CIS 624, Fall 2016, Final Examination, 12/5/2016

Please do not turn the page until everyone is ready.

Rules:

- The exam is closed-book, notes are allowed.
- **Please stop promptly at 16:45.**
- You can rip apart the pages, but please write your name on each page.
- There are **155 points** total, distributed *unevenly* among 7 questions. A perfect score is **100 points**, any extra points over 100 will be added to your midterm score (if it was less than 100).
- Most questions have multiple parts. You will receive points for any parts you complete.
- **You are not expected to complete all 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 questions you are not confident about and if you have time, come back to them later.** Remember you only need **100 points total**.
- 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>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100 (+55)</strong></td>
<td></td>
</tr>
</tbody>
</table>
IMP Language

\[ s ::= \text{skip} \mid x := e \mid s \mid \text{if } e \text{ } s \mid \text{while } e \text{ } s \]
\[ e ::= c \mid x \mid e + e \mid e * e \]
\[ \Gamma \vdash H \]
\[ \lambda x. e \]
\[ \tau \rightarrow 1 \]
\[ v \leq x \]
\[ e \rightarrow 1 \]
\[ e \rightarrow 1 \]
\[ \text{H; } c \Downarrow c \]
\[ \text{H; } x \Downarrow H(x) \]
\[ \frac{H; c \Downarrow c}{\text{ADD}} \]
\[ \frac{H; e_1 \Downarrow c_1 \quad H; e_2 \Downarrow c_2}{H; e_1 + e_2 \Downarrow c_1 + c_2} \]
\[ \frac{H; e_1 \Downarrow c_1 \quad H; e_2 \Downarrow c_2}{H; e_1 * e_2 \Downarrow c_1 * c_2} \]
\[ \text{H; } e \Downarrow c \]
\[ \text{H; } c \Downarrow c \]
\[ \text{assign} \]
\[ \text{H; } x := e \rightarrow H, x \mapsto c \]
\[ \text{seq1} \]
\[ \frac{H; \text{skip}; s \rightarrow H; s}{H; s \rightarrow H'} \]
\[ \frac{H; s_1; s_2 \rightarrow H'; s'_1; s'_2}{H; s_1 \rightarrow H'} \]
\[ \frac{H; s_1 \rightarrow H_2; s_2}{H; \text{while } e \rightarrow H; \text{if } (s; \text{while } e) \text{ skip} \}

Simply Typed Lambda Calculus with pairs

\[ e ::= \lambda x. e \mid x \mid e \mid c \mid (e, e) \mid e.1 \mid e.2 \]
\[ v ::= \lambda x. e \mid c \mid (v, v) \]
\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \tau * \tau \]
\[ e \rightarrow e' \text{ and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2 \]
\[ \frac{(\lambda x. e) \Downarrow e[v/x]}{e_1 \rightarrow e'_1} \]
\[ \frac{e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2}{e_2 \rightarrow e'_2} \]
\[ \frac{e_1 \rightarrow e'_1}{e_2 \rightarrow e'_2} \]
\[ \frac{(v_1, v_2).1 \rightarrow v_1}{(v_1, v_2).2 \rightarrow v_2} \]
\[ \frac{(v_1, v_2) \rightarrow (v_1, e'_2)}{(v_1, e_2) \rightarrow (v_1, e'_2)} \]
\[ \frac{(v_1, e_2) \rightarrow (v_1, e'_2)}{(v_1, e_2) \rightarrow (v_1, e'_2)} \]
\[ \frac{(v_1, e_2) \rightarrow (v_1, e'_2)}{(v_1, e_2) \rightarrow (v_1, e'_2)} \]
\[ \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \]
\[ \frac{e \rightarrow e'}{e.2 \rightarrow e'.2} \]
\[ \frac{(v_1, v_2).1 \rightarrow v_1}{(v_1, v_2).2 \rightarrow v_2} \]
\[ \Gamma \vdash c : \text{int} \]
\[ \Gamma \vdash x : \Gamma(x) \]
\[ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \]
\[ \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1} \]
\[ \frac{\tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4} \]
\[ \frac{(S-\text{Arrow})}{\tau \leq \tau} \]
\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \]
\[ \frac{(S-\text{Ref})}{\tau \leq \tau} \]
\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad \tau_3 \leq \tau_4}{\tau_1 \leq \tau_4} \]
\[ \frac{(S-\text{Trans})}{\tau \leq \tau} \]

2
- 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 $\Gamma \vdash e'[\alpha/x] : \tau$.

