Midterm Exam

COSE212: Programming Languages
2023 Fall

Instructor: Jihyeok Park

October 25, 2023. 13:30-14:45

- **If you are not good at English, please write your answers in Korean.**
  (영어가 익숙하지 않은 경우, 답안을 한글로 작성해 주세요.)
- **Write answers in good handwriting.**
  If we cannot recognize your answers, you will not get any points.
  (글씨를 알아보기 힘들면 점수를 드릴 수 없습니다. 답안을 읽기 좋게 작성해주세요.)
- **Write your answers in the boxes provided.**
  (답안을 제공된 박스 안에 작성해 주세요.)
- **There are 8 pages and 9 questions.**
  (시험은 8장으로 구성되어 있으며, 총 9개의 문제가 있습니다.)
- **Syntax and Semantics of Languages are given in Appendix.**
  (언어의 문법과 의미는 부록에서 참조할 수 있습니다.)

Student ID	
Student Name	

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	10	10	10	5	15	15	5	15	15	100
Score:										

1. 10 points The following sentences explain basic concepts of programming languages. Fill in the blanks with the following terms (**2pt per blank**):

binding	call-by-need	closure	first-order	pure
bound	call-by-reference	environment	free	shadowed
call-by-name	call-by-value	first-class	mutable	shadowing

- A(n) ☐ identifier is an identifier not yet defined in the program's current scope.
- A language supporting ☐ functions allows functions to be treated as values. Such a function value is called a(n) ☐ , defined with its environment that captures the bound identifiers when the function is defined.
- The semantics of function applications depend on the evaluation strategy:
  1. In ☐, addresses of variables used as arguments are passed to the function.
  2. In ☐, the evaluation of arguments is delayed until their values are needed and memoized for future reuse.

2. 10 points Consider the following FACE expression:

```
/* FACE */
val f = { x => h ( x ) } ;
//  1     2    3   4
val g = { x => g ( x ) } ;
//  5     6    7   8
{ val y = 42 ; y + 1 } +
//    9        10
{ 3 * f ( y ) }
//    11  12
```

Fill in the blanks in the following table (**2pt per blank**):

- If the identifier is a free identifier, write **F**.
- If the identifier is a bound occurrence, write the **index** k of the corresponding binding occurrence.

Identifier Name	h	x	g	x	y	f	y
Identifier Lookup (k)	3	4	7	8	10	11	12
Binding Occurrence (k) / Free (F)	F	2	☐	☐	☐	☐	☐

For example, already filled two cases represent that 1) the identifier `h` at index 3 is a free identifier (F), and 2) the bound occurrence of `x` at index 4 corresponds to the binding occurrence of `x` at index 2.

3. Write the results of evaluating each FACE expression with the **static scoping** and **dynamic scoping**, respectively.

   - If the expression e evaluates to a value v, write the value v.
   - If the expression e does not terminate, write **"not terminate"**.
   - If the expression e throws a run-time error, write **"error"**.

```
/* FACE */
val f = x => y => x * y;
val x = 3;
f(4)(5)
```

   (a) [2 points] Static Scoping: ______

   (b) [3 points] Dynamic Scoping: ______

```
/* FACE */
val f = x => f(x);
f(42)
```

   (c) [2 points] Static Scoping: ______

   (d) [3 points] Dynamic Scoping: ______

4. [5 points] In the following FACE expression, the identifier `fac` represents a recursive function that computes the factorial of a given integer. Fill in the blank (A) with an expression that evaluates the entire expression to `720 (= 6! = 6 * 5 * 4 * 3 * 2 * 1)`.

```
/* FACE */
val mkRec = body => {
  val f = fX => fX(fX);
  f(fY => body(   (A)   ))
};
val fac = mkRec(fac => n => {
  if (n < 2) 1
  else n * fac(n + -1)
});
fac(6)
```

`(A)` = ______

5. This question extends FACE with logical operators, conjunction (`&&`), disjunction (`||`), and negation (`!`). Note that they should support **short-circuit evaluation**:

   - `true || (1(2))` should evaluate to `true` without evaluating `1(2)`
   - `false && (1(2))` should evaluate to `false` without evaluating `1(2)`.

