UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 2 - SOLUTIONS
CSC324H1S

March 18, 2015 (**50 min.**)
Examination Aids: Aid sheet on back, detachable!

Name:

Student Number:

Please read the following guidelines carefully!

- Please write your name on the front **and back** of the exam.
- This examination has **4** questions. There are a total of **6 pages, DOUBLE-SIDED**.
- You may always write helper functions unless explicitly asked not to. You may not use mutation or any Racket iterative constructs.
- Any question you leave blank or clearly cross out your work and write "I don't know" is worth **10% of the marks**.

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.

Good luck!

1. These questions all deal with the *Racket* programming language.

   (a) **[2]** Consider the following snippet of Racket code:

   ```
   (define a 10)
   (define (f b) (+ a b))

   (let ([a 100]
         [b 13])
     (f a))
   ```

   State the output of this code in two cases: standard Racket using lexical scope, and if Racket were changed to used dynamic scope.

   Output under lexical scope: ___110___ **Output under dynamic scope**: ___200___

   (b) **[1]** Name one *similarity* between a Racket macro and a Racket function.

   Solution: See midterm 1.

   (c) **[2]** Define, **in English**, the term **identifier (or variable) shadowing**. Then, give a **Racket code example** of identifier shadowing.

   Solution: Identifier shadowing is when there is a local name declaration (either as a function parameter or in a `let` binding) where the name has already been declared in some outer scope.

   This is significant because any references to that name will now refer to the *innermost* name binding, ignoring any outer bindings.

   Part (a) actually show an instance of this: the let binding of `a` (to value `100`) shadows the global binding of `a` to `10`.

   Incidentally, just like the first midterm, I wanted you to see that variable shadowing "takes precedence" over dynamic vs. lexical scope! If a parameter of a function shadows an outer binding (either lexical or dynamic), any references to that parameter name inside the function body will refer to the parameter! This means that the `b` in the function body *always* refers to the parameter `b`. Even in a dynamically scoped language like bash, this occurs.

   (d) **[1]** Consider the following Racket function:

   ```
   (define (foldr f i lst)
     (if (empty? lst)
         i
         (f (first lst) (foldr f i (rest lst)))))
   ```

   What function is in **tail call position**?

   Solution: `f`.

2. (a) **[2]** Explain the difference between lazy evaluation (used in Haskell) and eager evaluation (used in Racket).

Solution: Under eager evaluation, every argument expression to a function must be fully evaluated before they are passed to a function. Under lazy evaluation, argument expressions are not evaluated, but instead substituted directly into the function body. Expressions are only evaluated when "aboslutely necessary", e.g., when they must be printed out in the interpreter, or for pattern matching.

[The Haskell implementation is a bit more complex than this, but this sort of answer is fine for our purposes.]

(b) **[3]** Define the function `iterate :: (a -> a) -> a -> [a]`, which takes a function `f` and argument `x`, and returns the *infinite* list `[x, f(x), f(f(x)), f(f(f(x))), ...]`. I.e., the n-th item in the list is equal to `f` applied to `x` n times.

Solution:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : map f (iterate f x)

-- or
iterate f x = x : iterate f (f x)
```

(c) **[2]** Show how to use `iterate` to define `nats`, a list of all natural numbers. Do not just repeat the definition given on the aid sheet!

Solution:

```
nats = iterate (+1) 0

-- or, if you don't like sectioning
add1 x = x + 1
nats = iterate add1 0
```

3. (a) **[2]** Though both Haskell and Java are *statically-typed* languages, they differ in a few significant ways in how they handle types. Define **type inference**, which is an action that Haskell performs and Java (mostly) does not.

Solution: type inference is the ability of a compiler to determine the types of expressions *without* explicit type declarations from the programmer (like in Java), nor evaluating the expression (like in dynamically-typed languages like Racket).

(b) **[7]** For each of the following Haskell definitions, give the type that Haskell would infer for it in the space provided. We are looking for proper understanding of basic types, currying, type variables, and typeclass constraints here.

