FINAL EXAM REVIEW PROBLEM SOLUTIONS

This document contains solutions to the problems on the final exam review problems posted earlier **except for** Problem 1, Problem 3, and Problem 15b.

Problem 2: Explicit Typing

The explicitly typed versions of the given programs are shown below. The type annotations are highlighted in boldface.

```
    (program (n)
     (bindrec ((even? (-> (int) bool)
                 (abs ((n int))
                   (if (= n 0)
                       #t
                       (odd? (- n 1)))))
               (odd? (-> (int) bool)
                 (abs ((n int))
                   (if (= n 0)
                       #f
                       (even? (- n 1))))))
       (prepend (even? 5)
         (prepend (odd? 5)
           (empty bool)))))

(program (hi)
  (bindrec
    ((map (forall (a b)
              (-> ((-> (a) b) (listof a))
                   (listof b)))
      (pabs (a b)
        (abs ((f (-> (a) b)) (lst (listof a)))
          (if (empty? lst)
              (empty b)
              (prepend (f (head lst))
                       ((papp map a b) f (tail lst)))))))
     (from-to (-> (int) (listof int))
      (abs ((lo int))
        (if (> lo hi)
            (empty int)
            (prepend lo (from-to (+ lo 1))))))
     )
    (bind test-list (from-to 1)
      (prepend ((papp map int (listof int))
                (abs ((n int)) (prepend n (empty int)))
                ((papp map int int)
                 (abs ((x int)) (* x x))
                 test-list))
        (prepend ((papp map bool (listof int))
                  (abs ((b bool))
                    (if b
                        (prepend 1 (empty int))
                        (prepend 0 (empty int))))
                  ((papp map int bool)
                   (abs ((y int)) (= (mod y 2) 0))
                   test-list))
          (prepend ((papp map int (listof int))
                    (abs ((z int))
                      (prepend z
                        (prepend (* 2 z)
                          (empty int))))
                    test-list)
                   (empty (listof (listof int))))))))
```

Problem 4: Non-local Exits

```
(define (fringe tree)
  (label return
    (letrec ((helper (lambda (tr address)
                       (if (leaf? tr)
                           (if (number? tr)
                               (jump return (cons tr address))
                               (list tr))
                           (append (helper (left tr)
                                           (postpend address ‘left))
                                   (helper (right tr)
                                           (postpend address ‘right)))))))
      (helper tree ‘()))))
```

Part a.

Expression	Value
`(fringe (node (node ‘a ‘b) (node ‘c ‘d)))`	`(a b c d)`
`(fringe (node (node ‘a 2) (node ‘c ‘d)))`	`(2 left right)`
`(fringe (node (node ‘a ‘b) (node 3 ‘d)))`	`(3 right left)`
`(fringe (node (node ‘a 2) (node 3 ‘d)))`	`(2 left right)`

Part b. When given a tree whose leaves are non-numbers, `fringe` returns a list of the leaves of the tree in left-to-right order. When given a tree containing one or more numbers, `fringe` returns an answer of the form `(`*`n . address`*`)` , where n is the leftmost number in the tree, and *`address`* is the address of that number. Here, an address is expressed as a sequence of the symbols `left` and `right` specifying the “directions” to the number from the root of the tree.

Part c. It is possible to implement `fringe` without `label` and `jump`, but not as elegantly as with them. Without non-local exits, it would be necessary to treat differently the two different kinds of return values: the list of leaves returned in the regular case and the list of elements returned in the exceptional case. The code to handle these cases differently would make the definition of fringe harder to read and modify.

Problem 5: Parameter Passing

Consider the following expression:

```
(let ((n 0))
    (let ((add-twice (lambda (x)
                       (begin (set! x (* 2 x))
                              (set! n (+ n x))
                              n))))
      (let ((test (lambda (z)
                    (+ (* 100 (add-twice n))
                       (+ (* 10 z) z)))))
        (test (add-twice 1)))))
```

For each of the following parameter-pasing mechanisms, indicate the value of the above expression in a version of lexically-scoped Scheme using that parameter-passing mechanism:

Parameter-Passing Mechanism	**Value of sample expression**
Call-by-value	622
Call-by-reference	822
Call-by-name	24
Call-by-need	22