   While the left operand of conjunction and disjunction should evaluate to a boolean value, the right operand accepts any value:

   - `1 && 2` should throw a run-time error because `1` is not a boolean value.
   - `true && 1` should evaluate to `1` even though `1` is not a boolean value.

   The following is the modified part of the abstract syntax of FACE:

   $$\text{Expressions} \quad \mathbb{E} \ni e \quad ::= \quad \dots \quad | \quad e \texttt{ \&\& } e \quad (\texttt{And}) \quad | \quad e \texttt{ || } e \quad (\texttt{Or}) \quad | \quad \texttt{!}\, e \quad (\texttt{Not})$$

   There are two different ways to define the semantics of the logical operators using **1)** syntactic sugar with **desugaring rules** or **2) big-step operational semantics**.

   (a) 5 points The following **desugaring rules** define the semantics of logical operators by treating them as syntactic sugar.

   $$\begin{array}{ll} \mathcal{D}[\![e_1 \texttt{ \&\& } e_2]\!] & = \texttt{if } (\mathcal{D}[\![e_1]\!])\ \mathcal{D}[\![e_2]\!] \texttt{ else false} \\ \mathcal{D}[\![e_1 \texttt{ || } e_2]\!] & = \texttt{if } (\mathcal{D}[\![e_1]\!]) \texttt{ true else } \mathcal{D}[\![e_2]\!] \\ \mathcal{D}[\![\texttt{! } e]\!] & = \texttt{if } (\mathcal{D}[\![e]\!]) \texttt{ false else true} \end{array}$$

   Write the result of the following desugaring using both the **original** and **above** rules:

   $\mathcal{D}[\![\texttt{val x = y + 1; x \&\& 2}]\!] =$

   (b) 10 points Define the **big-step operational semantics** of the newly added logical operators: conjunction (`&&`), disjunction (`||`), and negation (`!`).

6. This question modifies the semantics of FACE to support **lazy evaluation** but in a different way from the one we learned in class:

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\!\langle e_1 \rangle\!\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2} \qquad \texttt{Id}\ \frac{\sigma(x) = \langle\!\langle e \rangle\!\rangle \qquad \sigma \vdash e \Rightarrow v}{\sigma \vdash x \Rightarrow v}$$

with new kinds of values called **expression values** without environments:

$$\text{Values} \quad \mathbb{V} \ni v \quad ::= \quad \ldots \quad | \quad \langle\!\langle e \rangle\!\rangle \quad (\texttt{ExprV})$$

(a) 5 points While the following FACE expression throws a free identifier error in the original semantics, it should be evaluated to 42 in the modified semantics.

```
/* FACE */ (x => y => y)(z)(42)
```

Fill the blanks in the following **derivation tree** in the modified semantics of FACE.

$$\texttt{App}\ \frac{\cdots}{\varnothing \vdash (\lambda x.\lambda y.y)(z) \Rightarrow \langle \lambda y.y, \sigma_0 \rangle} \qquad \boxed{}$$

$$\texttt{App}\ \frac{}{\varnothing \vdash (\lambda x.\lambda y.y)(z)(42) \Rightarrow 42}$$

where $\sigma_0 = [x \mapsto \langle\!\langle z \rangle\!\rangle]$ and $\sigma_1 = \sigma_0[y \mapsto \langle\!\langle 42 \rangle\!\rangle]$.

(b) 10 points Write a FACE expression evaluating different number values in the original and modified semantics, and write each resulting number value in blanks.

Original: ________ / Modified: ________

7. [5 points] **True/False questions.** Answer O for True and X for False. (Each question is worth **1 points**, but you will get **-1 points** for each wrong answer.)

