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 and closures
- **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.
- **mini-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).
  - **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, ^, :) + begin (sequencing).
  - **HOILIC** = HOFL + implicit mutable cells for every variable; these are assigned to via <-.
  - **HOFLMT** = HOFL + explicit types.

The rest of this handout consists of 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.
Problem 1: OCaml Types

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

```ocaml
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))))
```

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.

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`:

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

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)`

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

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

What is the type of `curry2`?

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`.

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.

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:

\[
\text{(def make-updater}
\begin{align*}
&\text{(bind n 0)} \\
&\text{(fun (init update)} \\
&\text{(seq)} \\
&\text{(<- n (+ n 1))} \\
&\text{(bindpar ((uid n)} \\
&\text{(state init))} \\
&\text{(fun ()} \\
&\text{(seq (<- state (update uid state))} \\
&\text{state))))))) \\
\end{align*}
\]

\[
\text{(def (test)} \\
\begin{align*}
&\text{(bindseq ((a (make-updater #e (fun (x y) (prep x y)))))} \\
&\text{(b (make-updater 1 (fun (x y) (* x y)))))} \\
&\text{(c (make-updater 0 (fun (x y) (- x y)))))} \\
&\text{(list (list (a) (a))} \\
&\text{(list (b) (b) (b))} \\
&\text{(list (c) (c) (c) (c))))) \\
\end{align*}
\]

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

Problem 3: Parameter Passing

Consider the following HOILIC expression:

\[
\text{(bind a 1} \\
\text{(bind ((inc! (fun ()} \\
\text{(seq (<- a (+ a 1))} \\
\text{a)))} \\
\text{(f (fun (y z)} \\
\text{(seq (<- y (+ y 3))} \\
\text{(* a (* z z))))))} \\
\text{(f a (inc!))))}
\]

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.

<table>
<thead>
<tr>
<th>Parameter-Passing Mechanism</th>
<th>Value of sample expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call-by-value</td>
<td></td>
</tr>
<tr>
<td>Call-by-reference</td>
<td></td>
</tr>
<tr>
<td>Call-by-name</td>
<td></td>
</tr>
<tr>
<td>Call-by-need</td>
<td></td>
</tr>
</tbody>
</table>
Problem 4: Maximum

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

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

(max2 E₁ E₂) desugars to (if (>= E₁ E₂) E₁ E₂), where I₁ and I₂ are fresh

(max3 E₁ E₂) desugars to (bindseq ((I₁ E₁) (I₂ E₂)) (if (>= I₁ I₂) I₁ I₂)), where I₁ and I₂ are fresh.

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

Are there any expressions E₁ and E₂ such that (max1 E₁ E₂) and (max2 E₁ E₂) evaluate differently? If so, give an example of E₁ and E₂ and show how the two expressions evaluate differently. If not, explain why not.

1. Call-by-value
2. Call-by-name
3. Call-by-lazy

b: Repeat Part a, comparing (max1 E₁ E₂) and (max3 E₁ E₂) instead.

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

(max3' E₁ E₂) desugars to (bindseq ((a E₁) (b E₂)) (if (>= a b) a b))

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

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

(maxseq1 E) desugars to E
(maxseq1 E₁ E₂ ... ) desugars to (max2 E₁ (maxseq1 E₂ ... ))
(maxseq2 E₁ ... Eₙ) desugars to (foldr Ebinop Einit Elist)

1. Assuming that foldr is defined as usual (see Appendix A), give definitions of Ebinop, Einit, and Elist such that maxseq2 finds the maximum integer value of the expressions E₁, ..., Eₙ.

