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, pattern-matching, and do notation, 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>8</td>
<td>8</td>
</tr>
<tr>
<td>Q2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Q3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Q4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>27</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.

\[
\begin{align*}
\text{<expr>} & = \text{<times-expr>} \\
& \mid \text{<plus-expr>} \\
& \mid \text{NUMBER}
\end{align*}
\]

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

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

\(\star 20 50\)

\(+ 20 50\)

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

17

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

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

\[
\begin{align*}
f 0 &= 10 \\
f 1 &= 20 \\
f x &= (f \ (x - 1)) + (f \ (x - 2))
\end{align*}
\]

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

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

; Snippet 1
(define x 10)
(define f1 (lambda (n) (* x n)))
(let* ([x 2]
        [n 3])
  (f1 1))

; Snippet 2
(define f2 (lambda (n) (* y n)))
(let* ([y 2]
        [n 3])
  (f 1))

(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.
2. [6 marks] **Functional programming.** Consider the Racket function `indexes-where`, which takes exactly two arguments:

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

`indexes-where` returns a list containing the indexes of the elements of `lst` that satisfy `pred`.

```racket
> (indexes-where (lambda (s) (> (string-length s) 3)) (list "hi" "goodbye" "" "a" "hahaha"))
'(1 4)
```

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.

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

```racket
(define (indexes-where pred lst)
  ; your implementation here
)

(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.

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

(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?

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

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

(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.
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 \texttt{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.

```haskell
-- Return a list of the IntLiterals and Identifiers that appear in the given expression.
-- > extractPrimitives (IntLiteral 3)
--   [IntLiteral 3]
-- > extractPrimitives (Plus [Plus [IntLiteral 4], Plus [IntLiteral 10, Identifier "a"]]])
--   [Intliteral 4, IntLiteral 10, Identifier "a"]

extractPrimitives :: Expr -> [Expr]

extractPrimitives (IntLiteral n) =

extractPrimitives (Identifier s) =

extractPrimitives (Plus args) =
```

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

\[
\text{Plus [Plus [IntLiteral 4], Plus [IntLiteral 10, Identifier "a"]]]}
\]

can be simplified to

\[
\text{Plus [IntLiteral 4, IntLiteral 10, Identifier "a"]}
\]

Using \texttt{extractPrimitives} or otherwise, implement the function on the next page, which performs this simplification.
Midterm, CSC324H1

```
-- Simplify a Plus expression by flattening nested Pluses into a single Plus.
-- You may assume the input is a Plus expression.
-- > flattenPlus (Plus [Plus [IntLiteral 4], Plus [IntLiteral 10, Identifier "a"]])
-- Plus [IntLiteral 4, IntLiteral 10, Identifier "a"]

flattenPlus :: Expr -> Expr
flattenPlus (Plus 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 IntLiterals in the expression, so that no more than one IntLiteral appears in a single Plus expression.
Implement the following function, which performs this simplification.

-- Simplify a Plus expression by combining all integer literals into a single literal.
-- The subexpressions of the returned Plus expression can be in any order.
-- You may assume the input is a *flat* Plus expression (contains no nested Plus).
--
-- > combineLiterals (Plus [IntLiteral 4, IntLiteral 10, Identifier "a"])
-- Plus [IntLiteral 14, Identifier "a"]

combineLiterals :: Expr -> Expr
combineLiterals (Plus args) =
```
Midterm, CSC324H1

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.