CS251 Programming Languages **Handout #52**
Prof. Lyn Turbak **May 11, 2007**
Wellesley College

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.
- **real languages**: OCaml, Scheme, Java, Haskell, C. Note: You will be expected to read and write programs in OCaml. You will be expected to read *very* simple Java, Haskell, and C 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-languges:
  - INTEX = integer literals + arithmetic operations + program parameters
  - BINDEX = INTEX + local binding (`bind`)
  - VALEX = BINDEX + IF + booleans/characters/symbols/strings/lists
  - FOFL = VALEX + top-level function definitions.
  - FOBS = VALEX + FUNREC (nested functions definitions)
  - HOFL = VALEX + abstractions (allowing higher-order functions) + BINDREC (merges function and variable namespaces)
  - HOILEC = HOFL + explicit mutable cells (manipulated by CELL, ^, :=)
  - HOILIC = HOILIC with implicit mutable cells for every variable; these are assigned to via `<-`.

The rest of this handout consists of

1. The cover sheet for the Spring 2007 final
2. The appendix for the Spring 2007 final
3. Problems intended to help you review material for the final exam. These 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.

Note: the final exam is likely to take the full 2.5 hours. You should focus your energy on those parts that are most likely to earn you the most points. If you feel that a part is taking too long, move on to another part.

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 32 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. 13 containing definitions of higher-order list functions in HOILIC** that are similar to the list functions we studied in OCaml 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 32 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 [9]	
Problem 3 [16]	
Problem 4 [20]	
Problem 5 [14]	
Problem 6 [32]	
Total [100]	

APPENDIX A: HOILIC Pair and List Functions

```
(def (pair x y) (list x y))
(def (fst p) (nth 1 p)) ;; Assume NTH is a primop
(def (snd p) (nth 2 p)) ;; Assume NTH is a primop

(def (gen next done? seed)
  (if (done? seed)
      (empty)
      (prep seed (gen next done? (next seed)))))

(def (map f xs)
  (if (empty? xs)
      (empty)
      (prep (f (head xs))
       (map f (tail xs)))))

(def (map2 f xs ys)
  (if (|| (empty? xs) (empty? ys))
      (empty)
      (prep (f (head xs) (head ys))
       (map2 f (tail xs) (tail ys)))))

(def (filter pred xs)
  (if (empty? xs)
      (empty)
      (if (pred (head xs))
     (prep (head xs) (filter pred (tail xs)))
          (filter pred (tail xs)))))

(def (foldr binop init xs)
  (if (empty? xs)
      init
      (binop (head xs)
        (foldr binop init (tail xs)))))

(def (foldl binop init xs)
  (if (empty? xs)
      init
      (foldl binop (binop (head xs) init) (tail xs))))

(def (zip xs ys)
  (map2 pair xs ys))

(def (unzip pairs)
  (pair (map fst pairs)
        (map snd pairs)))
```

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 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 functions in call-by-value, statically-scoped HOILIC:

```
(def make-updater
  (bind n 0
    (fun (init update)
      (seq
        (<- n (+ n 1))
        (bindpar ((uid n)
                  (state init))
          (fun ()
            (seq (<- state (update uid state))
                 state)))))))

 (def (test)
   (bindseq ((a (make-updater #e (fun (x y) (prep x y))))
             (b (make-updater 1 (fun (x y) (* x y))))
             (c (make-updater 0 (fun (x y) (- x y)))))
     (list (list (a) (a))
           (list (b) (b) (b))
           (list (c) (c) (c) (c)))))
```

What is the value of `(test)`? Draw environment diagrams to justify your answer.

Problem 3: Maximum

Consider the following three HOILIC constructs for defining the maximum of two integers:

`(def (max1 a b) (if (>= a b) a b))`

`(max2` E_1 E_2`)` ***desugars to*** `(if (>=` E_1 E_2`)` E_1 E_2`)`,
where I_1 *and* I_2 *are fresh*

`(max3` E_1 E_2`)` ***desugars to*** `(bindseq ((`I_1 E_1`) (`I_2 E_2`) (if (>=` I_1 I_2`)` I_1 I_2`)`,
where I_1 *and* I_2 *are fresh*

Part a: For each of the following parameter-passing mechanisms, answer the following question:

> Are there any expressions E_1 and E_2 such that `(max1` E_1 E_2`)` and `(max2` E_1 E_2`)` evaluate differently? If so, give an example of E_1 and E_2 and show how the two expressions evaluate differently. If not, explain why not.

i. Call-by-value
ii. Call-by-name
iii. Call-by-lazy

Part b: Repeat Part a, comparing `(max1` E_1 E_2`)` and `(max3` E_1 E_2`)` instead.

Part c: Consider an alternative desugaring for `max3` that does not use fresh variables:

`(max3′` E_1 E_2`)` ***desugars to*** `(bindseq ((a` E_1`) (b` E_2`) (if (>= a b) a b)`

Write a call-by-value HOILIC expression that evaluates differently if `max3` is replaced by `max3′`.

Part d: Consider the following ways to find the maximum of a non-empty sequence of integers:

`(maxseq1` E`)` ***desugars to*** E
`(maxseq1` E_1 E_2 … E_n`)` ***desugars to*** `(`*max2* E_1 `(maxseq1` E_2 … E_n`))`

`(maxseq2` E_1 E_2 … E_n`)` ***desugars to*** `(foldr` E_{binop} E_{init} E_{list}`)`

i. Assuming that `foldr` is defined as usual (see Appendix A), give definitions of E_{binop}, E_{init}, and E_{list} such that `maxseq2` finds the maximum integer value of the expressions E_1 E_2 … E_n.
ii. Which do you think is better: `maxseq1` or `maxseq2`? Justify your answer.