Problem 6: Parameter Passing

Consider the following expression:

```
(let ((a 1))
  (let ((inc (lambda (x)
               (begin (set! a (+ a x))
                      a)))
        (f (lambda (y z)
             (begin
               (set! y (+ y 3))
               (+ a (* z z))))))
    (f a (inc 1))))
```

For each of the following parameter-pasing mechanisms, indicate the value of the above expression in a version of Scheme using that parameter-passing mechanism:

Parameter-Passing Mechanism	Value of sample expression
Call-by-value	6
Call-by-reference	9
Call-by-name	7
Call-by-need	5

Problem 7: Desugaring

Part a. For each of the following parameter passing mechanisms in an *imperative* version of statically-scoped Scheme, explain your answer to the following question:

Are `(or1` E_1 E_2`)` and `(or2` E_1 E_2`)` interchangeable for all expressions E_1 and E_2?

- **call-by-value**: No. Supppose E_1 evaluates to true and E_2 has a side effect (e.g., increments a global variable). Then `(or1` E_1 E_2`)` will perform E2's side `effect,` but `(or2` E_1 E_2`)` will not.
- **call-by-name**: Yes. E_1 and E_2 perform any side effects the same number of times and in the same order in both `(or1` E_1 E_2`)` and `(or2` E_1 E_2`)`.
- **call-by-need**: Yes. E_1 and E_2 perform any side effects the same number of times and in the same order in both `(or1` E_1 E_2`)` and `(or2` E_1 E_2`)`.

Part b. Without the assumption that I is fresh, name capture could occur. As a concrete example, consider the following expression:

```
(let ((x true)) (or2 false x))
```

If I is fresh (say `x1`), then this is equivalent to:

```
(let ((x true))
  (let ((x1 false))
    (if x1 x1 x)))
```

But if I is not fresh (say `x`), then the above would evaluate to false rather than true, because the reference to the outer `x` would be captured by the declaration of the inner `x`.

Part c. When a construct is defined in terms of desugaring, it is not necessary to extend the definitions of any functions that manipulate the abstract syntax trees of the language: e.g., free-variables, evaluation, substitution, type-checking, etc. In contrast, when a new construct is added as a new kind of AST node, any function manipulating the ASTs of the language must be modified.

Problem 8: Block Structure

The FOBS function declaration `index-of-bs` can be translated into the following two FOFL declarations:

```
(fun index-of-no-bs (elt lst)
  (index-loop 1 lst elt))

(fun index-loop (i L elt)
  (if (null? L)
   -1
   (if (eqv? elt (car L))
       i
       (index-loop (+ i 1) (cdr L) elt))))
```

The FOBS function declaration `cartesian-product-bs` can be translated into the following three FOFL declarations:

```
(fun cartesian-product-no-bs (lst1 lst2)
  (prod lst1 lst2))

(fun prod (lst1 lst2)
  (if (null? lst1)
   '()
    (let ((elt (car lst1)))
      (append (map-duple lst2 elt)
              (prod (cdr lst1) lst2)))))

(fun map-duple (L elt)
  (if (null? L)
   '()
    (cons (list elt (car L))
          (map-duple (cdr L) elt))))
```

Problem 9: Static vs. Dynamic Scope

Part a.

Scoping Mechanism	Value of `(sum (raise 2) 1 3)`
Lexical	$1^2 + 2^2 + 3^2 = 14$
Dynamic	$1^1 + 2^2 + 3^3 = 32$

Part b. Yes, a language can be lexically scoped without being block-structured. Block structure says that function declarations can be nested. Lexical scoping says that the meaning of free variables within functions is determined by lexical contours. Lexical scoping still has meaning even when function declarations are flat (i..e, only at top-level). In this case, the body of a function can still have free variables that reference global variables. For such functions, lexical and dynamic scoping could give different answers. Examples of lexically-scoped languages without block structure include C, Java, and the toy language FOFL.