2. Which do you think is better: maxseq1 or maxseq2? Justify your answer.

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

- Add a new primitive to the language (like +)
- Add a new construct to the language kernel (like if)
- Add a new function to the language (like max1 above)
- Add a new syntactic sugar construct to the language (like max2 above)
Problem 5: Static vs. Dynamic Scope

a: Consider the following definitions in call-by-value HOFL:

\[
\begin{align*}
  (\text{def} \ (\text{raise-to} \ n) \\
  \quad (\text{fun} (x) (\text{expt} x n)) \ ; \text{Assume} \ (\text{expt} x n) \text{ computes } x^n
\end{align*}
\]

\[
\begin{align*}
  (\text{def} \ (\text{sum} f \ n \ \text{limit}) \\
  \quad (\text{if} \ (> n \ \text{limit}) \\
  \quad \quad 0 \\
  \quad \quad (+ (f n) \\
  \quad \quad \quad (\text{sum} f (+ n 1) \ \text{limit})))
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Scoping Mechanism</th>
<th>Value of ((\text{sum} \ (\text{raise-to} \ 2) \ 1 \ 3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td></td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
</tr>
</tbody>
</table>

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?

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

Problem 6: 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:

\[
\begin{align*}
  (\text{def} \ \astrate^* \ 0.05) \\
  (\text{def} \ \text{taxed} \\
  \quad (\text{fun} \ (\text{amount}) \\
  \quad \quad (* \ \text{amount} (+ 1 \ \astrate^*))))
\end{align*}
\]

\[
\begin{align*}
  (\text{def} \ \text{with-rate} \\
  \quad (\text{fun} \ (\text{rate} \ \text{thunk}) \\
  \quad \quad (\text{let} ((\astrate^* \ \text{rate})) \\
  \quad \quad \quad \text{thunk})))
\end{align*}
\]

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

\[
\begin{align*}
  (+ \ (\text{taxed} \ 200) \\
  \quad (+ \ (\text{with-rate} \ 0.075 \ (\lambda () \ (\text{taxed} \ 1000))) \\
  \quad \quad (\text{taxed} \ 400)))
\end{align*}
\]

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 \(\text{with-rate}\) in statically-scoped HOILIC so that it has the same behavior as the above \(\text{with-rate}\) in a dynamically scoped HOILIC. \textit{Hint:} use side effects.
Problem 7: The Applicative Style of Programming

Assume that HOILIC is extended with a nullary \texttt{read-int} primitive that prompts the user for an integer (using the prompt \texttt{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:

\begin{verbatim}
(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)))))
\end{verbatim}

Here's a sample use of \texttt{even-pct}:

\begin{verbatim}
(even-pct)
int> 3
int> 8
int> 2
int> -1
0.66666 ; Two out of the three integers were even
\end{verbatim}

\textbf{a:} Rewrite even-pct as an applicative style program in terms of the higher-order procedures \texttt{gen}, \texttt{map}, \texttt{filter}, and \texttt{foldr}. (See Appendix A for definitions of these higher order procedures.) You may not assume the existence of a \texttt{length} function for lists; if you need one, you must define it in terms of \texttt{gen}, \texttt{map}, \texttt{filter}, and \texttt{foldr}.

\textbf{b:} Briefly describe two advantages of writing even-pct in the applicative style vs. the original style.

\textbf{c:} Briefly describe two disadvantages of writing even-pct in the aggregate data style vs. the original style.

\textbf{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 8: Loop Desugarings

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

\[(\text{while } E_{\text{test}} E_{\text{body}})\]
If \(E_{\text{test}}\) is false, returns \#f. If \(E_{\text{test}}\) is true, executes \(E_{\text{body}}\) and then evaluates \((\text{while } E_{\text{test}} E_{\text{body}})\) again.

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

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

\[
(\text{def } (\text{fact-while } n) \\
(\text{bind ans 1} \\
(\text{seq (while (> n 1)} \\
(\text{seq (<- ans (* n ans))} \\
(\text{<- n (- n 1)))})) \\
ans))
\]

\[
(\text{def } (\text{fact-for } n) \\
(\text{bind ans 1} \\
(\text{seq (for i 2 (<= i n) (+ i 1) (<- ans (* i ans)))} \\
ans)))
\]

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

\[(\text{while } E_{\text{test}} E_{\text{body}}) \text{ desugars to } (\text{if } E_{\text{test}} (\text{seq } E_{\text{body}} (\text{while } E_{\text{test}} E_{\text{body}})) \#f)\]

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

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

c: The for construct can be implemented via a desugaring that uses the while construct. Give such a desugaring.
Problem 9: Church Pairs