Part e: Consider the following four ways to add new functionality to a language. List them in order from most difficult to least difficult:

i. Add a new primitive to the language (like `+`)
ii. Add a new construct to the language kernel (like `if`)
iii. Add a new function to the language (like `max1` above)
iv. Add a new syntactic sugar construct to the language (like `max2` above)

Problem 4: Parameter Passing

Consider the following HOILIC expression:

```
(bind a 1
  (bind ((inc! (fun ()
                 (seq (<- a (+ a 1))
                      a)))
         (f (fun (y z)
              (seq (<- 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 HOILIC 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 HOILIC:

```
(def (raise-to n)
  (fun (x) (expt x n))) ; Assume (expt x n) computes xⁿ

(def (sum f n limit)
  (if (> n limit)
      0
      (+ (f n)
         (sum f (+ 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	

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: The Aggregate Data Style of Programming

Assume that HOILIC is extended with a nullary `read-int` primitive that prompts the user for an integer (using the prompt *`int>`*) and returns the integer entered by the user. Then here is a HOILIC function that prompts the user for a sequence of non-negative integers and returns the percentage of even integers in that sequence. The user specifies the end of the sequence by entering a negative number:

```
(def (even-pct)
  (bindrec ((loop (fun (n evens total)
                   (if (< n 0)
                       (f/ evens total) ; f/ is floating point division
                       (loop (read-int)
                             (if (even? n) (+ evens 1) evens)
                             (+ total 1))))))
     (loop (read-int) 0 0))))
```

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 `gen`, `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 `gen`, `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 aggregate data processing style. Briefly explain their claim.

PROBLEM 7: Scoping and Imperative Programming

H&R Block Structure, a tax software vendor, has developed a program for computing the cost of taxable items in a *dynamically-scoped*, 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 *statically-scoped*, call-by-value HOILIC. Show how to define `with-rate` in statically-scoped HOILIC so that it has the same behavior as the above `with-rate` in a dynamically scoped HOILIC. *Hint*: use side effects.

Problem 8: 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 `#f`. 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 I_{index}, which is initialized to the value of E_{init}. As long a 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 `#f`.

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

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

(def (fact-for n)
  (bind ans 1
    (seq (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} `(seq` E_{body} `(while` E_{test} E_{body}`)) #f)`

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.

Problem 9: Church Pairs

HOFL supports lists but not pairs. However, Scheme-like pairs can be implemented as follows:

```
(def (cons a b) (fun (f) (f a b)))
(def (car p) (p (fun (x y) x)))
(def (cdr p) (p (fun (x y) y)))
```

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.

Here is a sample use of such pairs:

```
(def (pair-test n)
  (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`. (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 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.

```
(def (cons a b)
  (bindpar ((a-cell (cell a))
            (b-cell (cell b)))
    (fun (f) (f <fill_1> <fill_2>  <fill_3> <fill_4>))))

(def (car p) (p (fun (x y sx sy) x)))
(def (cdr p) (p (fun (x y sx sy) y)))
(def (set-car! p v) (p (fun (x y sx sy) (sx v))))
(def (set-cdr! p v) (p (fun (x y sx sy) (sy v))))
```

Problem 10: Variables, Scoping, and Parameter Passing

Consider the following expression in statically-scoped HOILIC:

```
(bindpar ((a 20)
          (z a))
  (bind add! (abs x (seq (<- z (+ z x)) z))
    (bindrec ((s (prep b t))
              (t (map add! 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 11: Lazy Data

Let the term ***ordered duple*** (“orduple” for short) refer to a pair 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 orduple `b` if either

1. `(+ (fst a) (snd a))` is less than `(+ (fst b) (snd b))`

or 2. `(+ (fst a) (snd a))` is equal to `(+ (fst b) (snd b))` but `(fst a)` is less than `(fst 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 HOILIC 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 the existence of a HOILIC `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 HOILIC predicate `int?` 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.

APPENDIX A: HOILIC Pair and List Functions

See p. 3 of this handout

APPENDIX B: HOILIC Higher-Order Stream Operations

```
(def (sgen seed next done?)
  (if (done? seed)
      (sempty)
      (sprep seed (sgen (next seed) next done?))))

(def (smap f str)
  (if (sempty? str)
      str
      (sprep (f (shead str)) (smap f (stail str)))))

(def (smap2 f str1 str2)
  (if (|| (sempty? str1) (sempty? str2))
      (sempty)
      (sprep (f (shead str1) (shead str2))
             (smap2 f (stail str1) (stail str2)))))

(def (sappend str1 str2)
  (if (sempty? str1)
      str2
      (sprep (shead str1)
             (sappend (stail str1) str2))))

(def (sappend-delayed str1 delayed-str2)
  (if (sempty? str1)
      (force delayed-str2)
      (sprep (shead str1)
             (sappend-delayed (stail str1) delayed-str2))))

(def (sappend-stream-of-streams str)
  (if (sempty? str)
      str
      (sappend-delayed (shead str)
                       (delay (sappend-stream-of-streams (stail str))))))

(def (sappend-map f str) (sappend-stream-of-streams (smap f str)))

(def (sfilter pred str)
  (if (sempty? str)
      str
      (if (pred (shead str))
          (sprep (shead str) (sfilter pred (stail str)))
          (sfilter pred (stail str)))))

(def (sfoldr binop init str)
  (if (sempty? str)
      init
      (binop (shead str)
             (sfoldr binop init (stail str)))))

(define (sfoldl binop init str)
  (if (sempty? str)
      init
      (sfoldl binop (binop init (shead str)) (stail str)))))
```