```
g :: Bool -> [[Char]] -> [Char]
```

```
g x y = if x then head y else "Hello"
```

```
elem :: Eq a => a -> [a] -> Bool

elem _ [] = False
elem x (y:ys) = x == y || elem x ys
```

```
h :: (a -> b) -> a -> b

h x y = x y
```

```
k :: [[a]] -> [[a]]

k = filter (\x -> (length x) == 0)
```

(c) **[2]** Consider the following type definition:

```
data MaybeInt = Failure | Success Integer
```

Identify the **type(s)** that is/are defined, and how we could create values of those type(s) in Haskell.

Solution: The new type created is **MaybeInt**. We can express values of this type in two ways: using `Failure` (which is already a value of type `MaybeInt`), or `Success` (which takes in an `Integer` and returns a value of type `MaybeInt`).

4. For this question, you are given the following Haskell definitions from lecture. (We've replaced `>~>` with `>->` for typesetting purposes.)

```
type Stack = [Integer]
type StackOp a = Stack -> (a, Stack)

pop :: StackOp Integer
pop (x:xs) = (x, xs)

push :: Integer -> StackOp ()
push a stack = ((), a:stack)

(>>>) :: StackOp a -> StackOp b -> StackOp b
f >>> g = \s ->
  let (_, s1) = f s
  in  g s1

(>->) :: StackOp a -> (a -> StackOp b) -> StackOp b
f >-> g = \s ->
  let (x, s1) = f s
  in  g x s1

returnNoMutate :: a -> StackOp a
returnNoMutate x = \s -> (x, s)
```

   (a) [**2**] Recall from lecture that our `sumOfStack` function correctly computed the sum of a stack, but destroyed the stack in the process. Here is one attempt at fixing this problem:

```
sumOfStack :: StackOp Integer
sumOfStack [] = (0, [])
sumOfStack s = (
  pop >-> \x ->
  sumOfStack >-> \y ->
  returnNoMutate (x + y) >>>
  push x) s
```

   Explain the problem with this definition. (Note: a long explanation is not required. We wanted to give you more space for the next part on the following page.)

   Solution: the intuitive problem is that by ending with a `push`, the actual sum (`x + y`) is lost, because `push x` always returns void as its "data." But the problem is even worse than the function returning the wrong thing: as it stands now, this code won't even type check! Because we've ended with a `push x :: StackOp ()`, the entire parenthesized expression will have type `StackOp ()`, not `StackOp Integer`. (The easiest way to see this is that the output `StackOp` for both `>>>` and `>->` uses the type variable of the *second* argument, not the first.)

(b) [**4**] Implement the function `foldStack`, which behaves like a foldl/foldr on stacks. While you may use pattern matching for the empty stack case, **do not use any list functions in your answer**. We are looking for appropriate use of the stack functions found on the previous page in this question. Your function may destroy the stack – that is, you may return an empty stack as the "mutated" stack for this function.

Hint 1: Pay attention to the given type signature. This should tell you exactly how the arguments must be combined in the body of the function. It's your job to figure out how this works.

Hint 2: You should be able to implement the destructive `sumOfStack` function from lecture as `sumOfStack = foldStack (+) 0`.

Solution:

```
foldStack :: (a -> Integer -> a) -> a -> StackOp a
foldStack _ init [] = (init, [])
foldStack combine init s = (
    pop >-> \x ->
    foldStack combine init >-> \y ->
    returnNoMutate (combine y x)            -- Note the order of args!
    ) s

-- Or, in the reverse order
foldStack _ init [] = (init, [])
foldStack combine init s = (
    pop >-> \x ->
    foldStack combine (combine init x)
    ) s
```

One of these approaches is like `foldl`, and the other is like `foldr`. Which one is which?

Note: the first definition is the one I expected most of you to try; notice how similar it is to the `sumOfStack` from lecture (or part (a)). One of the fundamental concepts I want you to take away from this course is how to generalize a given function to a higher level of abstraction and reusability.