### System F (syntax)

- $e ::= c \mid x \mid \lambda x : \tau. e \mid e e \mid \Lambda \alpha. e \mid e[\tau]$
- $\tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \tau$
- $v ::= c \mid \lambda x : \tau. e \mid \Lambda \alpha. e$

### System F: $e \rightarrow e'$ and $\Delta; \Gamma \vdash e : \tau$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{\Delta; \Gamma \vdash e : \tau \quad \Delta \vdash \tau \rightarrow \tau' \quad \Delta ; \Gamma \vdash e' : \tau'}{\Delta ; \Gamma \vdash e[\tau] \rightarrow e'[\tau]}$</td>
<td>$\lambda x : \tau. e \rightarrow \lambda x : \tau' e'$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash \tau \rightarrow \tau \quad \Delta ; \Gamma \vdash e \rightarrow e'}{\Delta ; \Gamma \vdash e[\tau] \rightarrow e'[\tau]}$</td>
<td>$e \rightarrow e'$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash \tau \rightarrow \tau \quad \Delta ; \Gamma \vdash e \rightarrow e'}{\Delta ; \Gamma \vdash e[\tau] \rightarrow e'[\tau]}$</td>
<td>$e \rightarrow e'$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Delta ; \Gamma \vdash e_2 : \tau_2}{\Delta ; \Gamma \vdash e_1 ; e_2 : \tau_1}$</td>
<td>$\mu \alpha. \tau$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash e \rightarrow \tau_1}{\Delta ; \Gamma \vdash e \rightarrow \forall \alpha. \tau_1}$</td>
<td>$\forall \alpha. \tau$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash e : \forall \alpha. \tau_1}{\Delta ; \Gamma \vdash e \rightarrow \tau_2}$</td>
<td>$\forall \alpha. \tau$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash e_2 : \tau_1}{\Delta ; \Gamma \vdash e_1 ; e_2 : \tau_1}$</td>
<td>$\mu \alpha. \tau$</td>
</tr>
</tbody>
</table>

Simple System F examples: Let $\text{id} = \Lambda \alpha. \lambda x : \alpha. x$. Then $\text{id}$ has type $\forall \alpha. \alpha \rightarrow \alpha$; $\text{id} \; \text{[int]}$ has type $\text{int} \rightarrow \text{int}$; and $\text{id} \; \text{[int * int]}$ has type $(\text{int} \ast \text{int}) \rightarrow (\text{int} \ast \text{int})$.

### Sum types, iso-recursive types

- $e ::= \ldots | A(e) \mid B(e) \mid (\text{match } e \text{ with } A.x. e \mid B.x. e) \mid \text{fold}_{\tau} e \mid \text{unfold } e$
- $\tau ::= \ldots | \tau_1 + \tau_2 \mid \mu \alpha. \tau$
- $v ::= \ldots | A(v) \mid B(v) \mid \text{fold}_{\tau} v$

match $A(v)$ with $A.x. e_1 \mid B.y. e_2 \rightarrow e_1[v/x]$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{e \rightarrow e'}{A(e) \rightarrow A(e')}$</td>
<td>$A(e) \rightarrow A(e')$</td>
</tr>
<tr>
<td>$\frac{e \rightarrow e'}{B(e) \rightarrow B(e')}$</td>
<td>$B(e) \rightarrow B(e')$</td>
</tr>
<tr>
<td>$\frac{e \rightarrow e'}{\Delta ; \Gamma \vdash e \rightarrow e'}$</td>
<td>$\text{match } e \text{ with } A.x. e_1 \mid B.y. e_2 \rightarrow \text{match } e' \text{ with } A.x. e_1 \mid B.y. e_2$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash \text{fold}_{\mu \alpha. \tau} v \rightarrow v}{\Delta ; \Gamma \vdash \text{unfold } e \rightarrow \text{unfold } e'}$</td>
<td>$\text{fold}<em>{\mu \alpha. \tau} e \rightarrow \text{fold}</em>{\mu \alpha. \tau} e'$</td>
</tr>
</tbody>
</table>