Problem 10: Scoping

Part a. Determine the values of the following two expressions that use `fluid-bind`:

Expression	Value
`(let ((a 1)) (let ((f (lambda (x) (+ x a)))) (+ (fluid-bind a 20 (f 300)) (f 4000))))`	4321
`(let ((a 1)) (+ (fluid-bind a 20 (begin (set! a (+ a 300)) a)) a))`	321

Part b

```
(fluid-bind I Edef Ebody)

  => (let ((old I)
           (body-thunk (lambda () Ebody)))
       (begin
         (set! I Edef)
         (let ((result (body-thunk)))
           (set! I old)
           result)))
```

The purpose of `body-thunk` is to capture E_{body} at a point where no names have been introduced yet. In particular, if `(body-thunk)` were replaced by E_{body}, then E_{body} could accidentally capture the name `old`.

Part c Unlike a lexical `let`, `fluid-bind` requires some cleanup operations (restoring the old value of the variable) before it can return the value of the body expression. So the call to `(print-nums n)` in the body of `print-nums` is not tail-recursive because there is still work remaining to be done. The amount of pending work increases with n until memory is exhausted.

Part d A non-local jump out of a dynamic let will automatically restore the dynamic environment in effect at the target of the jump. However, a non-local jump out of `fluid-bind` will have the effect of not restoring the fluid-bound variable. As a concrete example, suppose that `pen-color` is a variable containing the default pen color (say it's yellow) and we execute:

```
(fluid-bind pen-color red (abort 'done))
```

Then `pen-color` will be left in the red state because the code to restore it to yellow was bypassed by the `abort.` With a dynamic `let`, however, the abort would force the system back to the default dynamic environment, in which `pen-color` would be yellow

PROBLEM 11: Scoping

H&R Block Structure, a tax software vendor, has developed a program for computing the cost of taxable items in a *dynamically scoped* imperative call-by-value version of Scheme. Their program includes the following top-level definitions:

```
(define *rate* 0.05)

(define taxed
  (lambda (amount)
    (* amount (+ 1 *rate*))))

(define with-rate
  (lambda (rate thunk)
    (let ((*rate* rate))
      (thunk))))
```

The global variable `*rate*` represents the default sales tax rate (5%). The procedure `taxed` uses the global value of `*rate*` unless it has been shadowed by a local binding of `*rate*`, such as that made by `with-rate`. This approach is more convenient than having to pass tax rates as explicit parameters throughout a large program. For example, consider the expression E_{tax}:

```
(+ (taxed 200)
   (+ (with-rate 0.075 (lambda () (taxed 1000)))
      (taxed 400)))
```

This expression evaluates to 210 + 1075 + 420 = 1705.

a. What is the value of E_{tax} in a statically-scoped version of Scheme? Explain.

b.. H&R Block Structure asks you to port their code to a *lexically-scoped* imperative call-by-value Scheme. Show how to define `with-rate` in lexically-scoped Scheme so that it has the same behavior as the above `with-rate` in a dynamically scoped mini-Scheme. *Hint*: use side effects. Also, compare with Problem 10.

Problem 12: Variables and Scoping

Consider the following expression in statically-scoped HOIL (the Higher-Order Imperative Language):

```
(bindpar ((a 20)
          (z (cell a)))
  (bind ((inc! (abs (x)
                 (seq (:= z (+ (^ z) x))
                      (^ z)))))
    (bindrec ((s (prepend b t))
              (t (map inc! s)))
      (+ (head t) (head (tail t))))))
```

Part a. The free variables in the above expression are:

- ⇨ The occurrence of `a` in `(cell a)`
- ⇨ The occurrence of `b` in `(prepend b t)`

Part b.

- ⇨ All occurrences of `z` within `(abs (x) …)` should point to the `z` declared in the `bindpar`.
- ⇨ The `x` in `(+ (^ z) x)` should point to the `x` declared in `(abs (x) …)`
- ⇨ The `t`s in `(prepend b t)`, `(head t)`, and `(tail t)` should point to the `t` declared in the `bindrec`.
- ⇨ The `s` in `(map inc! s)` should point to the `s` declared in the `bindrec`.
- ⇨ The `inc!` in `(map inc! s)` should point to the `inc!` declared in the `bind`.

Part c. Suppose that the above expression is evaluated in an environment in which

1. `map` is the usual higher-order mapping function.
2. all other free variables are initially bound to the number 1.

Give the value of the above expression under each of the following parameter passing mechanisms. If the expression loops, raises an error, or is otherwise undefined, say so.

