

FINAL EXAM REVIEW PROBLEMS

The CS251 final is a self-scheduled final exam held during the normal final exam period. The exam is open book; you may refer to class handouts, your notes, and whatever additional materials you find useful. However, you may **not** use a computer during the exam. By the Honor Code, you are **not** allowed to talk to anyone about the details of the exam before or after taking it, until the final examination period is over.

Here is a list of topics covered in CS251 that are fair game for the final exam:

- **programming paradigms:** functional, imperative, object-oriented
- **syntax:** abstract syntax trees, free variables, substitution, desugaring
- **evaluation models and interpreters:** substitution model, environment model
- **data:** first-class functions, aggregate data programming, sum-of-product datatypes, lazy data
- **scoping:** lexical, dynamic; block structure; environment diagrams & closures
- **parameter passing:** call-by-value, call-by-name, call-by-need, call-by-reference.
- **types:** dynamic vs. static; explicit (e.g. Java, C) vs. reconstructed (e.g. OCaml, Haskell) ;
- **imperative programming:** mutable data, mutable variables, memoization; benefits and drawbacks.
- **control constructs:** tail recursion, non-local exits (`label/jump`, `call-with-current-continuation`); exceptions (`raise`, `trap`, `handle`); continuation-passing style.
- **real languages:** OCaml, Scheme, Java, C, Haskell. Note: You will be expected to read and write programs in OCaml and Scheme. You will be expected to read very simple Java, C, Haskell, Pascal programs, but will not be expected to write them.
- **toy languages:** You will be expected to read and write programs in any of these mini-languages:
 - INTEX = integer literals + arithmetic operations + program parameters
 - BINDEX = INTEX + local binding (`bind`)
 - IBEX = BINDEX + boolean literals + relational & arithmetic ops + `if`
 - HOFL = FOBS + abstractions (allowing higher-order functions) + `bindrec` (merges function and variable namespaces) + `pairs` + `lists`
 - HOILEC = HOFL + explicit mutable cells + sequencing + strings
 - HOILIC = HOILIC with implicit mutable cells for every variable; these are assigned to via `<-`.

Below are (1) the cover sheet for the Spring 2004 final and (2) problems intended to help you review material for the final exam. They are not necessarily indicative of the kinds of questions that will be asked on the exam (i.e., some review questions are more difficult/time consuming than what would be on an exam.) They also do not cover all of the above topics.

CS251 FINAL EXAM

YOUR NAME: _____

This exam has **six** problems. Each problem has several parts. The number of points for each problem and part is shown in square brackets next to the problem or part. There are **100 total points** on the exam. **The last problem (Problem 6) is worth 38 points; be sure to allocate sufficient time to work on this problem.**

Write all your answers on the exam itself. Whenever possible, show your work so that partial credit can be awarded.

The exam is open book. You may refer to class handouts, your notes, and whatever additional materials (including books) would be useful. However, you may **not** use a computer during the exam. By the Honor Code, you are **not** allowed to talk to anyone about the details of the exam before or after taking it, until the final examination period is over.

The exam includes an **appendix on p. 12 containing definitions of higher-order list functions in HOILIC** that are similar to the list functions we studied in OCaml, Scheme, and Haskell during the semester.

Please keep in mind the following tips:

- Skim the entire exam before you begin solving problems. *Work first on the problems on which you feel most confident.* **You do not need to work on the problems in the order they are presented.** In particular, since Problem 6 is worth 38 points, you might not want to leave it until the very end of the exam.
- *Try to do something on every problem* so that you can receive partial credit. For programming problems, you can receive partial credit for explaining your strategy with words and pictures.
- *Show your work*, so that you can receive partial credit even if the final answer is incorrect. If your work does not fit on the page and you must place it elsewhere, indicate where it is.
- *Allocate your time carefully.* If you are taking too long on a problem, wrap it up and move on.
- If you finish early, *go back and check your answers.*

GOOD SKILL!

