CIS 624, Fall 2015, Midterm Examination
11 November 2015

Please do not turn the page until everyone is ready.

Rules:

• The exam is closed-book, limited notes as explained in class and on Piazza.

• Please stop promptly at 17:20.

• You can rip apart the pages, but please write your name on each page if you do that.

• There are 100 points total, distributed unevenly among 5 questions (which have multiple parts).
  Optional (extra) credit is clearly marked and is in addition to the 100 points for required questions.

Advice:

• Read questions carefully. Understand a question before you start writing.

• Write down thoughts and intermediate steps so you can get partial credit.

• The questions are not necessarily in order of difficulty. Skip around. In particular, make sure you get to all the problems.

• If you have questions, ask.

• Relax. You are here to learn.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6 optional</td>
<td>+8</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100 (+8)</td>
<td></td>
</tr>
</tbody>
</table>
Name:__________________________

For your reference:

IMP syntax as presented in class:

\[ s ::= \text{skip} \mid x := e \mid s \mid \text{if} \ e \ s \ \text{while} \ e \ s \]

\[ e ::= c \mid x \mid e + e \mid e \ast e \]

\(c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}\)

\(x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \ldots\}\)

IMP large-step semantics as presented in class:

\[
\begin{array}{c|c}
\text{CONST} & \text{VAR} \\
\hline
H ; e \downarrow & s ; e \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{ADD} & H ; e_1 \downarrow c_1 & H ; e_2 \downarrow c_2 & H ; e_1 + e_2 \downarrow c_1 + c_2 \\
\hline
\text{MULT} & H ; e_1 \downarrow c_1 & H ; e_2 \downarrow c_2 & H ; e_1 \ast e_2 \downarrow c_1 \ast c_2 \\
\end{array}
\]

IMP small-step semantics as presented in class:

\[
\begin{array}{c|c|c|c|c}
\text{ASSIGN} & \text{SEQ1} & \text{SEQ2} \\
\hline
H ; e \downarrow c & H ; \text{skip}; s \rightarrow H ; s & H ; s_1 \rightarrow H' ; s'_1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
H ; e \downarrow c & H ; e \downarrow c & H ; e \downarrow c & H ; e \downarrow c \\
\hline
H ; \text{if} e \ s_1 \ s_2 \rightarrow H ; s_1 & H ; \text{while} e \ s_2 \rightarrow H ; s_2 & H ; \text{if} e \ (s \text{while} e \ s) \rightarrow H & H ; e \downarrow c \\
\end{array}
\]

Call-by-value, left-to-right evaluation language presented in class:

\[ e ::= \lambda x. e \mid e \mid e \mid c \]

\[ v ::= \lambda x. e \mid c \]

\[ e \rightarrow e' \]

\[ (\lambda x. e) \ v \rightarrow e[v/x] \]

\[ e_1 \rightarrow e'_1 \]

\[ e_2 \rightarrow e'_2 \]

\[ e_2 \rightarrow e'_2 \]

\[ (\lambda y. e_1)[e/x] = \lambda y. e'_1 \]

\[ e_1[e/x] = c'_1 \quad y \neq x \quad y \notin \text{FV}(e) \]

\[ e_1[e/x] = c'_1 \quad e_2[e/x] = c'_2 \quad e_1[e/x] = e'_1 \]

\[ (e_1 e_2)[e/x] = e'_1 e'_2 \]

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash c : \text{int} \]

\[ \Gamma \vdash \chi : \Gamma(x) \]

\[ \Gamma \vdash x : \tau_1 \vdash e : \tau_2 \]

\[ \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \]

\[ \Gamma \vdash e_1 e_2 : \tau_1 \]

• If \( \vdash e : \tau \) and \( e \rightarrow e' \), then \( \vdash e' : \tau \).

• If \( \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) such that \( e \rightarrow e' \).

• If \( \Gamma, x: \tau' \vdash e : \tau \) and \( \Gamma \vdash e' : \tau' \), then \( \Gamma \vdash e[e'/x] : \tau \).
1. (24 points) OCaml and functional programming; continues on next page.

(a) (4 pts) Write OCaml code to define a new type `binary_tree` by filling in the blanks below. Each node of the binary tree stores an integer value.

```ocaml
type binary_tree = ...
```

(b) (4 pts) Write OCaml code to create the following tree:

```
OCaml code:
let mytree = ...
```

(c) (4 pts) Now write the OCaml code defining a binary tree type that has an integer in each internal (non-leaf) node.