- **call-by-value**: The expression gives an error because `(prepend b t)` requires the value of `t` before it has been defined.
- **call-by-name**: With this strategy, `(head t)` is `(head (map inc! s)`, which is `(head (map inc! (prepend 1 t))`. Evaluating this increments the cell `z` from 1 to 2 and returns 2. Next,

  `(head (tail t))`
  is equivalent to `(head (tail (map inc! s)))`
  is equivalent to `(head (tail (map inc! (prepend 1 t))`
  is equivalent to `(head (tail (map inc! (prepend 1 (map inc! s)))`
  is equivalent to `(head (tail (map inc! (prepend 1 (map inc! (prepend 1 t))))`
  is equivalent to `(head (map inc! (map inc! (prepend 1 t))))`
  is equivalent to `(inc! (inc! 1))`
  which evaluates to 4, since the cell `z` already contains 2.

  So `(+ (head t) (head (tail t)))` evaluates to 2 + 4 = 6.
- **call-by-need**: With this strategy, `s` denotes the lazy list `(1 2 3 …)` and `t` denotes the lazy list `(2 3 4 …)`, so `(+ (head t) (head (tail t)))` evaluates to 5.

Problem 13: The Aggregate Data Style of Programming

```
(define even-pct
  (lambda ()
   (letrec ((loop (lambda (n evens total)
                    (if (< n 0)
                        (/ evens total)
                        (loop (read-int)
                              (if (even? n) (+ evens 1) evens)
                              (+ total 1))))))
     (loop (read-int) 0 0))))
```

Part a. Here is a version of `even-pct` written in the signal processing style (also known as the aggregate data paradigm, also known as recursionless programming).

```
(define (even-pct)
  (let ((ints (generate (read-int)
                        (lambda (ignore) (read-int))
                        (lambda (int) (< int 0)))))
    (/ (foldr + 0 (map (lambda (x) 1)
                       (filter even? ints)))
       (foldr + 0 (map (lambda (x) 1)
                       ints)))))
```

Note that `(foldr + 0 (map (lambda (x) 1) lst)` computes the length of `lst.`

Part b. Briefly describe two advantages of writing `even-pct` in the signal processing style vs. the original style.

1. We can build the program out of mix-and-match parts that are useful for many other programs.

2. The resulting program is easier to reason about because it doesn't contain any loops.

Part c. Briefly describe two disadvantages of writing `even-pct` in the signal processing style vs. the original style.

1. The signal processing style version of `even-pct` takes more time than the monolithic loop to unwind all the abstractions involved. (However, there exist some compilation techniques for reducing this overhead.)

2. At least in a call-by-value language, the signal processing style program takes more space than the monolithic loop because it builds up intermediate lists that hold all the numbers typed in by the user. The monolithic loop is an iteration that requires only constant space.

3. When the definitions of `generate`, `map`, `filter`, and `foldr` are included, the modular definition of `even-pct` is much longer than the monolithic version. However, this doesn't take into account that the size definitions of the higher-order list operations can be amortized over all their uses.

Part d. Proponents of lazy functional programming languages claim that lazy evaluation is essential for programming in the signal processing style. Briefly explain their claim.

Lazy evaluation addresses the space drawback sketched in Part c. Lazy evaluation makes it possible to compute and communicate aggregate structures one element at a time on an as-needed basis. This makes it possible to glue together programs requiring aggregate structures that cannot fit in memory (including infinite ones!). Lazy evaluation also makes it possible to decompose loops into separate processes that perform the loop body and test for the termination condition.

PROBLEM 14 : Lazy Data

Part a. There are many ways to define the stream of all orduples. Here we describe three ways. All three use the following auxiliary function for making duples:

```
(define duple (lambda (fst snd) (list fst snd)))
```

There are many ways to define the stream of all orduples. Here we describe three ways. All three use the following auxiliary function for making duples:

```
(define next-orduple
  (lambda (dup)
    (let ((newfst (+ (first dup) 1))
          (newsnd (- (second dup) 1)))
      (if (> newfst newsnd)
          (duple 0 (+ newfst newsnd 1))
          (duple newfst newsnd)))))
```

