UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 1 – SOLUTIONS
CSC324H1S

February 6, 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. **[5]** Write a Racket function `(not-in-range f domain codomain)`, which returns the elements `x` of `codomain` satisfying the condition that *no* element in `domain` makes `f` evaluate to `x`. The items must be returned in the same order they appear in `codomain`. For example,

```
> (not-in-range (lambda (x) (+ x 3)) '(1 2 6) '(1 2 3 4 5))}
'(1 2 3)
```

You may use explicit recursion, higher-order functions, or a combination of the two. You are encouraged to define your own helper functions. **You may only use the list functions found on the aid sheet.**

Solution:

```
(define (not-in-range f domain codomain)
  (filter (lambda (x)
            (not (member x (map f domain))))
          codomain)

; Or, a very recursive approach
(define (not-in-range f domain codomain)
  (cond [(empty? codomain) '()]
        [(in-range f domain (first codomain))
         (cons (first codomain) (not-in-range f domain (rest codomain)))]
        [else (not-in-range f domain (rest codomain))]))

; Return true if an element in domain makes f evaluate to x
(define (in-range f domain x)
  (cond [(empty? domain) #f]
        [(equal? (f (first domain)) x) #t]
        [else (in-range f (rest domain) x)]))
```

Common errors:

- Misreading the question: returning elements of the domain instead of codomain, returning items in the domain which *were* in the range of `f`
- Returning items in `codomain` in reverse
- Type errors: calling functions on the wrong numbers/types of arguments, returning the wrong type.

2. (a) **[2]** Consider the following Racket function call expression, which contains other function call subexpressions:

```
(a b (c b) (d (e f g) h (i)) (j k))
```

Using what you know about evaluation order in Racket, state the order in which the *function calls* in these expressions are evaluated. You may list either the functions which are called (e.g., "`c`"), or the entire expression (e.g., "`(c b)`") in your answer.

Solution: c, e, i, d, j, a. (Common errors: listing values which were not functions being *called*; going right-to-left instead of left-to-right.)

(b) **[2]** Write a Racket macro `(unless <condition> <expr>)` which evaluates and returns the value of `<expr>` if and only if `<condition>` is `#f`, and otherwise evaluates to `(void)`.

Solution:

```
(define-syntax unless
  (syntax-rules ()
    [(unless <condition> <expr>)
     (if <condition>
         (void)
         <expr>)]))
```

(c) **[2]** Describe **one similarity** and **one difference** between a function and a macro in Racket.

Solution:

- Macros and functions both operate by substitution of expressions/values into other expressions.
- For the macros we've seen in lecture, their name must appear as the first identifier inside an expression, just like a function. That is, we signify their use in an expression in the same way. (This isn't true of all macros, but an acceptable answer given what you've seen in this course.)
- Macros do not evaluate expressions before substitution; functions do.
- Macros are expanded before the program is run; functions are only called during runtime.
- Macros can contain literals in their expressions; functions can't.
- Macros can expand into multiple definitions/expressions; functions just return a single value.
- Macros can expand into a definition that can be used in the rest of the program; `define` expressions in a function body are local to that body.

3. Recall the definition of `map` given in lecture:

```
(define (map f lst)
  (if (empty? lst)
      '()
      (cons (f x) (map f (rest lst)))))
```

(a) [**1**] Which function call in this definition of `map` is in tail call position? **Solution**: `cons`

(b) [**1**] Define **tail recursion**. **Solution**: A recursive function where the recursive call is in tail position.

(Common errors: describing tail call *optimization*)

(c) [**3**] Implement a tail-recursive version of `map`. Your function may have an extra "accumulator" parameter. You may *not* use any higher-order functions (like `foldl`). **Solution**:

```
(define (tail-map f lst acc)
  (if (empty? lst)
      acc
      (tail-map f
                (rest lst)
                (append acc
                        (list (f (first lst)))))))
```

Note: many students used an optional argument or a tail-recursive helper function to get the exact signature of map; this was perfectly acceptable. Generally well-done, with the most significant error being a misunderstanding of what tail-recursion actually is.

(d) [**1**] Give an example of how your function would be called, along with the expected output.
Solution: `(tail-map sqr '(1 2 3) '()`, which outputs `'(1 4 9)`.

4. (a) **[2]** Define a **free identifier** of a function. Give an example of a function definition with at least one free identifier and one bound identifier. State which identifier(s) are free and which one(s) are bound.

Solution: free: an identifier which is not local to a function (not a parameter, or defined in local scope inside the function body).

Example: `(lambda (x) (+ x y)`. `x` is bound, `y` is free.

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

```
(define x 10)
(define (make-sum y) (lambda (z) (+ x y z)))

(let ([x 100] [y 0] [z -30])
  ((make-sum 2) 3))
```

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: 15 ________ **Output under dynamic scope**: 105________

Note: some students didn't realize that our discussion of lexical vs. dynamic scope and closures only applied to free identifiers in a function. Function parameters always shadow identifiers defined outside of a function body!

(c) [**4**] Implement the following Racket function. Once again, **you may not use any list functions not found on the aid sheet**.

```
#|
(list-ref-many indexes)
  indexes: a list of non-negative integers

  Returns a function g which takes a list and returns the items in the list at the positions
  in 'indexes'. Any indexes which are out-of-bounds for an input list are ignored.

> ((list-ref-many '(1 2 4 0 1)) '("a" "b" "c" "d" "e"))
'("b" "c" "e" "a" "b")
> ((list-ref-many '(1 2 4 0 1)) '("a" "b" "c"))
'("b" "c" "a" "b")   ; the 4 is ignored
|#
```

Solution:

```
(define (list-ref-many indexes)
  (lambda (lst)
    (map (lambda (i) (list-ref lst i))
         (filter (lambda (i) (< i (length lst))) indexes))))
```

Or, doing the `map` first:

```
(define (list-ref-many indexes)
  (lambda (lst)
    (apply append
           (map (lambda (i)
                  (if (< i (length lst))
                      (list (list-ref lst i))
                      '()))
                lst))))
```

Or, a purely recursive approach:

```
(define (list-ref-many indexes)
  (lambda (lst)
    (cond [(empty? indexes) '()]
          [(>= (first indexes) (length lst))
           ((list-ref-many (rest indexes)) lst)]
          [else
           (cons (list-ref lst (first indexes))
                 ((list-ref-many (rest indexes)) lst))])))
```