Student Name: ________________________________

CS 2500/Accelerated Exam 2—Fall 2018

Amal Ahmed

November 27, 2018

- We will not answer questions during the exam. If you believe a problem statement is ambiguous, choose *any* non-trivial interpretation.
- Write down the answers in the space provided, including the back of the given spaces and pages marked ”intentionally left blank”.
- You may use the paper copy of the book, your notes, and design-recipe cards.
- You may *not* use any electronic gadgets (for example, watches, google glasses, phones, tablets, laptops). Any use of an electronic gadget will lead to immediate expulsion from the exam and class.
- You may use all the definitions, expressions, and functions found in ISL+. Define everything else.
- The phrase "design a program" means that you should apply the data and function design recipes. You may write ”`c-e`” as shorthand for `check-expect`.
- Unless a problem requests a solution that does not use ISL abstractions, you may use these abstractions. Similarly, unless a problem demands a solution that uses the abstractions of ISL, you do not have to use these abstractions.
- You may reuse any functions you define when solving other problems. You may also use any of the examples we provide for your own tests.
- Basic test-taking advice: Before you start answering any problems, read *every* problem, so your brain can be thinking about the harder problems in the background while you knock off the easy ones.

Problem		Max. Points
1		8
2		6
3		12
4		7
5		10
6		7
Total		/ 50
Extra Credit		4

1 Free and Bound Variables

Consider the following data definition for expressions.

```
; An Expr is one of:
; - String (representing a variable)
(define-struct lam [var body])
; - (make-lam String Expr)
(define-struct app [fun arg])
; - (make-app Expr Expr)
; interpretation An Expr represents an expression, and
; is either a String (which represents a variable),
; or a (make-lam v b) which represents a lambda
; expression where v is a variable bound by the
; lambda and b is the body of the lambda,
; or a (make-app f a) representing an application
; of an expression f to an argument a.
```

Problem 1 (8 points)

Design a function `free-bound` that consumes an `Expr` and replaces all occurrences of free variables in the expression with `"free"` and others with `"bound"`, leaving any variable `v` that appears in a `(make-lam v b)` unchanged.

Definition A variable `v` is *free* in an expression if it is not bound by an enclosing lambda expression.

The following tests should pass. (You don't have to define your own tests.)

```
(check-expect (free-bound "x") "free")
(check-expect (free-bound (make-lam "y" "z"))
              (make-lam "y" "free"))
(check-expect
  (free-bound (make-lam "y"
                (make-app (make-lam "z" "y")
                          "z")))
  (make-lam "y"
     (make-app (make-lam "z" "bound")
               "free")))
```

intentionally left blank

2 Recurrence Relations

Functions that operate over natural numbers and are recursively defined are often referred to as *recurrence relations*.

Here are some examples (expressed in mathematical notation):

```
Triangular numbers (0,1,3,6,10...):
tri(0) = 0
tri(n) = n + tri(n - 1)

Factorial n! (1,1,2,6,24....):
fact(0) = 1
fact(n) = n * fact(n - 1)

Fibonnaci (1,1,2,3,5...):
fib(0) = 1
fib(1) = 1
fib(n) = fib(n - 1) + fib(n - 2)

Numbers ("0","01","012","0123","01234"...):
numbers(0) = "0"
numbers(n) = string-append(numbers(n - 1),number->string(n))
```

Consider the following data definition `[RR X]` for recurrence relations:

```
(define-struct rr [bases recurrence])
; An [RR X] (where RR = Recurrence Relation) is a
;   (make-rr [List-of [Base X]] [Recursive X])
; interpretation A (make-rr base-cases recursive-computation)
; represents a recurrence relation where all elements of
; base-cases have unique "input"s and recursive-computation
; describes a single recursive case.

(define-struct base [input output])
; A [Base X] is a (make-base Nat X)
; interpretation A (make-base input result) tells us the result
; for a base-case input.

(define-struct recursive [gen-args combine-results])
; A [Recursive X] is a
;   (make-recursive [Nat -> [List-of Nat]]
;                   [Nat [List-of X] -> X])
; interpretation A (make-recursive gen comb) describes the
; recursive case of a recurrence relation
; where gen, given a natural-number input n, generates a
; list of k inputs for k recursive calls,
; and comb, given the (same) input n, and the results
; of the k recursive calls, combines those results to
; produce the output.
```

Example Triangular numbers can be represented as an `[RR Nat]` as follows:

```
(define tri
  (make-rr (list (make-base 0 0))
           (make-recursive (λ (n) (list (sub1 n)))
                           (λ (n subresults)
                              (apply + (cons n subresults))))))
```

Problem 2 (6 points)

Convert the other three examples (i.e., `fact`, `fib`, and `numbers`) into examples of recurrence relations `[RR X]`.