Using `next-orduple`, `all-orduples` can be created by using `generate-stream` or `map-stream`:

```
;; Version 1
(define all-orduples
  (generate-stream
    (duple 0 0)
    next-orduple
    (lambda (dup) #f) ; this stream is infinite!
    ))

;; Version 2
(define all-orduples
  (cons-stream (duple 0 0)
    (map-stream next-orduple all-orduples)))
```

An alternative strategy is to have an auxiliary function orduples-summing-to that returns a stream of all orduples summing to a particular integer:

```
 (define orduples-summing-to
  (lambda (n)
    (letrec ((local (lambda (i)
                      (if (> i (- n i))
                          the-empty-stream
                          (cons-stream (duple i (- n i))
                                       (local (+ i 1)))))))
      (local 0))))
```

Then `all-orduples` can be defined by mapping `orduples-summing-to` over the `nats`, and appending the resulting streams:

```
(define nats
  (cons-stream 0 (map-stream (lambda (x) (+ x 1)) nats)))

;; Version 3
(define all-orduples
  (append-map-stream orduples-summing-to nats))
```

The `append-map-stream` function is implemented using the following three auxiliary functions. The stream-appending function `append-streams-delayed` assumes that its second argument is a promise, so any caller must ensure that this is the case (usually via an explicit `delay`). For example, `append-stream-of-streams` calls append-streams-delayed with an explicitly delayed second argument to avoid an infinite regress.

```
    (define append-map-stream
     (lambda (f str)
       (append-stream-of-streams
        (map-stream f str))))

   (define append-stream-of-streams
     (lambda (str)
       (if (stream-null? str)
           the-empty-stream
           (append-streams-delayed
             (head str)
             (delay (append-stream-of-streams (tail str)))))))

   (define append-streams-delayed
     (lambda (str1 delayed-str2)
       (if (stream-null? str1)
           (force delayed-str2)
           (cons-stream (head str1)
                        (append-streams-delayed (tail str1)
                                                delayed-str2)))))
```

Part b. Below is a definition of `pythagoreans` that works as follows: first, it removes all orduples whose first component is 0, since a must be > 0 in the Pythagorean triple `(a b c)` ; next it maps over all remaining duples `(a b)` a function that creates the triple `(a b (sqrt (+ (square a) (square b))))`; finally, it keeps all such triples whose third component is an integer.

```
   (define triple (lambda (fst snd thd) (list fst snd thd)))

   (define square (lambda (x) (* x x)))

   (define pythagoreans
     (filter-stream (lambda (dup)
                      (integer? (third dup)))
                    (map-stream (lambda (dup)
                                  (triple (first dup)
                                          (second dup)
                                          (sqrt (+ (square (first dup))
                                                   (square (second dup))))))
                                (filter-stream (lambda (dup)
                                                 (> (first dup) 0))
                                               all-orduples))))
```

For example, here is a list of the first 10 Pythagorean triples:

```
   (prefix 10 pythagoreans)
   ;Value: ((3 4 5) (6 8 10) (5 12 13) (9 12 15) (8 15 17) (12 16 20)
   ;        (7 24 25) (10 24 26) (15 20 25) (20 21 29))
```

Part c. The definition of `all-orduples` from Part a will not work if lists are used in place of streams because all-orduples is an infinite sequence and it is impossible to create a list with an infinite number of elements. However, it is possible to create a stream with a (conceptually) infinite number of elements, because the elements are calculated on an as-needed basis.

Problem 15: Church Pairs

```
(program (n)
  (bindpar ((cons (abs (a b) (abs (f) (f a b)))
            (car (abs (p) (p (abs (x y) x))))
            (cdr (abs (p) (p (abs (x y) y)))))
    (bindpar ((p (cons (> n 0) n))
              (q (cons (* n 2) (* n n))))
      (if (car p)
          (car q)
          (+ (cdr p) (cdr q))))))
```

Part a. Use the substitution model to prove that `(car (cons 3 4))` yields 3 for the above definitions of `cons` and `car`.

```
(car (cons 3 4))
⇨ ((abs (p) (p (abs (x y) x))) ((abs (a b) (abs (f) (f a b)) 3 4))
⇨ ((abs (p) (p (abs (x y) x))) (abs (f) (f 3 4)))
⇨ ((abs (f) (f 3 4)) (abs (x y) x))
⇨ ((abs (x y) x) 3 4))
⇨ 3
```

Part b. Use the environment model to prove that `(car (cons 3 4))` yields 3 for the above definitions of `cons` and `car`.