The following table will be used in grading your exam:

Problem	Score
Problem 1 [9]	
Problem 2 [10]	
Problem 3 [16]	
Problem 4 [12]	
Problem 5 [16]	
Problem 6 [38]	
Total [100]	

Problem 1: OCaml Types

Consider the following sequence of function declarations in the OCaml language:

```
let test1 (x, f, g) = (x, f(x), g(x))
let test2 (x, f, g) = (x, f(x), g(f(x)))
let test3 (x, f, g) = (x, f(x), g(f(x)), f(g(x)))
let test4 (x, f, g) = (x, f(x), g(x, f(x)))
let test5 (x, f, g) = (x, f(x), g(f(x), f(g(x))))
let test6 (x, f, g) = (x, f(x), g(x, f(g(x))))
```

Part a. For each of the above function declarations, write down the type that OCaml would reconstruct for the function. If OCaml would not be able to reconstruct a type for a function, say so and explain why.

Part b.

- (1) Define a curried version of the `test1` function named `test1-curried`.
- (2) Give the type of `test1-curried`.
- (3) Below is an expression using `test1`. Show how to rewrite it using `test1-curried`:

```
test1(3, fun y -> y * 2, fun z -> z > 0)
```

Part c. Write a declaration of a function `f` that has the following OCaml type:

```
('a -> 'b list) -> ('b -> 'c list) -> ('a -> 'c list)
```

You may find it helpful to use the following OCaml list functions in your definition:

```
List.map: ('a -> 'b) -> ('a list) -> ('b list)
List.flatten ('a list list) -> ('a list)
```

Part d. Below is a `curry2` function that curries any function whose argument is a tuple of two values. What is the type of `curry2`?

```
let curry2 f = (fun x -> (fun y -> f(x,y)))
```

Part e. Define an `uncurry2` function that is the inverse of `curry2`. That is, for any curried function `f` of two arguments, `curry2(uncurry2(f))` should be indistinguishable from `f`; and for any uncurried function `g` of two arguments, `uncurry2(curry2(g))` should be indistinguishable from `g`.

Part f. While Scheme and OCaml are similar in many respects, Scheme is a dynamically typed language while OCaml is a statically typed language. Briefly discuss the advantages and disadvantages of static typing vs. dynamic typing.

Part g. While both OCaml and Java are statically typed languages, there are some key differences between the languages. Briefly describe the main differences.

Problem 2: Environment Diagrams and Mutation

Consider the following procedures in an imperative call-by-value statically-scoped Scheme:

```
(define make-updater
  (let ((n 0))
    (lambda (init update)
      (begin
        (set! n (+ n 1))
        (let ((uid n)
              (state init))
          (lambda ()
            (begin (set! state (update uid state))
                   state))))))))

(define try
  (lambda (u n)
    (if (= n 0)
        `()
        (let ((v (u)))
          (cons v (try u (- n 1)))))))

(define test
  (lambda ()
    (let* ((a (make-updater `() cons))
           (b (make-updater 1 *))
           (c (make-updater 0 +)))
      (list (try a 2) (try b 3) (try c 4)))))
```

What is the value of `(test)`? Draw environment diagrams to justify your answer. You need not draw environment frames for invocations of the `try` procedure.

Problem 3: Non-local Exits

For the purposes of this problem, a *binary tree* is either (1) a *leaf* or (2) the result of applying the `node` constructor to a left binary tree and a right binary tree. The `leaf?` predicate determines if a value is a leaf (non-node), and the selectors `left` and `right` extract the left and right subtrees of a binary tree.

