

Midterm Exam

COSE212: Programming Languages
2026 Fall

Instructor: Jihyeok Park

October 21, 2026. 18:30-21:00

- **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 10 pages and 12 questions.**
(시험은 10 장으로 총 12 문제로 구성되어 있습니다.)
- **Syntax and Semantics of Languages are given in Appendix.**
(언어의 문법과 의미는 부록에서 참조할 수 있습니다.)

Student ID	
Student Name	

[illegible]

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

axiomatic	call-by-name	copying	eager	lexical	reference counting
big-step	call-by-need	denotational	environment	mark-and-sweep	shadowing
binding	call-by-value	desugaring	first-class	operational	small-step
bound	closure	dynamic	free	pure	static

- A(n) semantics specifies the meaning of a program by the computation it induces when it is executed. Its style describes only the relation between an expression and its final value, not the individual steps of the computation.
- When an inner scope defines an identifier with the same name as one in an enclosing scope, the inner definition hides the outer one. This is called .
- Under scoping, the binding of an identifier is determined by the sequence of function calls at run time, rather than by the syntactic structure of the program.
- A garbage collector based on stores in each object the number of references to it and frees the object as soon as the number drops to zero. However, it cannot reclaim cyclic garbage.

2. 10 points [★☆☆] Consider the following FACE expression:

```

1 /* FACE */
2 val f = x => { val x = x + 1; x * 2 } * x;
3 // 0 1          2 3      4      5
4 val g = (f => {
5 // 6 7
6   val y = f => y + 3; y
7 // 8 9 10 11
8 })(x + 1); g(2)
9 // 12 13

```

Answer the following questions using **indices** (at the odd-numbered lines) of identifiers:

- (a) Write all **free variables** using their indices. (e.g., 4, 7, etc.)

- (b) Write all the pairs of **bound occurrences** and corresponding **binding occurrences** of variables in the form of $i \rightarrow j$ where i and j are the indices of the bound and binding occurrences, respectively. (e.g., $2 \rightarrow 1$, $5 \rightarrow 3$, etc.)

- (c) Write all the pairs of **shadowing variables** and corresponding **shadowed variables** in the form of $i \rightarrow j$ where i and j are the indices of the shadowing and shadowed variables, respectively. (e.g., $6 \rightarrow 2$, $3 \rightarrow 1$, etc.)

3. 5 points [☆☆☆] Consider the following **concrete syntax** of programs in a turtle graphics language:

```
// basic elements
<nonzero> ::= "1" | "2" | "3" | ... | "9"
<digit>   ::= "0" | <nonzero>
<number>  ::= "0" | <nonzero> <digit>*
// programs and commands
<prog>     ::= <cmd> (";" <cmd>)*
<cmd>      ::= "fd" <number> | "rt" <number>? | "repeat" <number> "[" <prog> "]"
```

Answer whether the following strings are valid programs according to the concrete syntax. Write O if it is valid and X if it is not valid. (Each question is worth **1 point**, but you will get **-1 point** for each wrong answer. The total score will not be negative.)

(a) fd 0; rt; fd 10

☐

(b) repeat 4 [fd 10 rt 90]

☐

(c) repeat 2 [repeat 2 [fd 5]; rt]

☐

(d) fd 007

☐

(e) repeat 3 []

☐

4. 10 points [☆☆☆] While the original semantics of FACE uses **static scoping**, we can modify the semantics to use **dynamic scoping** as follows:

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

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)(2);
(x => f(3))(7)
```

(a) 2 points Static Scoping:

(b) 3 points Dynamic Scoping:

```
/* FACE */
val f = n => if (n < 2) 1 else n * f(n + -2);
f(5)
```

(c) 2 points Static Scoping:

(d) 3 points Dynamic Scoping:

5. 10 points [★★☆] Fill in the blanks to complete the **derivation tree** of the FACE expression:

$$\text{Val} \frac{\text{Fun} \frac{}{\emptyset \vdash \lambda x. \text{true} \Rightarrow \langle \lambda x. \text{true}, \emptyset \rangle} \quad \text{If}_T \frac{\text{App} \frac{}{\sigma_0 \vdash f(1) \Rightarrow \text{true}} \quad \text{(B)} \quad \frac{}{\sigma_0 \vdash \text{if } (f(1)) \ 1 + 2 \text{ else } 3 * 5 \Rightarrow 3}}{\emptyset \vdash \text{val } f = \lambda x. \text{true}; \text{ if } (f(1)) \ 1 + 2 \text{ else } 3 * 5 \Rightarrow 3}}{\quad}$$

where $\sigma_0 = [f \mapsto \langle \lambda x. \text{true}, \emptyset \rangle]$. A blank may have **multiple** derivations or **none** (write “None”).

(A) =

(B) =

6. 10 points [★★☆] This question extends BMFAE with **booleans**, **conditionals**, and **comparison operators**:

Expressions $\mathbb{E} \ni e ::= \dots \mid b \mid \text{if } (e) \ e \text{ else } e \mid e < e \mid e == e \mid e <= e$ Values $\mathbb{V} \ni v ::= \dots \mid b$

The rules for booleans and conditionals are similar to those of FACE, and the rules for $<$ and $==$ are similar to the following rule for $<=$. The operator $||$ is **syntactic sugar** defined by the following desugaring rule:

$$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 <= e_2 \Rightarrow n_1 \leq n_2, M_2} \quad \mathcal{D}[[e_1 || e_2]] = \text{if } (\mathcal{D}[[e_1]]) \ \text{true} \ \text{else } \mathcal{D}[[e_2]]$$

- (a) 5 points Suppose that we **redefine** $<=$ as syntactic sugar with the following desugaring rule:

$$\mathcal{D}[[e_1 <= e_2]] = \mathcal{D}[(e_1 < e_2) || (e_1 == e_2)]$$

Write an expression whose **evaluation result** with the **original** inference rule is **different** from that with the **desugaring** rule, and write both results (a value, “**not terminate**”, or “**error**”).

Expression:

Original:

Desugared:

- (b) 5 points Write a **correct** desugaring rule for $<=$, which is equivalent to its inference rule.

7. 10 points [★★☆] Consider the following Scala program:

```
sealed trait Tree
case class Node(var l: Tree, v: Int, var r: Tree) extends Tree
case class Leaf(v: Int) extends Tree
var a = Node(Leaf(1), 2, Leaf(3))
var b = Node(a, 4, Leaf(5))    // (*)
a.l = Leaf(6); a.r = b; b.r = a; b = a    // (**)
```

The registers and memory layout at the point (*) are as follows:

Registers = a = 7, b = 14

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Memory =	7	2	1	9	2	3	4	1	1	2	4	1	2	5	1	7	4	12	3	8

- The **registers** store the values of **a** and **b**, and they are the **roots** of the garbage collection.
- Each memory cell stores either an integer or an address. A **Node** occupies four consecutive cells (tag 1, l, v, and r), and a **Leaf** occupies two consecutive cells (tag 2 and v). For example, **a** is a **Node** at address 7 whose l, v, and r fields are 1, 2, and 4.
- A **free** cell (not belonging to any object) contains a meaningless value. At (*), every cell not belonging to an object allocated by the program above is free. A new object is allocated at the **lowest** address with enough consecutive free cells.

Fill in the blanks with the registers and memory layout after performing the **mark-and-sweep** garbage collection at (**). Write X for each free cell (i.e., not belonging to any **reachable** object).

Registers = a = , b =

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Memory =																				

8. 5 points [★★☆] Consider the following FACE expression:

```
/* FACE */
val mkRec = x => (y => x(y(y)))(y => x(y(y)));
val sum = mkRec(sum => n => if (n < 1) 0 else sum(n + -1) + n);
sum(10)
```

Write the **result** of evaluating the expression (a value, “**not terminate**”, or “**error**”) and explain **why** by following the evaluation order of FACE.

9. 5 points [★★☆] This question extends FACE with **pairs** and **pattern matching** as in MiniFSharp:

```

<expr>      ::= ... | "(" <expr> "," <expr> ")"
               | "match" <expr> "{" ("case" <pattern> "=>" <expr> ";")+ "}"
<pattern> ::= <number> | <id> | "(" <pattern> "," <pattern> ")"

```

Expressions $\mathbb{E} \ni e ::= \dots$ (e, e) $\text{match } e \{ (\text{case } \pi \Rightarrow e;)^+ \}$ (Pair) (Match)	Patterns $\Pi \ni \pi ::= n$ (PNum) x (π, π) (PID) (PPair)
Values $\mathbb{V} \ni v ::= \dots (v, v)$ (PairV)	

The semantics of the new expressions is defined as follows:

Pair $\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash (e_1, e_2) \Rightarrow (v_1, v_2)}$	extend : $(\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}) \times \Pi \times \mathbb{V} \rightarrow (\mathbb{X} \xrightarrow{\text{fin}} \mathbb{V})$ $\text{extend}(\sigma, n, n)$ $\text{extend}(\sigma, x, v)$ $\text{extend}(\sigma, (\pi_1, \pi_2), (v_1, v_2))$ $= \sigma$ $= \sigma[x \mapsto v]$ $= \text{extend}(\text{extend}(\sigma, \pi_1, v_1), \pi_2, v_2)$
Match $\frac{\sigma \vdash e \Rightarrow v \quad i = \min\{j \mid (\sigma, \pi_j, v) \in \text{extend}\} \quad \text{extend}(\sigma, \pi_i, v) \vdash e_i \Rightarrow v_i}{\sigma \vdash \text{match } e \{ \text{case } \pi_1 \Rightarrow e_1; \dots \text{case } \pi_n \Rightarrow e_n; \} \Rightarrow v_i}$	

If no case matches, the evaluation throws a run-time error. The following Scala implementation of the **Match** case has a **different semantics** from the rule above. Here, **cs** is the list of pattern-body pairs, and **extend(env, p, v)** returns **Some(extend(σ, π, v))** if $(\sigma, \pi, v) \in \text{extend}$ and **None** otherwise:

```

case Match(e, cs) =>
  val v = interp(e, env)
  val results = cs.flatMap((p, b) =>
    extend(env, p, v).map(newEnv => interp(b, newEnv))
  )
  results match
    case Nil      => error("no matching case")
    case r :: _  => r

```

- (a) 2 points Write an expression whose evaluation results are **different** between the semantics and the implementation, and write both results (a value, “**not terminate**”, or “**error**”).

Expression:

Semantics:

Implementation:

- (b) 3 points Modify the **Match** rule so that it describes the semantics of the **implementation**.

10. 5 points [★★★] This question extends FACE with **mutually recursive functions** as in MiniFSharp:

```
<expr> ::= ... | "def" <fun> ("and" <fun>)* ";" <expr>
<fun>  ::= <id> "(" <id> ")" "=" <expr>
```

Expressions $\mathbb{E} \ni e ::= \dots \mid \text{def } f_1(x_1) = e_1 \text{ and } \dots \text{ and } f_n(x_n) = e_n; e \quad (\text{Rec})$

The semantics of the new expression is defined as follows:

$$\text{Rec} \frac{\sigma' = \sigma[f_1 \mapsto \langle \lambda x_1. e_1, \sigma' \rangle, \dots, f_n \mapsto \langle \lambda x_n. e_n, \sigma' \rangle] \quad \sigma' \vdash e \Rightarrow v}{\sigma \vdash \text{def } f_1(x_1) = e_1 \text{ and } \dots \text{ and } f_n(x_n) = e_n; e \Rightarrow v}$$

The following is its implementation, where `funs` is the list of (name, parameter, body) triples:

```
enum Expr { ...; case Rec(funs: List[(String, String, Expr)], scope: Expr) }
enum Value { ...; case CloV(param: String, body: Expr, env: () => Env) }