Part c. Would the above definitions work in a dynamically scoped version of HOFL? Explain.

No, they would not. They require that the abstraction `(abs (f) (f a b)` somehow "remember" the values of the free variables `a` and `b` -- that is, the values with which `(abs (a b) (abs (f) (f a b))` was called. Two forms of such memory for static scoping are illustrated in Part a and Part b above. But in dynamic scoping, the abstraction `(abs (f) (f a b)` has no memory and will find the values of `a` and `b` whereever it is called.

Part d. Translate the above HOFL program into the explicitly-typed HOFLEPT language. You will need to make each of `cons`, `car`, and `cdr` polymorphic. The type of `cons` should be:

```
(forall (c d)
  (-> (c d)
      (forall (e)
        (-> ((-> (c d) e)) e))))
```

A HOFLEPT version of the program appears below. Type annotations have been highlighted in boldface. Although `bindpar` does not allow types for the definitions, we have included such types as comments. To distinguish type variables from expression variables, we use capital letters for type variables and lowercase letters for expression variables.

```
(program (n)
 (bindpar ((cons ; (forall (C D)
                 ;   (-> (C D)
                 ;       (forall (E)
                 ;         (-> ((-> (C D) E)) E))))
             (pabs (C D)
               (abs ((a C) (b D))
                 (pabs (E)
                   (abs ((f (-> (C D) E)))
                     (f a b))))))
           (car ; (forall (C D)
                ;   (-> ((forall (E)
                ;          (-> ((-> (C D) E)) E)))
                ;       C))
             (pabs (C D)
               (abs ((p (forall (E) (-> ((-> (C D) E)) E))))
                 ((papp p C) (abs ((x C) (y D)) x)))))
           (cdr ; (forall (C D)
                ;   (-> ((forall (E)
                ;          (-> ((-> (C D) E)) E)))
                ;       D))
             (pabs (C D)
               (abs ((p (forall (E) (-> ((-> (C D) E)) E))) )
                 ((papp p D) (abs ((x C) (y D)) y)))))
           )
   (bindpar ((p ((papp cons bool int) (> n 0) n))
             (q ((papp cons int int) (* n 2) (* n n))))
     (if ((papp car bool int) p)
         ((papp car int int) q)
         (+ ((papp cdr bool int) p)
            ((papp cdr int int) q))))))
```

Part e. In Scheme, `cons`, `car`, and `cdr` are not only used to define general pairs, but can also be used to define lists. Is the same true in (untyped) HOFL? How about in explicitly typed HOFLEPT?

In untyped HOFL, cons/car/cdr can also be used to define lists. The unit value #u (or some other arbitrary value) can be used to represent the empty list, and a list node can be represented as a pair resulting from calling `cons`.

The same is not true in HOFLEPT. In HOFLEPT it necessary to encode the type `(listof T)` for any element type T. If we had a `pairof` type and an `eitherof` type, we could define (listof T) as:

```
(listof T) = (eitherof unit (pairof T (listof T)))
```

The type of cons is the pairof type. But we don't have an eitherof type. More importantly, listof is defined recursively, and we don't have a way in HOFLEPT of defining a recursive type.

Part f. In HOIL, the imperative version of HOFL, the above definitions can be extended to support Scheme's pair mutation operators `set-car!` and `set-cdr!`. Show how this can be done by filling out the the expressions <*`fill_i`*> below.

```
(bindpar
  ((cons (abs (a b)
           (bindpar ((a-cell (cell a))
                     (b-cell (cell b)))
            (abs (f) (f (^ a-cell)
                        (^ b-cell)
                        (abs (v) (:= a-cell v))
                        (abs (v) (:= b-cell v))
                     ))))))
   (car (abs (p) (p (abs (x y sx sy) x))))
   (cdr (abs (p) (p (abs (x y sx sy) y))))
   (set-car! (abs (p v) (p (abs (x y sx sy) (sx v)))))
   (set-cdr! (abs (p v) (p (abs (x y sx sy) (sy v)))))
   )
 expression using the above definitions)
```