match $B(v)$ with $A.x. e_1 \mid B.y. e_2 \rightarrow e_2[v/y]$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{e \rightarrow e'}{\Delta ; \Gamma \vdash e \rightarrow e'}$</td>
<td>$\text{match } e \text{ with } A.x. e_1 \mid B.y. e_2 \rightarrow \text{match } e' \text{ with } A.x. e_1 \mid B.y. e_2$</td>
</tr>
<tr>
<td>$\frac{\Delta ; \Gamma \vdash \text{unfold } e \rightarrow \text{unfold } e'}{\Delta ; \Gamma \vdash \text{fold}_{\mu \alpha. \tau} e \rightarrow \mu \alpha. \tau}$</td>
<td>$\text{fold}_{\mu \alpha. \tau} e \rightarrow \mu \alpha. \tau$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta ; \Gamma \vdash e : \tau_1$</td>
<td>$\Delta ; \Gamma \vdash e : \tau_1$</td>
</tr>
<tr>
<td>$\Delta ; \Gamma \vdash e : \tau_2$</td>
<td>$\Delta ; \Gamma \vdash e : \tau_2$</td>
</tr>
<tr>
<td>$\Delta ; \Gamma \vdash e : \tau[(\mu \alpha. \tau)/\alpha]$</td>
<td>$\Delta ; \Gamma \vdash e : \tau[(\mu \alpha. \tau)/\alpha]$</td>
</tr>
</tbody>
</table>

$\Delta ; \Gamma \vdash \text{unfold } e : \tau[(\mu \alpha. \tau)/\alpha]$
Name:______________________________

(Intentionally blank page; use as you wish)
Name: ________________________________

1. (12 points) For each of the following OCaml definitions, does it type-check in OCaml? If so, what type does it have? If not, why not?

(a) let a = 3 in (fun f -> (fun x y -> x) (f a) (f true))

(b) let b = (fun f -> (fun x y -> x) (f 1) (f (f (f 5)))))

(c) let c = (fun x y z -> x y z) (fun p q -> p * q) 5 10

(d) let d = (fun f -> (fun x y -> y) (f 3) (f (-10)))
2. (25 points) We want to extend IMP (defined on p. 2) with case conditional of the form

```plaintext
    case e of
    c1 : s1;
    c2 : s2;
    ...;
    cn : sn
endcase
```

where `case`, `of`, and `endcase` are new keywords; `e` is an arithmetic expression, each `ci` is an integer constant, and each `si` is a statement. This program is executed by first evaluating the expression `e` to obtain a constant `c`; if the first occurrence of `c` in the list `c1,...,cn` is `ci` (duplicates are allowed in the list `c1,...,cn`), then the statement `si` is executed. If `c` does not occur in the list `c1,...,cn`, then the program immediately terminates (i.e., is equivalent to `skip`).

(a) (5 points) Give a BNF definition of the syntax of case conditionals by extending the current IMP definition of statements, `s`. It can be helpful to (optionally) use a separate metavariable `CaseList` for the list of cases between `of` and `endcase`.

```plaintext
CaseList ::= c : s | c : s; CaseList
s ::= skip | x := e | s; s | if e s s | while e s
```

(b) (10 points) Give small-step operational semantics for case statements (you should have at least two new rules).
(c) (10 points) What is the value of x at the end of the program below? Show a correct sequence of steps, specifying the small-step judgement rule(s) used in each step.

```plaintext
x := 3;
case 2 * x of
  3: x := -1;
  6: x := x + (-1);
  5: x := x + 1;
  6: x := 0
endcase
```
3. (15 points) Define a list encoding using the simply-typed lambda calculus with functions, and integers as considered in class. A non-empty list can be represented as \( \lambda s. s \ h \ t \) where \( h \) and \( t \) are the head and tail of the list.