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Rec(funs, s) =>
    lazy val newEnv: Env = funs.foldLeft(env) {
      case (acc, (f, p, b)) => acc + (f -> CloV(p, b, () => newEnv))
    }
    interp(s, newEnv)
  case App(fun, arg) =>
    interp(fun, env) match
      case CloV(p, b, fenv) => interp(b, fenv() + (p -> interp(arg, env)))
      case v => error(s"not a function: ${v.str}")
```

- (a) 3 points Explain what goes wrong if we change the type of `env` in `CloV` to `Env`, and accordingly replace `() => newEnv` with `newEnv` in the `Rec` case and `fenv()` with `fenv` in the `App` case.

- (b) 2 points Suppose the function passed to `foldLeft` in the `Rec` case is changed as follows:

```
case (acc, (f, p, b)) =>
  lazy val newAcc: Env = acc + (f -> CloV(p, b, () => newAcc))
  newAcc
```

Write an expression whose evaluation results are **different** between the original and the changed interpreters, and write both results (a value, “**not terminate**”, or “**error**”).

Expression:

Original:

Changed:

11. 10 points [★★★] This question extends BMFAE with **reference counting**, which deallocates a box when no reference to it remains, and `RefCount(e)`, which returns the reference count of a box:

$\langle \text{expr} \rangle ::= \dots \mid \text{"RefCount" "(" } \langle \text{expr} \rangle \text{ ")"}$

Expressions $\mathbb{E} \ni e ::= \dots \mid \text{RefCount}(e) \quad (\text{RefCount})$

Memories $M \in \mathbb{A} \xrightarrow{\text{fin}} (\mathbb{V} \times \mathbb{N}) \quad (\text{Mem})$

Each memory cell stores a **value** and its **reference count**. A new box has the count 0, and the count is increased when a reference to it is stored and decreased when the reference is removed. The following is its implementation:

```
enum Expr { ...; case RefCount(box: Expr) }
type Mem = Map[Addr, (Value, BigInt)]

def malloc(mem: Mem): Addr = ... // a fresh address not in mem

def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  ...
  case NewBox(e) =>
    val (v, m) = interp(e, env, mem)
    val a = malloc(m)
    (BoxV(a), inc(m, v) + (a -> (v, 0)))

  case GetBox(b) =>
    val (BoxV(a), m) = interp(b, env, mem)
    (m(a)._1, m)

  case SetBox(b, e) =>
    val (BoxV(a), bm) = interp(b, env, mem)
    val (v, m) = interp(e, env, bm)
    (v, set(dec(inc(m, v), m(a)._1), a, v))

  case Var(x, e1, e2) =>
    val (v1, m1) = interp(e1, env, mem)
    val b = malloc(m1)
    val (v2, m2) = interp(e2, env + (x -> b), inc(m1 + (b -> (v1, 0)), v1))
    (v2, dec(m2 - b, m2(b)._1))

  case Assign(x, e) =>
    val (v, m) = interp(e, env, mem)
    val a = env(x)
    (v, set(dec(inc(m, v), m(a)._1), a, v))

  case App(f, e) =>
    val (CloV(p, b, fenv), fm) = interp(f, env, mem)
    val (v, m) = interp(e, env, fm)
    val a = malloc(m)
    val (rv, rm) = interp(b, fenv + (p -> a), inc(m + (a -> (v, 0)), v))
    (rv, dec(rm - a, rm(a)._1))