   1. We can apply the tail-call optimization to the following Scala function. ☐

   ```
   def sum(n: Int): Int = if (n == 0) 0 else n + sum(n - 1)
   ```

   2. A naïve reference counting cannot deal with reference cycles. ☐

   3. A mark-and-sweep garbage collection algorithm has a fragmentation problem. ☐

   4. There is no free list to maintain in a copying garbage collection algorithm. ☐

   5. A copying garbage collection algorithm can allocate a memory cell anywhere, regardless of the from-space and the to-space. ☐

8. This question extends MFAE into IMFAE with an **increment operator** (++) and **call-by-reference** evaluation strategy **only in variable definitions**.

   The following is the modified part of the concrete/abstract syntax of IMFAE:

   ```
   <expr> ::= ... | <id> ++
   ```

   Expressions $\mathbb{E} \ni e \quad ::= \quad \ldots \quad | \quad x \,\texttt{++} \quad (\texttt{Inc})$

   and the following is the modified part of the big-step operational semantics of IMFAE:

   $$\texttt{Inc}\ \frac{x \in \text{Domain}(\sigma) \qquad \sigma(x) = a \qquad M(a) = n}{\sigma, M \vdash x \,\texttt{++} \Rightarrow n, M[a \mapsto n+1]}$$

   $$\texttt{Var}_x\ \frac{y \in \text{Domain}(\sigma) \qquad \sigma[x \mapsto \sigma(y)], M \vdash e \Rightarrow v, M'}{\sigma, M \vdash \texttt{var}\ x\texttt{=}y;\ e \Rightarrow v, M'} \qquad \texttt{Var}\ \frac{\ldots \qquad \forall y \in \mathbb{X}.\ e_1 \neq y}{\sigma, M \vdash \texttt{var}\ x\texttt{=}e_1;\ e_2 \Rightarrow v_2, M_2}$$

   The following is the Scala code for the modified part of the interpreter:

```
enum Expr
  ...
  case Inc(x: String)

def lookupId(env: Env, name: String): Addr =
  env.getOrElse(name, error(s"free identifier: $name"))

def getNumber(v: Value): BigInt = v match
  case NumV(n) => n
  case _       => error(s"not a number: ${v.str}")

def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Var(x, Id(y), e) => [ (A) ]
  case Var(x, e1, e2) => ...
  ...
  case Inc(x) => [ (B) ]
```

(a) 5 points Fill in the blank (A) in the Scala code (Hint: use `lookupId`).

(A) =

(b) 5 points Fill in the blank (B) in the Scala code (Hint: use `lookupId` and `getNumber`).

(B) =

(c) 5 points Write the result of evaluating the following IMFAE expression.

```
/* IMFAE */
var f = z => z = z * 5;
var x = 1;
var y = x;
f(x); y++; y * x++
```

Result:

9. This question extends MFAE into PMFAE with **pointers** and **loops**.
   The following is the modified part of the concrete/abstract syntax of PMFAE:

```
<expr> ::= ...
        | "&" <id>
        | "*" <expr>
        | "*" <expr> "=" <expr>
        | "until0" "(" <expr> ")" <expr>
```

$$\mathbb{E} \ni e ::= \ldots \mid \texttt{\&}\ x \ (\texttt{Ref}) \mid \texttt{*}\ e \ (\texttt{Deref}) \mid \texttt{*}\ e\texttt{=}e \ (\texttt{RefAssign}) \mid \texttt{until0}\ (e)\ e \ (\texttt{UntilZero})$$

with new kinds of values called **pointer values**:

$$\text{Values} \quad \mathbb{V} \ni v \quad ::= \quad \ldots \quad | \quad a \quad (\texttt{PtrV})$$