You can (optionally) use the definition of booleans and pairs from lecture, or other helper expressions.

\[
\begin{align*}
\text{"true"} & \quad \lambda x. \lambda y. x \\
\text{"false"} & \quad \lambda x. \lambda y. y \\
\text{"mkpair"} & \quad \lambda x. \lambda y. \lambda z. z \ x \ y \\
\text{"fst"} & \quad \lambda p. p \ \lambda x. \lambda y. x \\
\text{"snd"} & \quad \lambda p. p \ \lambda x. \lambda y. y
\end{align*}
\]

Define the lambda functions for each of the following operations. You can use previously defined shortcut names (e.g., “mkpair”, “true”, etc.).

(a) Create an empty list:

(b) Create a non-empty list containing the integers 1, 2, and 3 (you can use numbers directly, no need for Church encoding).

(c) Get the last element (tail) of the list containing 1, 2, and 3 (you can use numbers directly, no need for Church encoding).
4. (26 points) This problem uses System F with pairs and extended with integer pairs and a new operation, pair subtraction. For example, with pair subtraction \((3, 4) - (1, 3)\) should result in \((2, 1)\). Note that the answers to all parts should be brief.

(a) True or false: In System F, typing rules are syntax-directed (extra credit: what does syntax-directed mean?)

(b) Define a large-step operational rule for subtraction of expressions of the form \(e_1 - e_2\) where \(e_1\) and \(e_2\) can be reduced to values that are pairs of integer constants.

\[
\begin{align*}
\text{E-Sub} \\
\hline
\quad e_1 - e_2 \Downarrow
\end{align*}
\]

(c) Give the appropriate System F typing rule for subtraction of expressions of the form \(e_1 - e_2\) where the types of \(e_1\) and \(e_2\) are pairs of ints.

\[
\begin{align*}
\text{T-Sub} \\
\hline
\Delta; \Gamma \vdash e_1 - e_2 :
\end{align*}
\]

(d) Consider a typing context where:

- There are no type variables in scope.
- \(x\) is the only term variable in scope and it has type \(\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha\).

i. What does \(\tau\) need to be for the program fragment

\[
x [\tau] (\lambda y : \text{int} \ast \text{int}. \lambda z : \text{int} \ast \text{int}. y - z) (10, 2) (5, 2)
\]

to typecheck? (Recall application — of types or terms — associates to the left.)

\(
\tau \text{ is }
\)

ii. Given your choice for \(\tau\) above, what is the type of this expression after reducing it:

\[
x [\tau] (\lambda y : \text{int} \ast \text{int}. \lambda z : \text{int} \ast \text{int}. y - z) (10, 2) (5, 2)
\]
(e) If $v$ is an arbitrary value such that

$$v \ [\tau] \ (\lambda y:\text{int} \ast \text{int}. \ \lambda z:\text{int} \ast \text{int}. \ y - z) \ (10, 2) \ (5, 2)$$

type-checks (notice $v$ is a value and no longer polymorphic), then:

i. What type does $v$ have? (Hint: it’s different from the answers to part c).

ii. What might the following expression evaluate to?

$$v \ (\lambda y:\text{int} \ast \text{int}. \ \lambda z:\text{int} \ast \text{int}. \ y - z) \ (10, 2) \ (5, 2)$$
5. (15 points)

Consider a typed λ-calculus with sum types, pair types, recursive types, unit, and int.

(a) Define a type \( t_1 \) for a binary tree of integers where:
   - Each interior node has one integer and two children.
   - Each leaf node has no data.
   - Your type definition should have the form \( \mu \alpha \cdot \cdot \cdot \).

(b) Give a type \( t_2 \) for a binary tree of integers where:
   - Each node has one integer and two optional children (meaning each child may or may not be another binary tree).
   - Your type definition should have the form \( \mu \alpha \cdot \cdot \cdot \).

(c) Explain in English how there is exactly one value of type \( t_1 \) that cannot be translated to an equivalent value of type \( t_2 \).
6. Continuation passing style in OCaml.

