p take?

Solution

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

(ii) Write an expression to access the y attribute of p.

Solution

```
(p 'y)
```

(iii) What is the difference between the expressions (p 'size) and ((p 'size))?

Solution

```
(p 'size) refers to a function (the size “method”), while ((p 'size)) calls the function, and returns the size of the point.
```

(c) [1 mark] Suppose we switch the order of attribute lookup in the my-class macro, so that class_dict is searched before self_dict.

Explain how this would change the behaviour of the my-class macro.

Solution

This means that when we define a class that has an attribute and method with the same name, the method is the one returned when we send a message with that name to instances of the class.

We could say that the method name “shadows” the attribute name.
4. [7 marks] Abstract Syntax Trees. Here is a Haskell definition of small arithmetic expression language.

```haskell
data Expr = IntLiteral Int    -- An integer literal
  | Identifier String -- An identifier
  | Plus [Expr]      -- The sum of an arbitrary number of expressions

-- Example expressions
Plus [IntLiteral 3, IntLiteral 5]  -- (+ 3 5)
Plus [IntLiteral 10, Identifier "x", Identifier "y"]  -- (+ 10 x y)
Plus [Plus [IntLiteral 4], Plus [IntLiteral 10, Identifier "a"] ]  -- (+ (+ 4) (+ 10 a))
```

For all parts of this question, the only list functions you may use are the ones found on the aid sheet, and `concat`, which takes a list of lists and flattens them into a single list.

(a) [3 marks] Implement the following Haskell function. We have started the pattern-matching for you.

**Solution**

```haskell
extractPrimitives :: Expr -> [Expr]
extractPrimitives (IntLiteral n) = [IntLiteral n]
extractPrimitives (Identifier s) = [Identifier s]
extractPrimitives (Plus args) = concat (map extractPrimitives args)
```

(b) [2 marks] Because of the properties of addition, it is possible to (statically) flatten nested `Plus` expressions without changing their value. For example,

```
Plus [Plus [IntLiteral 4], Plus [IntLiteral 10, Identifier "a"]]
```

can be simplified to

```
Plus [IntLiteral 4, IntLiteral 10, Identifier "a"]
```

Using `extractPrimitives` or otherwise, implement the function on the next page, which performs this simplification.

**Solution**

```haskell
flattenPlus :: Expr -> Expr
flattenPlus (Plus args) = Plus (extractPrimitives (Plus args))
-- Or, using concat again
flattenPlus (Plus args) = Plus (concat (map extractPrimitives args))
```

(c) [2 marks] **Warning:** this is a more challenging part, and not worth much. Please only attempt this after completing the rest of this test.

A further simplification we can make to a `Plus` expression is to compute the sum of all `IntLiteral` in the expression, so that no more than one `IntLiteral` appears in a single `Plus` expression.

Implement the following function, which performs this simplification.

**Solution**

```haskell
combineLiterals :: Expr -> Expr
combineLiterals (Plus args) =
```

let ints = extractInts args
  ids = filter isIdentifier args
in
  if ints == []
    then Plus ids
    else
      Plus (IntLiteral (foldl (+) 0 ints) : ids)

extractInts :: [Expr] -> [Int]
extractInts [] = []
extractInts (IntLiteral n: exprs) = n : extractInts exprs
extractInts (expr: exprs) = extractInts exprs

isIdentifier :: Expr -> Bool
isIdentifier (Identifier _) = True
isIdentifier _ = False
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.