Assume that `append` is a procedure that takes two lists and returns a new list containing all of the elements of the first followed by all of the elements of the second. E.g.:

```
(append '(a b c) '(d e)) returns the list (a b c d e)
```

Assume that `postpend` is a procedure that takes a list `L` and a value `V` and returns a new list containing all of the elements of `L` followed by `V`. E.g.:

```
(postpend '(a b c) d) returns the list (a b c d)
```

Consider the following `fringe` procedure, which is written in a version of call-by-value, statically-scoped Scheme supporting the `label` and `jump` constructs:

```
(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. For each of the three expressions in the following table, indicate the value of the expression. Assume that the operand expressions of a function application are evaluated in left-to-right order.

Expression	Value
<code>(fringe (node (node 'a 'b) (node 'c 'd)))</code>	
<code>(fringe (node (node 'a 2) (node 'c 'd)))</code>	
<code>(fringe (node (node 'a 'b) (node 3 'd)))</code>	
<code>(fringe (node (node 'a 2) (node 3 'd)))</code>	

Part b. Give an English specification for `fringe`.

Part c. Write an function `fringe-cps` that behaves the same as `fringe`, but uses explicit continuations rather than the implicit ones used by `fringe`.

Part d. Describe the difficulties that would be encountered in implementing `fringe` without using explicit or implicit continuations.

Problem 4: Parameter Passing

Consider the following Scheme expression:

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

For each of the following parameter-passing mechanisms, indicate the value of the above expression in a version of Scheme using that parameter-passing mechanism. Assume that all operands are evaluated in left-to-right order.

Parameter-Passing Mechanism	Value of sample expression
Call-by-value	
Call-by-reference	
Call-by-name	
Call-by-need	

Problem 5: Static vs. Dynamic Scope

Part a. Consider the following definitions in call-by-value Scheme:

```
(define (raise-to n)
  (lambda (x) (expt x n))) ; (expt x n) computes  $x^n$ 

(define (sum proc n limit)
  (if (> n limit)
      0
      (+ (proc n)
         (sum proc (+ n 1) limit))))
```

For each of the following two scoping mechanisms, indicate the value of the expression `(sum (raise-to 2) 1 3)` in a version of Scheme using that scoping mechanism:

Scoping Mechanism	Value of <code>(sum (raise-to 2) 1 3)</code>
Lexical	
Dynamic	

Part b. Suppose E is an expression in which no abstraction has free variables. Can the value of E be different in a statically-scoped and dynamically-scoped interpreter?

Part c. Can a language be lexically scoped without being block structured? Briefly explain your answer.

Problem 6: Desugaring

One way to define an `or` construct is as a user-defined procedure:

```
(define or1 (lambda (a b) (if a a b)))
```

An alternative way to define an `or` construct is via syntactic sugar:

```
(or2 E1 E2) desugars to (let ((I E1)) (if I I E2)) ; assume I fresh
```

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 E1 E2)` and `(or2 E1 E2)` interchangeable for all expressions E_1 and E_2 ?

- call-by-value
- call-by-name
- call-by-need

Part b. The desugaring for `or2` has the side condition "assume I fresh". What could go wrong with this desugaring if the side condition were omitted?

Part c. What are the advantages of defining a language construct via desugaring rather than adding it as a kernel construct of the language?

Problem 7: Block Structure

Translate each of the following two block-structured top-level HOFL function declarations into an equivalent collection of non-block-structured HOFL function declarations.

```
(def (index-of-bs elt lst)
  (bindrec ((index-loop (fun (i L)
                        (if (null? L)
                            -1
                            (if (= elt (head L))
                                i
                                (index-loop (+ i 1) (tail L))))))
    (index-loop 1 lst)))

(def (cartesian-product-bs lst1 lst2)
  (bindrec ((prod (fun (lst)
                  (if (null? lst)
                      (null)
                      (let ((elt (head lst)))
                        (bindrec ((duple (fun (b)
                                          (prepend elt
                                            (prepend b
                                              (empty))))))
                          (map-duple (fun (L)
                                      (if (null? L)
                                          (null)
                                          (prepend
                                           (duple (head L))
                                           (map-duple (tail L))))))
                            (append (map-duple lst2)
                                   (prod (tail lst))))))))
    (prod lst1)))
```

PROBLEM 8: 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 HOILIC. Their program includes the following top-level definitions:

```
(def *rate* 0.05)

(def taxed
  (fun (amount)
    (* amount (+ 1 *rate*))))

(def with-rate
  (fun (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 HOILIC? Explain.