The following is the modified part of the Scala code for PMFAE interpreter:

```
enum Expr:
  ...
  case Ref(x: String)
  case Deref(expr: Expr)
  case RefAssign(ref: Expr, expr: Expr)
  case UntilZero(cond: Expr, body: Expr)

enum Value:
  ...
  case PtrV(addr: Addr)

def getAddr(v: Value): Addr = v match
  case PtrV(addr) => addr
  case _ => error(s"not a reference: ${v.str}")

def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case Ref(name) => (PtrV(lookupId(env, name)), mem)

  case Deref(expr) =>
    val (ev, emem) = interp(expr, env, mem)
    (emem(getAddr(ev)), emem)

  case RefAssign(ref, expr) =>
    val (rv, rmem) = interp(ref, env, mem)
    val (ev, emem) = interp(expr, env, rmem)
    (ev, emem + (getAddr(rv) -> ev))

  case UntilZero(cond, body) =>
    val (cv, cmem) = interp(cond, env, mem)
    cv match
      case NumV(0) => (NumV(0), cmem)
      case NumV(_) =>
        val (_, bmem) = interp(body, env, cmem)
        interp(expr, env, bmem)
      case _ => error(s"not a number: ${cv.str}")
```

(a) [12 points] Write the inference rules for the **big-step operational semantics** of the newly added four syntactic cases (`Ref`, `Deref`, `RefAssign`, and `UntilZero`) in PMFAE according to the Scala code.

(b) [3 points] Fill in the blanks in the following PMFAE expression to make it swap the values of `a` and `b` using the `swap` function (**1pt per blank**).

```
/* PMFAE */
var swap = x => y => {     (A)     };
var a = 1; var b = 2;
swap(     (B)     )(     (C)     );
// a == 2 and b == 1
```

(A) =

(B) =

(C) =

This is the last page.
I hope that your tests went well!

Appendix

FACE – Functions and Arithmetic Conditional Expressions

Syntax

```
<expr> ::= <id> | <number> | "true" | "false"
         | "(" <expr> ")" | "{" <expr> "}"
         | <expr> + <expr> | <expr> * <expr> | <expr> < <expr>
         | "val" <id> "=" <expr> ";" <expr> |
         | <id> "=>" <expr> | <expr> "(" <expr> ")"
         | "if" "(" <expr> ")" <expr> "else" <expr>
```

$$\begin{array}{llllllll}
\text{Expressions} & \mathbb{E} \ni e ::= x & (\texttt{Id}) & \mid e + e & (\texttt{Add}) & \mid \lambda x.e & (\texttt{Fun}) \\
& \quad\quad\quad\;\; \mid n & (\texttt{Num}) & \mid e \times e & (\texttt{Mul}) & \mid e(e) & (\texttt{App}) \\
& \quad\quad\quad\;\; \mid b & (\texttt{Bool}) & \mid e < e & (\texttt{Lt}) & \mid \texttt{if}\ (e)\ e\ \texttt{else}\ e & (\texttt{If})
\end{array}$$

where

$$\begin{array}{llllll}
\text{Integers} & n \in \mathbb{Z} & (\texttt{BigInt}) & \text{Identifiers} & x, y \in \mathbb{X} & (\texttt{String}) \\
\text{Booleans} & b \in \mathbb{B} = \{\texttt{true}, \texttt{false}\} & (\texttt{Boolean}) & & &
\end{array}$$

Semantics

$$\boxed{\sigma \vdash e \Rightarrow v}$$

$$\texttt{Id}\ \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \qquad \texttt{Num}\ \frac{}{\sigma \vdash n \Rightarrow n} \qquad \texttt{Bool}\ \frac{}{\sigma \vdash b \Rightarrow b}$$

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

$$\texttt{Lt}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2} \qquad \texttt{Fun}\ \frac{}{\sigma \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle}$$

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

$$\texttt{If}_T\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 \Rightarrow v_1} \qquad \texttt{If}_F\ \frac{\sigma \vdash e_0 \Rightarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if}\ (e_0)\ e_1\ \texttt{else}\ e_2 \Rightarrow v_2}$$

where

$$\begin{array}{llllll}
\text{Values} & \mathbb{V} \ni v ::= n & (\texttt{NumV}) & \text{Environments} & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Env}) \\
& \quad\quad\quad\;\; \mid b & (\texttt{BoolV}) & & & \\
& \quad\quad\quad\;\; \mid \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) & & &
\end{array}$$