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

\[
\begin{align*}
(\text{def} & \ (\text{cons} \ a \ b) \ (\text{fun} \ (f) \ (f \ a \ b))) \\
(\text{def} & \ (\text{car} \ p) \ (p \ (\text{fun} \ (x \ y) \ x))) \\
(\text{def} & \ (\text{cdr} \ p) \ (p \ (\text{fun} \ (x \ y) \ y)))
\end{align*}
\]

When called on two arguments, \(a\) and \(b\), \text{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 \text{car} procedure takes such a pair \(p\) and applies it to a function that returns the first of its two arguments, while \text{cdr} applies \(p\) to a function that returns the second of its two arguments. This representation pairs is called a \textit{Church pair} after its inventor, the logician Alonzo Church.

Here is a sample use of such pairs:

\[
\begin{align*}
(\text{def} & \ (\text{pair-test} \ n) \\
 & \ (\text{bindpar} \ ((p \ (\text{cons} \ (> \ n \ 0) \ n)) \\
 & \ \ \ \ \ \ \ (q \ (\text{cons} \ (* \ n \ 2) \ (* \ n \ n)))) \\
 & \ (\text{if} \ (\text{car} \ p) \\
 & \ (\text{car} \ q) \\
 & \ (+ \ (\text{cdr} \ p) \ (\text{cdr} \ q))))
\end{align*}
\]

\textbf{a:} Use the substitution model to prove that \((\text{car} \ (\text{cons} \ 3 \ 4))\) yields 3 for the above definitions of \text{cons} and \text{car}. (A similar argument would show that \((\text{cdr} \ (\text{cons} \ 3 \ 4))\) yields 4.)

\textbf{b:} Use the environment model to prove that \((\text{car} \ (\text{cons} \ 3 \ 4))\) yields 3 for the above definitions of \text{cons} and \text{car}.

\textbf{c:} Would the above definitions work in a dynamically scoped version of HOFZ? Explain.

\textbf{d:} Is it possible to anotate the above definitions of \text{cons}, \text{car}, and \text{cdr} with explicit types in the HOFLEMT language so that they work in all situations? If so, give the HOFLEMT definitions of these functions. If not, explain why not.

\textbf{e:} In HOILEC, the imperative version of HOFZ with explicit cells, the above definitions can be extended to support Scheme’s pair mutation operators \text{set-car!} and \text{set-cdr!}, which can change the values stored in the first or second slot of a pair, respectively. Show how this can be done by filling out the the expressions \(E_i\) below.

\[
\begin{align*}
(\text{def} & \ (\text{cons} \ a \ b) \\
 & \ (\text{bindpar} \ ((a\text{-cell} \ (\text{cell} \ a)) \\
 & \ \ \ \ \ (b\text{-cell} \ (\text{cell} \ b))) \\
 & \ (\text{fun} \ (f) \ (f \ E_1 \ E_2 \ E_3 \ E_4)))
\end{align*}
\]

\[
\begin{align*}
(\text{def} & \ (\text{car} \ p) \ (p \ (\text{fun} \ (x \ y \ sx \ sy) \ x))) \\
(\text{def} & \ (\text{cdr} \ p) \ (p \ (\text{fun} \ (x \ y \ sx \ sy) \ y))) \\
(\text{def} & \ (\text{set-car!} \ p \ v) \ (p \ (\text{fun} \ (x \ y \ sx \ sy) \ (sx \ v)))) \\
(\text{def} & \ (\text{set-cdr!} \ p \ v) \ (p \ (\text{fun} \ (x \ y \ sx \ sy) \ (sy \ v))))
\end{align*}
\]
Problem 10: 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 and only if either of the following conditions holds:

1. $\left( + \left( \text{fst } a \right) \left( \text{snd } a \right) \right)$ is less than $\left( + \left( \text{fst } b \right) \left( \text{snd } b \right) \right)$; or

2. $\left( + \left( \text{fst } a \right) \left( \text{snd } a \right) \right)$ is equal to $\left( + \left( \text{fst } b \right) \left( \text{snd } b \right) \right)$ but $\left( \text{fst } a \right)$ is less than $\left( \text{fst } b \right)$.

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)

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.

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.

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