(d) (6 pts) Write an OCaml function that prints the contents of a tree of the type defined in (c) as a comma-separated list of integers. Use preorder traversal (recall that preorder visits each node, then the left subtree, then the right subtree).
Name: ________________________________

1. (Continued)
(e) (6 pts) Consider the following OCaml code.

```ocaml
let fix f = (fun ('X x) -> f(x ('X x))) ('X(fun ('X x) y -> f(x ('X x)) y));;

let myfun f = function
    | 0 -> 1
    | n -> (-1) * n * f(n-1) ;;

print_string (string_of_int (fix myfun 3) ^ "\n");
```

i. What output does it produce?

ii. What is the type of `myfun`?

iii. What kind of function is `fix` and why is it needed in this case?

Solution:

(a) type binary_tree = Leaf of int | Node of int * binary_tree * binary_tree

(b) let atree =
    Node(3, (* root *)
        Leaf 1, (* left subtree *)
    Node(7, (* right subtree *)
        Node(6, Leaf 4, Leaf 5),
        Leaf 8))

(c) type binary_tree = Empty | Node of int * binary_tree * binary_tree;;

(d) let rec print t =
    match t with
        | Empty -> print_string "";
    | Node(i,l,r) ->
        print_string ((string_of_int i) ^ ",");
        print l;
        print r

(e) i. -6
    ii. val myfun : (int -> int) -> int -> int = <fun>
    iii. fix is a combinator and is used to enable recursion.
2. (14 points) This problem extends IMP statements with this strange new syntax and small-step evaluation rules:

\[ s ::= \ldots | s \# s \]

\[
\begin{align*}
H ; \text{skip} \# s & \rightarrow H ; s \quad & H ; s_1 \rightarrow H' ; s'_1 \\
H ; s_1 \# s_2 \rightarrow H' ; s'_1 \# s'_2
\end{align*}
\]

(a) (5 points) Explain in 1–3 informal but precise English sentences the meaning of \( s_1 \# s_2 \).

(b) (3 points) Is IMP still deterministic? Explain briefly.

(c) (3 points) Give example \( H, s_1, \) and \( s_2 \) such that \( H ; s_1 \) terminates, \( H ; s_2 \) terminates, \( H ; s_1 ; s_2 \) terminates, but \( H ; s_1 \# s_2 \) does not terminate.

(d) (3 points) Give example \( H, s_1, \) and \( s_2 \) such that \( H ; s_1 \) terminates, \( H ; s_2 \) terminates, \( H ; s_1 \# s_2 \) terminates, but \( H ; s_1 ; s_2 \) does not terminate.

Solution:

(a) \( s_1 \# s_2 \) executes \( s_1 \) and \( s_2 \) by alternating which substatement takes the next step, starting with \( s_1 \). In other words, it interleaves their execution with “time slices” of one execution step. After one statement reaches \( \text{skip} \), the other statement finishes executing.

(b) Yes, in fact it is still the case that for all \( H \) and \( s \), either \( s \) is \( \text{skip} \) or there is exactly one derivation of an execution step.

(c) One answer: Let \( H \) be \( \cdot \), let \( s_1 \) be \( x := 1; \text{while } y \text{ skip} \), and let \( s_2 \) be \( y := 1 \).

(d) One answer: Let \( H \) be \( \cdot \), let \( s_1 \) be \( y := 0; y := 0; y := 1 \), and let \( s_2 \) be \( \text{while } y \text{ skip} \). (Interestingly, changing \( s_1 \) to \( y := 0; y := 1 \) is still correct, but changing \( s_1 \) to \( \text{skip}; y := 1 \) is incorrect, even though \( H(y) = 0 \).
3. (38 points) This problem considers a language that is like the language for IMP expressions (not statements), but where we have pixels instead of integers. A pixel value contains three numbers between 0 and 255 (the first for red, the second for green, the third for blue, but that is not relevant much). Here is the syntax and an English description of the semantics:

\[
e ::= p \mid x \mid e + e \mid \text{lighten } e \mid \text{darken } e
\]

\[
p ::= \langle c, c, c \rangle
\]

\[
H ::= \cdot \mid H, x \mapsto p
\]

\[
\begin{align*}
(c \in \{0, 1, \ldots, 255\}) \\
(x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \})
\end{align*}
\]

- Heaps and variables work as usual, with values being pixels.
- As indicated in the syntax, all parts of pixel values must always be between 0 and 255 inclusive.
- Addition adds each component of its pixel arguments separately to produce a new pixel, with sums greater than 255 “rounded down” to 255.
- A “lighten” expression produces a pixel with each component being one more than it was in the argument, again with a max of 255 (i.e., 255 stays 255).
- A “darken” expression produces a pixel with each component being one less than it was in the argument, with a min of 0 (i.e., 0 stays 0).

(Notice higher values are lighter.)