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

Problem 9: Variables and Scoping

Consider the following expression in statically-scoped HOILEC (the Higher-Order Imperative Language with Explicit Cells):

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

Part a. Circle all of the free variable references in the above expression.

Part b. For each bound variable reference, draw an arrow from the reference to the point where the variable is declared.

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:
- call-by-name
- call-by-need

Problem 10: The Aggregate Data Style of Programming

Here's a Scheme procedure that prompts the user for a sequence of non-negative integers and returns the percentage of even integers in that sequence. The user indicates the end of the sequence by typing a negative integer:

```
(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))))
```

Assume that the nullary `read-int` procedure prompts the user (via the prompt `int>`) for a single integer and returns this integer. Then here's a sample use of `even-pct`:

```
(even-pct)
int> 3
int> 8
int> 2
int> -1
0.66666 ; Two out of the three integers were even.
```

Part a. Rewrite `even-pct` as an aggregate data style program in terms of the higher-order procedures `generate`, `map`, `filter`, and `foldr`. (See Appendix A for definitions of these higher order procedures.) You may *not* assume the existence of a `length` function for lists; if you need one, you must define it in terms of `generate`, `map`, `filter`, and `foldr`.

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

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

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

PROBLEM 11: Lazy Data

Let the term *ordered duple* (“orduple” for short) refer to a list of two non-negative integers in which the first integer is less than or equal to the second integer. E.g. (0 2), (1 2) and (2 2) are all orduples, but (-1 2) and (2 1) are not orduples. Orduple *a* is said to be less than duple *b* if either

1. (+ (first a) (second a)) is less than (+ (first b) (second b))
- or 2. (+ (first a) (second a)) is equal to (+ (first b) (second b))
but (first a) is less than (first b).

For example, the first nine orduples in order are:

(0 0) (0 1) (0 2) (1 1) (0 3) (1 2) (0 4) (1 3) (2 2)

Part a. Using Scheme streams, define an infinite sorted stream of all orduples named `all-orduples`. You may use whatever auxiliary procedures you find helpful as part of your definition, including the higher order stream operators in appendix B.

Part b. Pythagorean triples are length-3 lists of the form (a b c) where $0 < a \leq b$ and $a^2 + b^2 = c^2$. Using `all-orduples` from Part a and the stream operators from Appendix B, define an infinite stream `pythagoreans` that contains all Pythagorean triples.

You may assume that the Scheme `sqrt` function returns an integer when called on a perfect square. That is, (`sqrt 25`) returns the integer 5, not the floating point number 5.0. The Scheme predicate `integer?` tests whether a given value is an integer.

Part c. The definition of `all-orduples` from part a will not work if lists are used in place of streams. Explain why.

Problem 12: Control

Consider the following `map2` procedure in a version of Scheme supporting exception handling (via `raise/trap/handle`):

```
(define map2
  (lambda (f lst1 lst2)
    (cond
      ((and (null? lst1) (null? lst2))
       '())
      ((or (null? lst1) (null? lst2))
       (raise length (list lst1 lst2)))
      (else (cons (f (car lst1) (car lst2))
                  (map2 f (cdr lst1) (cdr lst2)))))))
```

`map2` maps a two-argument procedure over the corresponding elements of two lists:

```
> (map2 * '(1 2 3) '(4 5 6))
(4 10 18)
```

If the two lists do not have the same length, `map2` raises the `length` error.

a. One way to handle lists of different lengths is to ignore the elements of the longer list that do not correspond to elements in the shorter one. Below, write a `map2-truncate` procedure that has this behavior. For example:

```
> (map2-truncate * '(1 2 3) '(4 5 6))
(4 10 18)

> (map2-truncate * '(1 2 3 4 5) '(4 5 6))
(4 10 18)

> (map2-truncate * '(1 2 3) '(4 5 6 7 8))
(4 10 18)
```

