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>
For your reference (next 2 pages):

**IMP Language**

\[ s ::= \text{skip} | x := e | s ; s | \text{if } e \text{ } s | \text{while } e \text{ } s \]
\[ e ::= c | x | e + e | e * e \]
\[ (c \in \{\ldots, -2, -1, 0, 1, 2, \ldots\}) \]
\[ (x \in \{x_1, x_2, \ldots, y_1, y_2, \ldots, z_1, z_2, \ldots, \}) \]

\[ H ; e \Downarrow c \]

<table>
<thead>
<tr>
<th>CONST</th>
<th>VAR</th>
<th>ADD</th>
</tr>
</thead>
</table>
| \[ H ; c \Downarrow c \] | \[ H ; x \Downarrow H(x) \] | \[ H ; e_1 \Downarrow c_1 \quad H ; e_2 \Downarrow c_2 \]
| \[ H ; e_1 \Downarrow c_1 \quad H ; e_2 \Downarrow c_2 \] | \[ H ; e_1 + e_2 \Downarrow c_1 + c_2 \] |

\[ H_1 ; s_1 \Rightarrow H_2 ; s_2 \]

\[ H ; e \Downarrow c \quad \text{assign} \]

\[ H ; x := e \Rightarrow H, x \mapsto c ; \text{skip} \quad \text{skip} ; s \Rightarrow s \quad \text{while } e \Rightarrow H, s_1 ; s_2 \Rightarrow H' ; s_1', s_2' \]

\[ H ; e \Downarrow c \quad \text{if } c > 0 \quad H ; e \Downarrow c \quad \text{if } c \leq 0 \]

\[ H ; \text{if } e \text{ } s_1 \text{ } s_2 \Rightarrow H ; s_1 \quad H ; \text{if } e \text{ } s_1 \text{ } s_2 \Rightarrow H ; s_2 \]

\[ H ; \text{while } e \text{ } s \Rightarrow H ; \text{if } (s ; \text{while } e) \text{ } \text{skip} \]

**Simply Typed Lambda Calculus with pairs**

\[ e ::= \lambda x. e | x | e \text{ } e | c | (e, e) | e.1 | e.2 \]
\[ v ::= \lambda x. e | c | (v, v) \]
\[ \tau ::= \text{int} | \tau \rightarrow \tau | \tau \times \tau \]

\[ e \rightarrow e' \quad \text{and } \Gamma \vdash e : \tau \text{ and } \tau_1 \leq \tau_2 \]

\[ (\lambda x. e) \text{ } v \rightarrow e[v/x] \quad e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2 \quad e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2 \]
\[ e_1 \rightarrow e'_1 \quad e_2 \rightarrow e'_2 \quad (e_1, e_2) \rightarrow (e'_1, e'_2) \quad (v_1, v_2) \rightarrow (v'_1, v'_2) \]
\[ e.1 \rightarrow e'.1 \quad e.2 \rightarrow e'.2 \quad (v_1, v_2).1 \rightarrow v_1 \quad (v_1, v_2).2 \rightarrow v_2 \]
\[ \Gamma \vdash c : \text{int} \quad \Gamma \vdash x : \Gamma(x) \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma, x : \tau_1 \vdash \lambda x. e : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \rightarrow \tau_1 \quad \Gamma \vdash e_1 e_2 : \tau_1 \]

\[ \tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4 \quad (\text{S-Arrow}) \quad \tau \leq \tau \quad (\text{S-Refl}) \quad \tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3 \quad (\text{S-Trans}) \]
• If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash e' : \tau$.
• If $\cdot \vdash e : \tau$, 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 $\Gamma \vdash e'[x] : \tau$.

**System F (syntax)**

$\begin{align*}
e & ::= \ c \mid x \mid \lambda \alpha. e \mid e \mid \Lambda \alpha. e \mid e[\tau] \\
\tau & ::= \ int \mid \tau \rightarrow \tau \mid \forall \alpha. \tau \\
v & ::= \ c \mid \lambda \alpha. e \mid \Lambda \alpha. e \\
\Gamma & ::= \ \cdot \mid \Gamma, x : \tau \\
\Delta & ::= \ \cdot \mid \Delta, \alpha
\end{align*}$

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

$\begin{align*}
e & \rightarrow e' \\
e_1 \rightarrow e_2 & \rightarrow e' \ e_2 \\
v \rightarrow v & \rightarrow e' \\
(e \tau) \rightarrow (e'[\tau]) & \rightarrow (\Lambda \alpha. e)[\tau] \rightarrow e[\tau/\alpha]
\end{align*}$

$\begin{align*}
\Delta; \Gamma & \vdash x : \Gamma(x) \\
\Delta; \Gamma & \vdash e : \int \\
\Delta; \Gamma & \vdash \lambda x : \tau_1. e : \tau_2 \rightarrow \tau_2 \\
\Delta; \Gamma & \vdash \lambda : \lambda \alpha. \tau_1. e : \tau_2 \\
\Delta; \Gamma & \vdash \forall \alpha. \tau_1 \rightarrow \tau_2 \\
\Delta; \Gamma & \vdash \forall \alpha. \tau_1 \rightarrow \tau_2
\end{align*}$

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 $\text{id} [\text{int} * \text{int}]$ has type $(\text{int} * \text{int}) \rightarrow (\text{int} * \text{int})$.