(a) (10 pts) Give a large-step operational semantics for this language, with a judgment of the form \(H ; e \Downarrow p\).

- Hint: Use 5 rules. This is a good hint: there are other approaches that need many more rules. However, you will not lose points if you use more rules that fully define the semantics.
- Assume that you can use from mathematics (“blue math” in terms of lecture) the following operations: \(\min(x, y)\) and \(\max(x, y)\) for computing the minimum and maximum of two numbers. Also use addition and subtraction.

(b) (16 pts) Using your answer to part (a), prove this: \(H ; \text{lighten } e \Downarrow p\) if and only if \(H ; e + \langle 1, 1, 1 \rangle \Downarrow p\). Make sure to prove both directions.

(c) (8 pts) Define inference rules for a predicate \(\text{noblack}(e)\) that holds if none of the pixel constants in \(e\) are the constant \(\langle 0, 0, 0 \rangle\) and \(e\) contains no “darken” expressions.

For example, \(\text{noblack(\text{lighten } (1, 123, 0)) \text{noblack((1, 2, 3)) do hold, but noblack(\text{darken } (1, 123, 0))\) and noblack((0,0,0)+(1,1,1)) do not hold.}

You can assume that the comparison operators = and \(\ne\) can be used with pixels.

(d) (4 pts) Disprove this: If \(H ; e \Downarrow p\) and \(\text{noblack}(e)\), then \(p \neq \langle 0, 0, 0 \rangle\).

Solution:
(a)

\[
\begin{align*}
H : p & \Downarrow p & H : x & \Downarrow H(x) \\
H : e_1 & \Downarrow \langle c_1, c_2, c_3 \rangle & H : e_2 & \Downarrow \langle c_4, c_5, c_6 \rangle \\
H : e_1 + e_2 & \Downarrow \langle \min(255, c_1 + c_4), \min(255, c_2 + c_5), \min(255, c_3 + c_6) \rangle \\
H : e & \Downarrow \langle c_1, c_2, c_3 \rangle \\
H : \text{lighten } e & \Downarrow \langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle \\
H : \text{darken } e & \Downarrow \langle \max(0, c_1 - 1), \max(0, c_2 - 1), \max(0, c_3 - 1) \rangle
\end{align*}
\]
Solution:

(b) Prove the two directions separately. First assume $H : \text{lighten } e \downarrow p$. Inversion (only the lighten rule applies) ensures there is some $c_1, c_2,$ and $c_3$ such that $p$ is $\langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle$ and $H : e \downarrow \langle c_1, c_2, c_3 \rangle$. So we can use $H : e \downarrow \langle c_1, c_2, c_3 \rangle$ and the addition rule to derive:

$$H : e \downarrow \langle c_1, c_2, c_3 \rangle \quad \frac{H : \langle 1, 1, 1 \rangle \downarrow \langle 1, 1, 1 \rangle}{H : e + \langle 1, 1, 1 \rangle \downarrow p}$$

Now assume $H : e + \langle 1, 1, 1 \rangle \downarrow p$. Then inversion (only the addition rule applies) ensures there is some $c_1, c_2,$ and $c_3$ such that $p$ is $\langle \min(255, c_1 + 1), \min(255, c_2 + 1), \min(255, c_3 + 1) \rangle$ and $H : e \downarrow \langle c_1, c_2, c_3 \rangle$ (because another inversion ensures $\langle 1, 1, 1 \rangle$ can evaluate only to itself). So we can use $H : e \downarrow \langle c_1, c_2, c_3 \rangle$ and the lighten rule to derive:

$$H : e \downarrow \langle c_1, c_2, c_3 \rangle \quad \frac{H : \text{lighten } e \downarrow p}{H}$$

(c) (Note if you insist that $=$ and $\neq$ be used only on integers, then we need three rules for pixel constants.)

$$\begin{array}{cccc}
\text{noblack}(x) & p \neq \langle 0, 0, 0 \rangle & \text{noblack}(e_1) & \text{noblack}(e_2) & \text{noblack}(e) \\
\text{noblack}(p) & \text{noblack}(e_1 + e_2) & \text{noblack}(\text{lighten } e) \\
\end{array}$$

(d) This is false because of variables. Consider any heap $H$ where $H(x) = \langle 0, 0, 0 \rangle$. Then $H : x \downarrow \langle 0, 0, 0 \rangle$ but $\text{noblack}(x)$. 

8
Name: ________________________________

4. (12 points) In this problem, we use the untyped lambda calculus with small-step call-by-value left-to-right evaluation (recall that in call-by-value, function application evaluates the argument before it proceeds to the evaluation of the function’s body).