(a) (12 points) Assume that the \texttt{eqk}, \texttt{addk}, \texttt{timesk}, \texttt{divk} functions are defined as follows.

\begin{verbatim}
let eqk a b k = k (a = b);;
let addk a b k = k (a + b);;
let subk a b k = k (a - b);;
let times a b k = k (a * b);;
let divk a b k = k (a / b);;
\end{verbatim}

Using only the above functions, implement a CPS function \texttt{abcdk} that takes four integer arguments \texttt{a,b,c,d}, a regular continuation \texttt{k}, and an exception continuation \texttt{xk}, to compute the following integer expression: \(a \times (b + c)/d\). If \(d\) is 0, call the exception continuation \texttt{xk} and pass the offending value to it.

\begin{verbatim}
# let abcdk a b c d k xk = ...;;
val abcdk : int -> int -> int -> int -> (int -> 'a) -> (int -> 'a) -> 'a = <fun>
\end{verbatim}
(b) (20 points) Consider the direct style function that given a list of integers, returns the sum of squares of all values.

\[
\text{let rec sumsquares } l = \\
\text{match } l \text{ with} \\
\quad [] \rightarrow 0 \\
\quad h::tl \rightarrow (h^2) + (\text{sumsquares } tl)
\]

i. What is the type of \texttt{sumsquares} above?

ii. For a given call to \texttt{sumsquares} above, approximately how deep would the call-stack grow in terms of the function arguments?

iii. Write a version of \texttt{sumsquares} called \texttt{sumsquaresk} in continuation-passing style (i.e., it should take as arguments a list of integers and a continuation function:

\[
\text{let rec sumsquaresk } l \ k = \ldots, 
\]

which uses a small constant amount of stack space. You can assume that the following CPS functions are defined (you can assume only integer division is supported, e.g., \texttt{divk 5 2 (fun x->x)} returns 2).

\[
\text{open List;;} \\
\text{let eqk arg1 arg2 } k = k \text{ (arg1 == arg2);} \\
\text{let timesk arg1 arg2 } k = k \text{ (arg1 * arg2);} \\
\text{let divk arg1 arg2 } k = k \text{ (arg1 / arg2);} \\
\text{let hdk lst } k = k \text{ (hd lst);} \\
\text{let tlk lst } k = k \text{ (tl lst);} \\
\text{let addk arg1 arg2 } k = k \text{ (arg1 + arg2);}
\]

(continues on next page)
iv. What is the type of the `sumsquaresk` function you wrote in part b.iii?
7. (30 points) In this problem, we consider a call-by-value lambda-calculus with very basic support for profiling: In addition to computing a value, it computes how many times an expression of the form \( \text{count} \ e \) is evaluated. Here is the syntax and operational semantics:

\[
e ::= \lambda x. \ e \mid x \mid e \ e \mid c \mid \text{count} \ e
\]

\[
\begin{align*}
&c; e \rightarrow \ c'; e' \\
&c; (\lambda x. \ e) \ v \rightarrow c; e[v/x] \\
&c; e_1 \rightarrow \ c'; e'_1 \\
&c; e_2 \rightarrow \ c'; e'_2 \\
&c; v \ e_2 \rightarrow \ c'; v \ e'_2 \\
&c; \text{count} \ v \rightarrow c + 1; v \\
&c; \text{count} \ e \rightarrow \ c'; \text{count} \ e'
\end{align*}
\]

Given a source program \( e \), our initial state is 0; \( e \) (i.e., the global count starts at 0). A program state \( c; e \) type-checks if \( e \) type-checks (i.e., the count can be any number).

(a) (5 points) Complete the below typing rule for \( \text{count} \ e \) that is sound and not unnecessarily restrictive:

\[
\frac{}{\Gamma \vdash \text{count} \ e :}
\]

(b) (10 points) State an appropriate Preservation Lemma for this language. Prove just the case(s) directly involving \( \text{count} \ e \) expressions – i.e., only cases for which the bottom of the derivation looks like \( \Gamma \vdash e : \tau \).

(c) (10 points) State an appropriate Progress Lemma for this language. Prove just the case(s) directly involving \( \text{count} \ e \) expressions – i.e., only cases for which the bottom of the derivation looks like \( \Gamma \vdash e : \tau \).

(d) (5 points) Give an example program that terminates in our language and would terminate if we changed function application to be call-by-name but under call-by-name it would produce a different resulting count\(^1\).

\(^1\)Recall that in the call-by-value parameter passing mechanism the expression argument to a function is evaluated before the function is applied, while in call-by-name, the expression argument to a function is substituted for all the occurrences of the formal parameter and the resulting expression is then evaluated normally.
Name: ________________________________

(This page intentionally blank, feel free to use.)