**Sum types, iso-recurcive types**

$\begin{align*}
e & ::= \ldots \mid A(e) \mid B(e) \mid (\text{match } e \text{ with } Ax. e \mid Bx. e) \mid \text{fold}_e \mid \text{unfold } e \\
\tau & ::= \ldots \mid \tau_1 + \tau_2 \mid \mu \alpha. \tau \\
v & ::= \ldots \mid A(v) \mid B(v) \mid \text{fold}_e \mid \text{unfold } v
\end{align*}$

$\begin{align*}
\text{match } A(v) \text{ with } Ax. e_1 \mid By. e_2 & \rightarrow e_1[v/x] \\
\text{match } B(v) \text{ with } Ax. e_1 \mid By. e_2 & \rightarrow e_2[v/y]
\end{align*}$

$\begin{align*}
e & \rightarrow e' \\
A(e) \rightarrow A(e') & \rightarrow B(e) \rightarrow B(e') \\
\text{match } e \text{ with } Ax. e_1 \mid By. e_2 & \rightarrow \text{match } e' \text{ with } Ax. e_1 \mid By. e_2 \\
\text{unfold } (\text{fold}_{\mu \alpha. \tau} v) & \rightarrow v \\
\text{fold}_e \rightarrow \text{fold}_{\mu \alpha. \tau} e' & \rightarrow \text{unfold } e \rightarrow \text{unfold } e'
\end{align*}$

$\begin{align*}
\Delta; \Gamma & \vdash e : \tau_1 \rightarrow \tau_2 \\
\Delta; \Gamma & \vdash \lambda x : \tau_1. e : \tau_2 \\
\Delta; \Gamma & \vdash \forall \alpha. \tau_1 \rightarrow \tau_2 \\
\Delta; \Gamma & \vdash \forall \alpha. \tau_1 \rightarrow \tau_2
\end{align*}$

$\begin{align*}
\Delta; \Gamma & \vdash e : \tau_1 + \tau_2 \\
\Delta; \Gamma & \vdash \text{match } e \text{ with } Ax. e_1 \mid By. e_2 : \tau \\
\Delta; \Gamma & \vdash e : \tau[\mu \alpha. \tau/\alpha] \\
\Delta; \Gamma & \vdash \text{fold}_{\mu \alpha. \tau} e : \mu \alpha. \tau \\
\Delta; \Gamma & \vdash \text{fold}_{\mu \alpha. \tau} e : \mu \alpha. \tau
\end{align*}$

$\begin{align*}
\Delta; \Gamma & \vdash \text{unfold } e : \tau[\mu \alpha. \tau/\alpha]
\end{align*}$
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)))

Solution:

(a) Does not type-check: The type-inferencer will conclude that g must be a function takes an int and a function that takes a bool, and these cannot both hold.

(b) Type-checks: (int -> int) -> int

(c) Type-checks: int

(d) Type-checks: (int -> 'a) -> 'a
2. (25 points) We want to extend IMP (defined on p. 2) with case conditional of the form

```
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`.

```
CaseList ::= c : s | c : s; CaseList
s ::= skip | x := e | 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.

\[
\begin{align*}
x &:= 3; \\
\text{case } 2 \times x \text{ of} \\
3 & : x := -1; \\
6 & : x := x + (-1); \\
5 & : x := x + 1; \\
6 & : x := 0 \\
\end{align*}
\]

endcase

Solution:

(a)

\[
\begin{align*}
\text{CaseList} &::= c : s | c : s; \text{CaseList} \\
s &::= \text{skip} | x := e | s; s | \text{if } e \text{ } s \text{ } s \\
&| \text{case } e \text{ of } \text{CaseList} \text{ endcase}
\end{align*}
\]