  case RefCount(e) =>
    val (BoxV(a), m) = interp(e, env, mem)
    (NumV(m(a)._2), m)
```

- (a) 4 points The auxiliary functions **inc**, **dec**, and **set** are defined as follows, where $\text{inc}(M, v) = \text{dec}(M, v) = M$ if v is not an address:

$$\begin{aligned} \text{inc}(M, a) &= M[a \mapsto (v, n + 1)] \quad \text{where } M(a) = (v, n) \\ \text{dec}(M, a) &= \text{dec}(M \setminus \{a\}, v) \quad \text{if } M(a) = (v, 1) \\ \text{dec}(M, a) &= M[a \mapsto (v, n - 1)] \quad \text{if } M(a) = (v, n) \text{ and } n > 1 \\ \text{set}(M, a, v) &= M[a \mapsto (v, n)] \quad \text{where } M(a) = (_, n) \end{aligned}$$

The rule for **RefCount** is as follows:

$$\text{RefCount} \frac{\sigma, M \vdash e \Rightarrow a, M_1 \quad M_1(a) = (v, n)}{\sigma, M \vdash \text{RefCount}(e) \Rightarrow n, M_1}$$

Write the inference rules for **Assign** and **App**.

- (b) 6 points Write the results of evaluating the following expressions.

```
/* BMFAE with reference counting */
var b = Box(1);
var f = x => RefCount(b);
f(b) * f(b) * RefCount(b)
```

Result:

```
/* BMFAE with reference counting */
var b1 = Box(1);
var b2 = Box(b1);
var c = b1;
b2 = 0;
RefCount(c)
```

Result:

12. 10 points [★★★] This question extends BMFAE with **lazy evaluation**: the declaration `var x = e1; e2` **delays** the evaluation of e_1 until x is used. For this, values are extended with an **expression value** $\langle\langle e, \sigma \rangle\rangle$:

$$\text{Values } \mathbb{V} \ni v ::= \dots \mid \langle\langle e, \sigma \rangle\rangle$$

The following rules define **call-by-name**:

$$\text{Var} \frac{a \notin \text{dom}(M) \quad \sigma[x \mapsto a], M[a \mapsto \langle\langle e_1, \sigma \rangle\rangle] \vdash e_2 \Rightarrow v, M'}{\sigma, M \vdash \text{var } x = e_1; e_2 \Rightarrow v, M'} \quad \text{Id} \frac{(\sigma(x), M) \Downarrow (v, M')}{\sigma, M \vdash x \Rightarrow v, M'}$$

where the **strict evaluation** $(a, M) \Downarrow (v, M')$ of a cell is defined as follows:

$$\text{SENum} \frac{M(a) = n}{(a, M) \Downarrow (n, M)} \quad \text{SEAddr} \frac{M(a) = a'}{(a, M) \Downarrow (a', M)}$$

$$\text{SEClO} \frac{M(a) = \langle\lambda x. e, \sigma_0\rangle}{(a, M) \Downarrow (\langle\lambda x. e, \sigma_0\rangle, M)} \quad \text{SEExpr} \frac{M(a) = \langle\langle e, \sigma_0 \rangle\rangle \quad \sigma_0, M \vdash e \Rightarrow v, M_1}{(a, M) \Downarrow (v, M_1)}$$

- (a) 3 points Modify the semantics into **call-by-need**, where the stored expression is evaluated **at most once**, by changing only the strict evaluation rule for expression values (**SEExpr**). Write the changed rule.

- (b) 2 points Write an expression whose evaluation results are **different** between call-by-name and call-by-need, and write both results (a value, “**not terminate**”, or “**error**”).

Expression:

Call-by-name:

Call-by-need:

- (c) 5 points We now modify the semantics into **call-by-name-cache**: it evaluates the stored expression **at most once between updates**. For this, it **invalidates every cache** whenever a variable or a box is updated. The expression value is **redefined** to carry a **cache** c , where $\perp$ means no cache:

$$\text{Values } \mathbb{V} \ni v ::= \dots \mid \langle\langle e, \sigma, c \rangle\rangle \quad \text{Caches } c \in \mathbb{V} \cup \{\perp\}$$

$$\text{reset}(v) = \begin{cases} \langle\langle e, \sigma, \perp \rangle\rangle & \text{if } v = \langle\langle e, \sigma, c \rangle\rangle \\ v & \text{otherwise} \end{cases} \quad \mathcal{I}(M) = \{a \mapsto \text{reset}(M(a)) \mid a \in \text{dom}(M)\}$$

The function **reset** removes the cache of an expression value and keeps any other value unchanged, and $\mathcal{I}$ applies it to **every** cell of the memory. The rules are changed as follows:

- **Var**: stores the expression **without** a cache.
- **Assign** and **SetBox**: invalidate every cache **after** the update.
- **SEExpr_{cache}**: returns the cache of an expression value **with** a cache.
- **SEExpr**: evaluates the expression of an expression value **without** a cache and caches the result.

Write the **five** rules described above (Var, Assign, SetBox, SEExpr_{cache}, and SEExpr).

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

Appendix

FACE – Functions and Arithmetic Conditional Expressions

The following is the **concrete syntax** of FACE:

```
// basic elements
<digit>      ::= "0" | "1" | "2" | ... | "9"
<number>     ::= "-"? <digit>+
<alphabet>   ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>    ::= <alphabet> | "_"
<idcont>     ::= <alphabet> | "_" | <digit>
<keyword>    ::= "true" | "false" | "val" | "if" | "else"
<id>         ::= <idstart> <idcont>* butnot <keyword>
// expressions
<expr> ::= <number> | "true" | "false" | "(" <expr> ")" | "{" <expr> "}"
        | <expr> "+" <expr> | <expr> "*" <expr> | <expr> "<" <expr>
        | <id> | "val" <id> "=" <expr> ";" <expr> | <id> "=>" <expr>
        | <expr> "(" <expr> ")" | "if" "(" <expr> ")" <expr> "else" <expr>
```

The following are the **abstract syntax** of FACE and the precedence and associativity of operators:

Expressions	$\mathbb{E} \ni e ::= n$	(Num)	$ e * e$	(Mul)	$ \lambda x. e$	(Fun)
	$ b$	(Bool)	$ e < e$	(Lt)	$ e(e)$	(App)
	$ e + e$	(Add)	$ x$	(Id)	$ \text{val } x = e; e$	(Val)
					$ \text{if } (e) e \text{ else } e$	(If)

Numbers $n \in \mathbb{Z}$ (BigInt) Booleans $b \in \mathbb{B} = \{\text{true}, \text{false}\}$ (Boolean) Identifiers $x, y, z \in \mathbb{X}$ (String)

Description	Operator	Precedence	Associativity
Multiplicative	*	3	left
Additive	+	2	
Relational	<	1	

The **big-step operational (natural) semantics** of FACE is defined as:

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

$$\begin{array}{l}
\text{Num} \frac{}{\sigma \vdash n \Rightarrow n} \quad \text{Bool} \frac{}{\sigma \vdash b \Rightarrow b} \quad \text{Add} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \\
\\
\text{Mul} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 * e_2 \Rightarrow n_1 \times n_2} \quad \text{Lt} \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2} \quad \text{Id} \frac{x \in \text{dom}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)} \\
\\
\text{Fun} \frac{}{\sigma \vdash \lambda x. e \Rightarrow \langle \lambda x. e, \sigma \rangle} \quad \text{App} \frac{\sigma \vdash e_0 \Rightarrow \langle \lambda x. e_2, \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \sigma'[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2} \\
\\
\text{Val} \frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{val } x = e_1; e_2 \Rightarrow v_2} \\
\\
\text{If}_T \frac{\sigma \vdash e_0 \Rightarrow \text{true} \quad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_1} \quad \text{If}_F \frac{\sigma \vdash e_0 \Rightarrow \text{false} \quad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \text{if } (e_0) e_1 \text{ else } e_2 \Rightarrow v_2}
\end{array}$$

where

Values	$\mathbb{V} \ni v ::= n$	(NumV)	Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$	(Env)
	$ b$	(BoolV)			
	$ \langle \lambda x. e, \sigma \rangle$	(CloV)			

BMFAE – Boxes, Mutable Variables, Functions, and Arithmetic Expressions

The following is the **concrete syntax** of BMFAE:

```
// basic elements
...
<keyword> ::= "var" | "Box"
<id>      ::= <idstart> <idcont>* butnot <keyword>
// expressions
<expr> ::= <number> | "(" <expr> ")" | "{" <expr> "}"
        | <expr> "+" <expr> | <expr> "*" <expr>
        | <id> | <id> "=" <expr> | <expr> "(" <expr> ")"
        | "Box" "(" <expr> ")" | <expr> "." "get" | <expr> "." "set" "(" <expr> ")"
        | "var" <id> "=" <expr> ";" <expr> | <id> "=" <expr> | <expr> ";" <expr>
```

The following are the **abstract syntax** of BMFAE and the precedence and associativity of operators:

Expressions	$\mathbb{E} \ni e ::= n$	(Num)	$ x$	(Id)	$ \text{Box}(e)$	(NewBox)	$ \text{var } x = e; e$	(Var)
	$ e + e$	(Add)	$ \lambda x. e$	(Fun)	$ e.\text{get}$	(GetBox)	$ x = e$	(Assign)
	$ e * e$	(Mul)	$ e(e)$	(App)	$ e.\text{set}(e)$	(SetBox)	$ e; e$	(Seq)

Numbers	$n \in \mathbb{Z}$	(BigInt)	Identifiers	$x, y, z \in \mathbb{X}$	(String)
---------	--------------------	----------	-------------	--------------------------	----------

Description	Operator	Precedence	Associativity
Multiplicative	*	3	left
Additive	+	2	
Assignment	=	1	right

The **big-step operational (natural) semantics** of BMFAE is defined as:

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

$$\begin{array}{c}
\text{Num} \frac{}{\sigma, M \vdash n \Rightarrow n, M} \quad \text{Id} \frac{x \in \text{dom}(\sigma)}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M} \quad \text{Fun} \frac{}{\sigma, M \vdash \lambda x. e \Rightarrow \langle \lambda x. e, \sigma \rangle, M} \\
\text{Add} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 + e_2 \Rightarrow n_1 + n_2, M_2} \quad \text{Mul} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 * e_2 \Rightarrow n_1 \times n_2, M_2} \\
\text{App} \frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x. e_3, \sigma' \rangle, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \quad a \notin \text{dom}(M_2) \quad \sigma'[x \mapsto a], M_2[a \mapsto v_2] \vdash e_3 \Rightarrow v_3, M_3}{\sigma, M \vdash e_1(e_2) \Rightarrow v_3, M_3} \quad \text{GetBox} \frac{\sigma, M \vdash e \Rightarrow a, M_1}{\sigma, M \vdash e.\text{get} \Rightarrow M_1(a), M_1} \\
\text{NewBox} \frac{\sigma, M \vdash e \Rightarrow v, M_1 \quad a \notin \text{dom}(M_1)}{\sigma, M \vdash \text{Box}(e) \Rightarrow a, M_1[a \mapsto v]} \quad \text{SetBox} \frac{\sigma, M \vdash e_1 \Rightarrow a, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1.\text{set}(e_2) \Rightarrow v, M_2[a \mapsto v]} \\
\text{Var} \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \quad a \notin \text{dom}(M_1) \quad \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \text{var } x = e_1; e_2 \Rightarrow v_2, M_2} \\
\text{Assign} \frac{\sigma, M \vdash e \Rightarrow v, M' \quad x \in \text{dom}(\sigma)}{\sigma, M \vdash x = e \Rightarrow v, M'[\sigma(x) \mapsto v]} \quad \text{Seq} \frac{\sigma, M \vdash e_1 \Rightarrow _, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1; e_2 \Rightarrow v_2, M_2}
\end{array}$$

where

Values	$\mathbb{V} \ni v ::= n$	(NumV)	Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{A}$	(Env)
	$ a$	(BoxV)	Memories	$M \in \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$	(Mem)
	$ \langle \lambda x. e, \sigma \rangle$	(CloV)	Addresses	$a \in \mathbb{A}$	(Addr)