The semantics of variable definitions is defined as syntactic sugar, and other cases recursively apply the desugaring rule to sub-expressions.

$$\mathcal{D}[\![\texttt{val}\ x\texttt{=}e\texttt{;}\ e']\!] = (\lambda x.\mathcal{D}[\![e']\!])(\mathcal{D}[\![e]\!])$$

MFAE – Mutable Variables, Functions, and Arithmetic Expressions

Syntax

```
<expr> ::= <number> | "(" <expr> ")" | "{" <expr> "}"
         | <expr> "+" <expr> | <expr> "*" <expr>
         | <id> | <id> "=>" <expr> | <expr> "(" <expr> ")"
         | "var" <id> "=" <expr> ";" <expr>
         | <id> "=" <expr> | <expr> ";" <expr>
```

$$
\begin{array}{lllllllll}
\text{Expressions} & \mathbb{E} \ni e ::= n & (\texttt{Num}) & \mid x & (\texttt{Id}) & \mid \texttt{var}\ x\texttt{=}e;\ e & (\texttt{Var}) \\
& \quad \mid e + e & (\texttt{Add}) & \mid \lambda x.e & (\texttt{Fun}) & \mid x\texttt{=}e & (\texttt{Assign}) \\
& \quad \mid e \times e & (\texttt{Mul}) & \mid e(e) & (\texttt{App}) & \mid e;\ e & (\texttt{Seq})
\end{array}
$$

where

$$
\text{Integers} \quad n \in \mathbb{Z} \quad (\texttt{BigInt}) \qquad\qquad \text{Identifiers} \quad x, y \in \mathbb{X} \quad (\texttt{String})
$$

Semantics

$$\boxed{\sigma, M \vdash e \Rightarrow v, M}$$

$$\texttt{Num}\ \frac{}{\sigma, M \vdash n \Rightarrow n, M}$$

$$\texttt{Add}\ \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 + e_2 \Rightarrow n_1 + n_2, M_2}$$

$$\texttt{Mul}\ \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2, M_2}$$

$$\texttt{Id}\ \frac{x \in \mathrm{Domain}(\sigma)}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M} \qquad \texttt{Fun}\ \frac{}{\sigma, M \vdash \lambda x.e \Rightarrow \langle \lambda x.e, \sigma \rangle, M}$$

$$\texttt{App}\ \frac{\begin{array}{ll}\sigma, M \vdash e_1 \Rightarrow \langle \lambda x.e_3, \sigma' \rangle, M_1 & \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \\ a \notin \mathrm{Domain}(M_2) & \sigma'[x \mapsto a], M_2[a \mapsto v_2] \vdash e_3 \Rightarrow v_3, M_3\end{array}}{\sigma, M \vdash e_1(e_2) \Rightarrow v_3, M_3}$$

$$\texttt{Var}\ \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \qquad a \notin \mathrm{Domain}(M_1) \qquad \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \texttt{var}\ x\texttt{=}e_1;\ e_2 \Rightarrow v_2, M_2}$$

$$\texttt{Assign}\ \frac{\sigma, M \vdash e \Rightarrow v, M' \qquad x \in \mathrm{Domain}(\sigma)}{\sigma, M \vdash x\texttt{=}e \Rightarrow v, M'[\sigma(x) \mapsto v]} \qquad \texttt{Seq}\ \frac{\sigma, M \vdash e_1 \Rightarrow \_, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1;\ e_2 \Rightarrow v_2, M_2}$$

where

$$
\begin{array}{llllllll}
\text{Environments} & \sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{A} & (\texttt{Env}) & \qquad \text{Memories} & M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Mem}) \\
\text{Values} & \mathbb{V} \ni v ::= n & (\texttt{NumV}) & \qquad \text{Addresses} & a \in \mathbb{A} & (\texttt{Addr}) \\
& \quad \mid \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) & & &
\end{array}
$$