(b)

\[
\begin{align*}
\text{case 1} & \\
&H ; e \Downarrow c_i \\
\hline
&H ; \text{case } e \text{ of } c_1 : s_1; \cdots ; c_i : s_i; \cdots ; c_n : s_n \text{ endcase} \rightarrow H ; s_i \\
\hline
\text{case 2} & \\
&H ; e \Downarrow c \quad c \notin \{c_1, \cdots, c_n\} \\
\hline
&H ; \text{case } e \text{ of } c_1 : s_1; \cdots ; c_i : s_i; \cdots ; c_n : s_n \text{ endcase} \rightarrow H ; \text{skip}
\end{align*}
\]

(c)

\[
\begin{align*}
H &= \{\}; x := 3; \text{case } 2 \times x \text{ of } 3 : x := -1; 6 : x := x - 1; 5 : x := x + 1; 6 : x := 0 \text{ endcase} \\
\rightarrow^2 &H = x \rightarrow 3; \text{case } 2 \times x \text{ of } 3 : x := -1; 6 : x := x - 1; 5 : x := x + 1; 6 : x := 0 \text{ endcase} \\
&[\text{Seq2, Assign}] \\
\rightarrow &H = x \rightarrow 3; x := x - 1 \ [\text{Case1}] \\
\rightarrow &H = x \rightarrow 3; x := 2; \text{skip} [\text{Assign}]
\end{align*}
\]
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.

- “true” = $\lambda x. \lambda y. x$
- “false” = $\lambda x. \lambda y. y$
- “mkpair” = $\lambda x. \lambda y. \lambda z. z\ x\ y$
- “fst” = $\lambda p. p\ \lambda x. \lambda y. x$
- “snd” = $\lambda p. p\ \lambda x. \lambda y. y$

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:

```
“emptylst” = “mkpair” “false” “false”
```

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

```
“mkpair” “1” (“mkpair” “2” (“mkpair” “3” “false”))
```

(c) Get the last element (tail) of the list containing 1, 2, and 3 (you can use numbers directly, no need for Church encoding).

```
“tail” = λz. “snd” (“snd” (“snd” z)) = λz.λp. p(λx. λy. y)(λp. p(λx. λy. y)z)
```

**Solution:**

(a) “emptylst” = “mkpair” “false” “false” (each node represented by a pair whose first element is head of the list, and second element is the tail; “false” as the first element of a pair designates the empty list)

(b) “mkpair” “1” (“mkpair” “2” (“mkpair” “3” “false”))

(c) “tail” = $\lambda z. \ “snd” \ (“snd” \ (“snd” \ z)) = \lambda z.\lambda p.\ p(\lambda x. \lambda y. y)(\lambda p.\ p(\lambda x. \lambda y. y)z)$

8
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.

\[
\text{E-Sub} \quad \frac{}{e_1 - e_2 \Downarrow}
\]

(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.

\[
\text{T-Sub} \quad \frac{}{\Delta; \Gamma \vdash e_1 - e_2 :}
\]

(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

\[
\begin{align*}
\text{x \ [\tau]} \ (\lambda y : \text{int} \times \text{int}. \ \lambda z : \text{int} \times \text{int}. \ y - z) \ (10, 2) \ (5, 2)
\end{align*}
\]

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

\[
\begin{align*}
\text{x \ [\tau]} \ (\lambda y : \text{int} \times \text{int}. \ \lambda z : \text{int} \times \text{int}. \ y - z) \ (10, 2) \ (5, 2)
\end{align*}
\]
(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?
\[
v (\lambda y : \text{int} \ast \text{int}. \lambda z : \text{int} \ast \text{int}. y - z) (10, 2) (5, 2)
\]

Solution:

(a) System F typing rules are syntax-directed. This means that there is a set of rules for all possible (grammatically correct) input strings in the language, which a type checker uses to determine the types of language constructs.

(b)
\[
\begin{align*}
e_1 & \triangleright (c_1, c_2) \quad e_2 \triangleright (c_3, c_4) \\
e_1 - e_2 & \triangleright (c_1 - c_3, c_2 - c_4)
\end{align*}
\]

(c)
\[
\begin{array}{c}
\Delta; \Gamma \vdash e_1 : \text{int} \ast \text{int} \\
\Delta; \Gamma \vdash e_2 : \text{int} \ast \text{int}
\end{array} \quad \Delta; \Gamma \vdash e_1 - e_2 : \text{int} \ast \text{int}
\]

(d) i. \( \tau \) must be \( \text{int} \ast \text{int} \rightarrow \text{int} \ast \text{int} \rightarrow \text{int} \ast \text{int} \\
    \text{int} \ast \text{int}
\]

ii. \( \text{int} \ast \text{int} \rightarrow \text{int} \ast \text{int} \rightarrow \text{int} \ast \text{int} \rightarrow \text{int} \ast \text{int} \rightarrow \tau_1 \) for any \( \tau_1 \).

(e) i. It could produce any value whatsoever.
5. (15 points)
Consider a typed \( \lambda \)-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 \( t_2 \).