`map2-truncate` should be expressed as an exception handler wrapped around a call to `map2`.

```
(define map2-truncate
  (lambda (f lst1 lst2)
    <your code goes here>))
```

b. Another way to handle lists of different lengths is to return the symbol `failed`. Below, write a `map2-fail` procedure that has this behavior. For example:

```
> (map2-fail * '(1 2 3) '(4 5 6))
(4 10 18)

> (map2-fail * '(1 2 3 4 5) '(4 5 6))
failed

> (map2-fail * '(1 2 3) '(4 5 6 7 8))
failed
```

`map2-fail` should be expressed as an exception handler wrapped around a call to `map2`.

```
(define map2-fail
  (lambda (f lst1 lst2)
    <your code goes here>))
```

c. In a language that supports `label` and `jump` constructs in addition to `raise` and `trap`, it is possible to simulate `handle` by using `trap` in conjunction with `label` and `jump`. Show this by writing a procedure `map2-fail2` that :

1. behaves exactly like `map-fail`;
2. is implemented by wrapping a `trap` (not `handle`) handler around a call to `map2`.

```
(define map2-fail2
  (lambda (f lst1 lst2)
    <your code goes here>))
```

Problem 13: Church Pairs

HOFL supports both lists and pairs. However, it turns out that pairs need not be primitive in HOFL; they could be implemented as suggested by the following program:

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

When called on two arguments, `a` and `b`, `cons` returns a procedure (call it `p` for pair) as a result. The pair `p` is a procedure of one argument, `f`, that calls `f` on `a` and `b`. The `car` procedure takes such a pair `p` and applies it to a function that returns the first of its two arguments, while `cdr` applies `p` to a function that returns the second of its two arguments. This representation pairs is called a **Church pair** after its inventor, the logician Alonzo Church.

Part a. Use the substitution model to prove that `(car (cons 3 4))` yields 3 for the above definitions of `cons` and `car`. (A similar argument would show that `(cdr (cons 3 4))` yields 4.)

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.

Part d. 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?

Part e. In HOILEC, the imperative version of HOFL with explicit cells, 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 (fun (a b)
          (bindpar ((a-cell (cell a))
                    (b-cell (cell b)))
                    (fun (f) (f <fill_1> <fill_2> <fill_3> <fill_4>))))
    (car (fun (p) (p (fun (x y sx sy) x))))
    (cdr (fun (p) (p (fun (x y sx sy) y))))
    (set-car! (fun (p v) (p (fun (x y sx sy) (sx v)))))
    (set-cdr! (fun (p v) (p (fun (x y sx sy) (sy v)))))
  )
  expression using the above definitions)
```

Problem 14: Loop Desugarings

Summer intern Bud Lojack has been asked to add the following `while` and `for` loop constructs to HOILIC:

`(while E_{test} E_{body})`
If E_{test} is false, returns false. If E_{test} is true, executes E_{body} and then evaluates `(while E_{test} E_{body})` again.

`(for I_{index} E_{init} E_{test} E_{update} E_{body})`
Introduces the variable `index`, which is initialized to the value of E_{init} . As long as E_{test} is true, executes E_{body} and then changes I_{index} to have the value of E_{update} . If E_{test} becomes false, the `for` loop returns the `false` value.

For example, here are two different versions of the factorial function written in terms of these loops:

```
(def (fact-while n)
  (bind ans 1
    (begin (while (> n 1)
              (begin (<- ans (* n ans))
                    (<- n (- n 1))))
            ans)))

(def (fact-for n)
  (bind ans 1
    (begin (for i 2 (<= i n) (+ i 1) (<- ans (* i ans)))
            ans)))
```

Part a. Bud realizes that both constructs can be implemented via desugaring. Here is his first attempt at a desugaring rule for `while`:

```
(while  $E_{test}$   $E_{body}$ ) desugars to
(if  $E_{test}$ 
  (begin  $E_{body}$  (while  $E_{test}$   $E_{body}$ ))
  false)
```

Bud's rule has a big problem. What is it?

Part b. Help Bud out by writing a correct desugaring rule for `while`.

Part c. The `for` construct can be implemented via a desugaring that uses the `while` construct. Give such a desugaring.

Appendix A: Definitions of Higher-Order List Operations

```
(define zip
  (lambda (lst1 lst2)
    (if (or (null? lst1) (null? lst2))
        '()
        (cons (list (car lst1) (car lst2))
              (zip (cdr lst1) (cdr lst2))))))

(define generate
  (lambda (seed next done?)
    (if (done? seed)
        '()
        (cons seed (generate (next seed) next done?)))))

(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))

(define map2
  (lambda (f lst1 lst2)
    (map (lambda (duple)
          (f (first duple) (second duple)))
         (zip lst1 lst2))))

(define filter
  (lambda (pred lst)
    (if (null? lst)
        '()
        (if (pred (car lst))
            (cons (car lst) (filter pred (cdr lst)))
            (filter pred (cdr lst))))))

(define foldr
  (lambda (binop init lst)
    (if (null? lst)
        init
        (binop (car lst) (foldr binop init (cdr lst))))))

(define foldr2
  (lambda (ternop init lst1 lst2)
    (foldr (lambda (duple result)
            (ternop (first duple) (second duple) result))
          init
          (zip lst1 lst2))))

(define foldl
  (lambda (binop init lst)
    (if (null? lst)
        init
        (foldl binop (binop (car lst) init) (cdr lst))))))

(define foldl2
  (lambda (ternop init lst1 lst2)
    (foldl (lambda (duple result)
            (ternop (first duple) (second duple) result))
          init
          (zip lst1 lst2))))
```

```

(define forall?
  (lambda (pred lst)
    (if (null? lst)
        #t
        (and (pred (car lst))
              (forall? pred (cdr lst))))))

(define forall2?
  (lambda (pred lst1 lst2)
    (forall? (on-duple pred)
              (zip lst1 lst2))))

(define exists?
  (lambda (pred lst)
    (if (null? lst)
        #f
        (or (pred (car lst))
             (exists? pred (cdr lst))))))

(define exists2?
  (lambda (pred lst1 lst2)
    (exists? (on-duple pred)
              (zip lst1 lst2))))

(define some
  (lambda (pred lst)
    (if (null? lst)
        #f
        (if (pred (car lst))
            (car lst)
            (some pred (cdr lst))))))

(define some2
  (lambda (pred lst1 lst2)
    (some (on-duple pred)
           (zip lst1 lst2))))

(define on-duple
  (lambda (f)
    (lambda (duple)
      (f (first duple) (second duple)))))

```

Appendix B: Definitions of Higher-Order Stream Operations

```

(define generate-stream
  (lambda (seed next done?)
    (if (done? seed)
        the-empty-stream
        (cons-stream seed
                      (generate-stream (next seed) next done?))))))

(define map-stream
  (lambda (f str)
    (if (stream-null? str)
        the-empty-stream
        (cons-stream (f (head str))
                      (map-stream f (tail str))))))

```



```

(define map-stream2
  (lambda (f str1 str2)
    (if (or (stream-null? str1) (stream-null? str2))
        the-empty-stream
        (cons (f (head str1) (head str2))
              (map-stream2 f (tail str1) (tail str2))))))

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

(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)))))

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

(define filter-stream
  (lambda (pred str)
    (if (stream-null? str)
        the-empty-stream
        (if (pred (head str))
            (cons-stream (head str)
                          (filter-stream pred (tail str)))
            (filter-stream pred (tail str))))))

(define foldr-stream
  (lambda (op init str)
    (if (null? str)
        init
        (op (head str)
            (foldr-stream op init (tail str))))))

(define foldl-stream
  (lambda (op init str)
    (if (stream-null? str)
        init
        (foldl-stream op (op init (head str)) (tail str))))))

```