Wellesley College   CS251 Programming Languages   Spring, 2000

FINAL EXAM REVIEW PROBLEMS

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 procedures, aggregate data programming, lazy data
- scoping: lexical, dynamic, global; block structure; environment diagrams & closures
- parameter passing: call-by-value, call-by-name, call-by-need, call-by-reference.
- types: dynamic vs. static; explicit vs. reconstructed ; monomorphic vs. polymorphic
- imperative programming: mutable data, mutable variables, memoization; non-local exits (`abort`, `label`/`jump`); benefits and drawbacks.
- real languages: Scheme, ML. (Note: You will be expected to read and write programs in Scheme. You will be expected to read ML programs and understand ML type notation, but will not be expected to write ML programs.)
- toy languages: INTEX, BINDEX, IBEX, FOFL, FOBS, HOFL, HOFLEMT, HOFLEPT, HOFLIMT, HOFLIPT, HOIL

Note: *The following problems are 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.*

Problem 1: ML Types

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

```
fun test1 (x, f, g) = (x, f(x), g(x))
fun test2 (x, f, g) = (x, f(x), g(f(x)))
fun test3 (x, f, g) = (x, f(x), g(f(x)), f(g(x)))
fun test4 (x, f, g) = (x, f(x), g(x, f(x)))
fun test5 (x, f, g) = (x, f(x), g(f(x), f(g(x))))
fun 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 ML would reconstruct for the function. If ML 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, fn y => y * 2, fn z => z > 0)
```

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

```
fun curry2 f = (fn x => (fn y => f(x,y)))
```

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

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

Problem 2: Explicit Typing

For each of the following two programs in the implicitly-typed HOFLIPT language, translate the program into the explicitly-typed HOFLEPT language.

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


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

PROBLEM 3: Environment Diagrams and Mutation

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

```scheme
(define make-counter1
  (let ((count 0))
    (lambda ()
      (lambda ()
        (begin (set! count (+ count 1))
               count)))))

(define make-counter2
  (lambda ()
    (let ((count 0))
      (lambda ()
        (begin (set! count (+ count 1))
               count)))))

(define make-counter3
  (lambda ()
    (lambda ()
      (let ((count 0))
        (begin (set! count (+ count 1))
               count)))))

(define test-counters
  (lambda (make-counter)
    (let ((a (make-counter))
          (b (make-counter)))
      (list (a) (b) (a)))))
```

a. For each of the following expressions, (1) give the value of the expression and (2) draw an environment diagram that justifies why the expression has that value. You should assume that all operands are evaluated in left-to-right order.

i. `(test-counters make-counter1)`

ii. `(test-counters make-counter2)`

iii. `(test-counters make-counter3)`

b. Which, if any, of the four procedures defined above could be defined in Pascal? Explain.

Problem 4: Non-local Exits

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
`(fringe (node (node 'a 'b) (node 'c 'd)))`	
`(fringe (node (node 'a 2) (node 'c 'd)))`	
`(fringe (node (node 'a 'b) (node 3 'd)))`	
`(fringe (node (node 'a 2) (node 3 'd)))`	

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

Part c. Describe the difficulties that would be encountered in implementing `fringe` without `label` and `jump`.

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	
Call-by-reference	
Call-by-name	
Call-by-need	

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	
Call-by-reference	
Call-by-name	
Call-by-need	

Problem 7: 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 8: Block Structure

Translate each of the following two block-structured top-level FOBS function declarations into an equivalent collection of FOFL (non-block-structured) function declarations. Recall that a collection of top-level `fun` declarations in FOBS desugar into a `funrec`, so that each of the following function declarations is recursively defined.

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

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

Problem 9: Static vs. Dynamic Scope

a [6]. 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 `(sum (raise 2) 1 3)`
Lexical	
Dynamic	

b [2]. Can a language be lexically scoped without being block structured? Briefly explain your answer.

Problem 10: Scoping

Renowned naming expert Dan Emmet Schoop is experimenting with a new binding construct called `fluid-bind`. Dan writes the following specification for his construct:

```
(fluid-bind I E_def E_body)
```
Temporarily assign I to the value of `E_def` during the evaluation of `E_body` and then reset I to its original value. Returns the value of `E_body`. Signals an error if `I` is not already bound in the enclosing lexical context.

Dan adds his construct to an interpreter for statically-scoped imperative Scheme by extending the `env-eval` function with the following clause:

```
((fluid-bind? exp)
 (let ((name (fluid-bind-name exp))    ; Extracts I
       (def (fluid-bind-def exp))      ; Extracts E_def
       (body (fluid-bind-body exp)))   ; Extracts E_body
   (let ((cell (env-lookup name env))) ; Gives either cell or unbound token
     (if (unbound? cell)
         (error "Unbound" name)) ; Complain if NAME not bound
         (let ((old (cell-ref cell)))  ; Save old value of I
           (begin
              (cell-set! cell (env-eval def env)) ; Install new value of I
              (let ((result (env-eval body env)))
                (begin
                   (cell-set! cell old) ; Restore old value of I
                   result))))))))
```

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)))))```	
```(let ((a 1))` ` (+ (fluid-bind a 20` ` (begin` ` (set! a (+ a 300))` ` a))` ` a))```	

Part b. Dan could have defined `fluid-bind` as syntactic sugar for other Scheme constructs. Below, give a desugaring rule that indicates how `fluid-bind` can be written in terms of other Scheme constructs. Write your desugaring rule as a rewrite rule whose form is similar to the following desugaring rule for `let`:

```
(let ((I Edef) ...) Ebody)
  desugars to ((lambda (I ...) Ebody) Edef ...)
```

You should strive to avoid accidental name capture.

Part c. Dan's friend Bud Lojack writes the following `print-nums` procedure in Dan's extended version of Scheme.

```
(define (print-nums n)
  (begin
    (print n)
    (fluid-bind n (+ n 1)
      (print-nums n))))
```

Bud evaluates `(print-nums 0)`, and watches as the numbers 0, 1, 2, etc. are displayed on the screen. However, Bud is surprised when his procedure eventually crashes due to a lack of stack space. "I thought Scheme was properly tail recursive!" he exclaims. "Why doesn't my procedure behave like an infinite loop?" Kindly explain to Bud why his program has run out of stack space.

Part d Dan points out that in many situations `fluid-bind` acts like a dynamically scoped version of `let` with a single binding. However, he notes that `fluid-bind` and a dynamically scoped `let` can give different behavior in the case where a non-local exit (such as `abort`, `jump` or Scheme's `error` construct) is encountered during the evaluation of the body expression. Briefly explain what Dan means by this comment.

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)

                  (begin (:= z (+ (^ z) x))

                         (^ z)))))

      (bindrec ((s (cons 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 13: 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 a signal processing 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 signal processing 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 14 : Lazy Data

Part a. 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)`

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 $0 < a \le b$ 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.

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

Problem 15: Church Pairs

Although HOFL supports lists, it does not support Scheme-like pairs that can glue together any two values. However, it is possible to implement Scheme-like pairs as HOFL functions, as illustrated by the following HOFL program:

```
(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))))))
```

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. 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 (a b)
  (-> (a b)
      (forall (c)
        (-> ((-> (a b) c)) c))))
```

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?

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 <fill_1> <fill_2>  <fill_3> <fill_4>)))))
   (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)
```

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 map-append-stream
  (lambda (f str)
    (if (stream-null? str)
        the-empty-stream
        (append-streams (f (head str))
                        (map-append-streams f (tail 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)))))
```