Solution:

(a) \( \mu \alpha. \text{unit} + (\text{int} \times \alpha \times \alpha) \)
(b) \( \mu \alpha. \text{int} \times (\text{unit} + \alpha) \times (\text{unit} + \alpha) \)
(c) The empty tree can be represented with a value of type \( t_1 \) but not with \( t_2 \) because every \( t_2 \) has at least one int.
(a) (12 points) Assume that the `eqk`, `addk`, `timesk`, `divk` functions are defined as follows.

```ocaml
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);;
```

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

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

```ocaml
let rec sumsquares l =
  match l with
    [] -> 0
  | h::tl -> (h*h) + (sumsquares tl)
```

i. What is the type of `sumsquares` above?

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

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

```ocaml
let rec sumsquaresk l k = ...
```

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., `divk 5 2 (fun x->x) returns 2`).

```ocaml
open List;;
let eqk arg1 arg2 k = k (arg1 == arg2);;
let timesk arg1 arg2 k = k (arg1 * arg2);;
let divk arg1 arg2 k = k (arg1 / arg2);;
let hdsk lst k = k (hd lst);;
let tlk lst k = k (tl lst);;
let addk arg1 arg2 k = k (arg1 + arg2);;
```
iv. What is the type of the `sumsquaresk` function you wrote in part b.iii?

Solution:

(a) $\% a \times (b + c) / d$

```ml
let abcdk a b c d k xk =
  eqk d 0
    (fun ex -> if ex then xk d
      else addk b c
        (fun bc -> timesk a bc
          (fun abc -> divk abc d k))));;
```

(b) Sum the squares of values in list.

i. `int list -> int`

ii. Its depth will be proportional to the length of the list $l$.

iii. `let rec sumsquaresk l k =

```ml
  eqk l []
    (fun empty -> if empty then k 0
      else hdk l
        (fun h -> timesk h h
          (fun h2 -> tlk l
            (fun ltail -> sumsquaresk ltail
              (fun t -> addk h2 t k))))));;
```

(* To test: *)

```ml
let print_int i =
  print_string (string_of_int i); print_newline();

sumsquaresk [1;2;3] print_int;;
```

iv. `sumsquaresk` has type `int list -> (int -> 'a) -> 'a`
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:

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

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

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:

\[
\text{T-Count}
\]

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

(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 \( \Gamma \vdash e : \tau \) \( \Gamma \vdash \text{count } e : \tau \).

(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 \( \Gamma \vdash e : \tau \) \( \Gamma \vdash \text{count } e : \tau \).

(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\(^1\).

Solution:

(a)

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

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

\(^1\)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.
(b) If \( \cdot \vdash e : \tau \) and \( c; e \rightarrow c'; e' \), then \( \cdot \vdash e' : \tau \). We can prove this by induction on the derivation of \( \cdot \vdash e : \tau \). In the case we’re asked to prove, the bottom of the derivation looks like:

\[
\frac{\cdot \vdash e_0 : \tau}{\cdot \vdash \text{count} e_0 : \tau}
\]

There are two possible ways \( c; \text{count} e_0 \) can step to some \( e' \). If \( e_0 \) is a value, then \( e' = e_0 \) and the assumed derivation’s hypothesis \( \cdot \vdash e_0 : \tau \) suffices. If \( e_0 \) is not a value, then \( e' = \text{count} e'_0 \) where \( c; e_0 \rightarrow c'; e'_0 \). So using \( \cdot \vdash e_0 : \tau \) and induction, \( \cdot \vdash e'_0 : \tau \), so we can derive \( \cdot \vdash \text{count} e'_0 : \tau \).

(c) If \( \cdot \vdash e : \tau \), then \( e \) is a value or there exists an \( e' \) and \( c' \) such that \( c; e \rightarrow c; e' \). In the case we’re asked to prove the bottom of the derivation looks like:

\[
\frac{\cdot \vdash e_0 : \tau}{\cdot \vdash \text{count} e_0 : \tau}
\]

So using \( \cdot \vdash e_0 : \tau \), by induction either \( e_0 \) is a value or \( c; e_0 \rightarrow c'; e'_0 \) for some \( c' \) and \( e'_0 \). If \( e_0 \) is a value, then \( c; \text{count} e_0 \rightarrow c + 1; e_0 \). If \( c; e_0 \rightarrow c'; e'_0 \), then we can derive \( c; \text{count} e_0 \rightarrow c'; \text{count} e'_0 \).

(d) One of an infinite number of examples is \((\lambda x. 0)(\text{count} 0)\).
(This page intentionally blank, feel free to use.)
(This page intentionally blank, feel free to use.)