Recall this encoding of pairs:

- “mkpair” λx. λy. λz. z x y
- “fst” λp. p λx. λy. x
- “snd” λp. p λx. λy. y

We would expect a correct encoding to show “fst” (“mkpair” y z) evaluates to y. But this sequence of steps allegedly shows that “fst” (“mkpair” y z) evaluates to “fst”:

\[
\begin{align*}
&\quad \quad (\lambda p. p \lambda x. \lambda y. x)(\lambda x. \lambda y. \lambda z. z x y) y z \\
&\rightarrow (\lambda p. p \lambda x. \lambda y. x)((\lambda y. \lambda z. z y y) z) \\
&\rightarrow (\lambda p. p \lambda x. \lambda y. x)(\lambda z. z z z) \\
&\rightarrow (\lambda z. z z z) \lambda x. \lambda y. x \\
&\rightarrow (\lambda x. \lambda y. x) (\lambda x. \lambda y. x) (\lambda x. \lambda y. x) \\
&\rightarrow (\lambda x. (\lambda x. \lambda y. x)) (\lambda x. \lambda y. x) \\
&\rightarrow \lambda x. \lambda y. x
\end{align*}
\]

(a) There are errors in the reduction sequence above. Which steps are wrong and why are they wrong?
(b) Show a correct sequence of steps that produces y but is otherwise very similar to the sequence of steps shown above.

Solution:

(a) The first two steps both capture z. We should α-convert λz. z x y in order to perform these first two steps properly.

(b)
5. **(12 points)** In this problem, assume the simply-typed lambda calculus with constants (call-by-value left-to-right evaluation).

(a) **(6 pts)** For each of the following, write the set of free variables $FV$ (2 points each). Recall that in the absence of parentheses, an abstraction $\lambda x. e$ extends as far to the right as possible.

i. $x \ y \ \lambda x. \ x \ y$

ii. $\lambda x. \ x \ y \ x \ x$

iii. $\lambda x. \ (\lambda y. \ y) \ y \ \lambda x. \ x$

(b) **(6 pts)** Compute the result of the following substitutions, renaming bound variables if necessary.

i. $(\lambda y. \ \lambda z. \ z \ y) \ x = (\lambda z. \ z \ y)[x/y]$

ii. $(\lambda x. \ \lambda z. \ x \ y \ z) \ (x \ x) = (\lambda x. \ x \ y \ z)[(x \ x)/x]$

iii. $(\lambda x. \ \lambda z. \ x \ z) \ (z \ x) = (\lambda z. \ x \ z)[(z \ x)/x]$

**Solution:**

(a) i. $FV = x, y$

ii. $FV = y$

iii. $FV = y$

(b) i. $(\lambda y. \ \lambda z. \ z \ y) \ x = (\lambda z. \ z \ y)[x/y] = (\lambda z. \ z \ x)$

ii. $(\lambda x. \ \lambda z. \ x \ y \ z) \ (x \ x) = (\lambda q. \ \lambda z. \ q \ y \ z) \ (x \ x) = (\lambda z. \ q \ y \ z)[(x \ x)/q] = \lambda z. \ (x \ x) \ y \ z$

iii. $(\lambda x. \ \lambda z. \ x \ z) \ (z \ x) = (\lambda q. \ \lambda r. \ q \ r) \ (z \ x) = (\lambda r. \ q \ r)[(z \ x)/q] = \lambda r. \ (z \ x) \ r$
6. **OPTIONAL, extra credit (+8 points)** In this problem, we consider the simply-typed lambda-calculus (using small-step call-by-value left-to-right evaluation). We suppose the integer constants $c$ (of type \texttt{int}) include only positive integers (1, 2, 3, ...), i.e., we **remove negative numbers**. We add a subtraction operator ($e ::= \ldots | e - e$) and these rules:

\[
\frac{c_3 \text{ is math's subtraction of } c_2 \text{ from } c_1}{c_1 - c_2 \rightarrow c_3} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 - e_2 : \text{int}}
\]

(a) (4 points) Our small-step operational semantics needs two additional rules. Give them.

(b) (4 points) Our language is not type-safe. Demonstrate this.

**Solution:**

(a)

\[
\frac{e_1 \rightarrow e'_1}{e_1 - e_2 \rightarrow e'_1 - e_2} \quad \frac{e_2 \rightarrow e'_2}{v - e_2 \rightarrow v - e'_2}
\]

(b) Consider an expression like 3 - 4. It type-checks under the empty context (\texttt{\_}) with type \texttt{int}, but it cannot take a step because the result of the mathematical subtraction is -1, which is not in our language, so no rule applies.
Name: __________________________________________

(This page intentionally blank. Feel free to use.)