Problem 3 (12 points)

Design a function `recurrence->func` that converts an `[RR X]` to its corresponding `[Nat -> X]` function.

It should pass the following test:

```
  (check-expect (build-list 5 (recurrence->func tri))
                '(0 1 3 6 10))
```

Please also provide similar tests using the other three recurrence relations (`fact`, `fib`, and `numbers`) that you defined for the previous problem.

3 Full Trees and Quick Lists

Consider the following data definition for full trees:

```
(define-struct node (size value left right))

; A [FullTree X] is one of:
; - 'leaf
; - (make-node Nat X [FullTree X] [FullTree X])
; interpretation A [FullTree X] is either an empty tree ('leaf)
; or a node (make-node sz val l r) where sz represents the
; number of nodes (including this node itself and the nodes
; in the subtrees) that are in the tree, val is some data,
; l is the left subtree, and r is the right subtree.
; A full tree is guaranteed to be perfectly balanced, which
; means the height of its two subtrees are equal and both
; of its subtrees are perfectly balanced. In other words,
; non-empty full trees can only contain 1, 3, 7, 15, 31, etc.
; values.
```

Some examples of full trees:

```
(define LEAF 'leaf)
(define ft1 (make-node 1 'good LEAF LEAF))
(define ft2 (make-node 3 'hello
                       ft1
                       (make-node 1 'bye LEAF LEAF)))
(define ft3 (make-node 3 'so
                       (make-node 1 'it LEAF LEAF)
                       (make-node 1 'is LEAF LEAF)))
(define ft4 (make-node 7 'foo ft2 ft3))
```

Problem 4 (7 points)

Design the function `tree-combine`, which generates a new full tree. It takes a value to place at the root of the newly generated tree and takes two full trees of the same size to be the new tree's left and right subtrees.

Problem 5 (10 points)

The values in a full tree are ordered such that the 0-th value in the tree is at the root, and values in the left subtree come before values in the right subtree.

Design the function `tree-ref`, which given a full tree and a natural number `n` gets the `n`-th value in the tree. Assume the size of the tree is greater than `n`.

The following tests should pass. (You do not have to write your own tests.)

```
(check-expect (tree-ref ft1 0) 'good)
(check-expect (tree-ref ft2 0) 'hello)
(check-expect (tree-ref ft2 1) 'good)
(check-expect (tree-ref ft2 2) 'bye)
(check-expect (build-list 7 (λ (i) (tree-ref ft4 i)))
              '(foo hello good bye so it is))
```

Tip It will be easier to design the function if you first draw the full trees used in the above tests and label their nodes.

intentionally left blank

Problem 6 (7 points)

If we frequently need to access the `n`-th value in a list that contains a large number of values, then instead of representing this "flat list of values" as a `[List-of X]`, we would do better if we represent it as a "list of trees of values".

We will now use a `[List-of [FullTree X]]` to represent what is conceptually a list of X values, with the goal of quickly accessing the `n`-th value even when it's deep inside the list.

```
; A [QuickList X] is a [List-of [FullTree X]]
; and represents a list of X values, where if the
; first tree of the quick list is a full tree of
; size n, then the first n elements of the
; represented list are contained in the first tree.
; All of the trees in a quick list are guaranteed
; to be non-empty.
```

Design `quicklist-ref`, which given a `[QuickList X]` and a natural number `n` produces the `n`-th element of the represented list. Assume the total number of elements in the represented list (note: *not* the number of trees in the quick list, but the total number of values inside all of those trees) is greater than `n`.

The following test should pass. (You do not have to write your own tests.)

```
(check-expect
   (build-list 4
      (λ (i) (quicklist-ref (list ft1 ft3) i)))
   '(good so it is))
```

intentionally left blank

Problem 7 EXTRA CREDIT (4 points)

So, we know how to use our quick list to access an element deep inside the list, but how do we actually build it?

Design `quick-cons`, which takes a value and a quick list and adds the value to the quick list. It works as follows: If there at least two trees in the quick list and the first two are the same size, then make a new tree by combining those two and adding the given value at the root. Otherwise, add a new tree to the front of the list that contains just the given element

Given this strategy, we will refine our quick list data definition. This refinement will ensure that as we build up our quick list with quick-cons, we will be creating sizable trees so we can take advantage of the speed of tree-ref, as we would gain no advantage if all of the trees in the quick list were small.

```
; A [QuickList X] is a [List-of [FullTree X]]
; and represents a list of X values, where if the
; first tree of the quick list is a full tree of
; size n, then the first n elements of the
; represented list are contained in the first tree.
; All of the trees in a quick list are guaranteed
; to be non-empty.
; All of the trees are guaranteed to be successively
; larger, with the possible exception of the first
; two trees being the same size.
```

Be sure to test quick-cons on quick lists that satisfy all the constraints specified in the updated data definition of `[QuickList X]`.

intentionally left blank