Midterm Exam

COSE212: Programming Languages

2025 Fall

Instructor: Jihyeok Park

October 22, 2025. 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 11 questions.**
  (시험은 10 장으로 총 11 문제로 구성되어 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	10	11	Total
Points:	10	10	5	10	10	5	10	10	10	10	10	100
Score:												

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

address	call-by-reference	combined	eager	pure
call-by-name	call-by-value	desugaring	first-class	syntax
call-by-need	closure	dynamic	first-order	semantics

- A $\boxed{\qquad\qquad\qquad\qquad}$ is a function value that consists of a function definition along with the environment in which the function was defined.
- The $\boxed{\qquad\qquad\qquad\qquad}$ of a programming language defines the meaning of syntactic elements of the language, as opposed to its $\boxed{\qquad\qquad\qquad\qquad}$ which defines the structure of the elements.
- A $\boxed{\qquad\qquad\qquad\qquad}$ evaluation strategy delays the evaluation of function arguments until their values are used in the function body. Each use of the argument re-evaluates the expression.
- A function is said to be $\boxed{\qquad\qquad\qquad\qquad}$ if it always produces the same output for the same input and has no side effects (i.e., it does not modify any external state).

2. $\boxed{\text{10 points}}$ [★☆☆] Consider the following FACE expression:

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

$\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

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

$\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

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

$\boxed{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

3. 5 points [☆☆☆] Consider the following **concrete syntax** of expressions:

```
// basic elements
<digit>    ::= "0" | "1" | "2" | ... | "9"
<number>   ::= "-"? <digit>+
<alphabet> ::= "a" | "b" | "c" | ... | "z"
<id>       ::= <alphabet>+
// expressions
<expr>     ::= <number> | <id> | <expr> "+" <number> | <id> "=" <expr> | "(" <expr> ")"
```

Answer whether the following strings are valid expressions 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) `1 + -2`

(b) `x + (1 + 2)`

(c) `x0 = 3 + 5`

(d) `(x = y) = 7`

(e) `x = (y = z) + 3`

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

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

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;
(x => f(2)(x + 3))(5)
```

(a) 2 points Static Scoping:

(b) 3 points Dynamic Scoping:

```
/* FACE */
val f = {
  val f = x => x + 3;
  y => f(y + 2)
}; f(42)
```

(c) 2 points Static Scoping:

(d) 3 points Dynamic Scoping:

5. $\boxed{\text{10 points}}$ [★★☆] Fill in the blanks to complete the **derivation tree** of the FACE expression:

$$\texttt{Val}\ \dfrac{\texttt{Val}\ \dfrac{}{\varnothing \vdash \lambda\texttt{x.x} \Rightarrow \langle \lambda\texttt{x.x}, \varnothing\rangle} \quad \texttt{App}\ \dfrac{\boxed{\texttt{(A)}} \quad \boxed{\texttt{(B)}} \quad \boxed{\texttt{(C)}}}{\sigma_0 \vdash \texttt{f(f)(1 + 2)} \Rightarrow 3}}{\varnothing \vdash \texttt{val f = } \lambda\texttt{x.x; f(f)(1 + 2)} \Rightarrow 3}$$

where $\sigma_0 = [\texttt{f} \mapsto \langle \lambda\texttt{x.x}, \varnothing\rangle]$.

$\boxed{\texttt{(A)}} =$

$\boxed{\texttt{(B)}} =$

$\boxed{\texttt{(C)}} =$

6. $\boxed{\text{5 points}}$ [★★☆] In the following FACE expression, the identifier `sum` represents a recursive function that computes the sum from 1 to a given integer. Fill in the blank $\boxed{\texttt{(A)}}$ with an expression that evaluates the entire expression to `55 (= 1 + 2 + ... + 10)`.

```
/* FACE */
val mkRec = f => {
  (x => f(v => x(x)(v)))(      (A)      )
};
val sum = mkRec(sum => n => if (n < 1) 0 else sum(n + -1) + n);
sum(10)
```

$\boxed{\texttt{(A)}} =$

7. $\boxed{\text{10 points}}$ [★★★] This question extends FACE to support **lists** with list operations:

- `nil` is the empty list.
- e_0 `::` e_1 prepends the element e_0 to the list e_1.
- `foldr` e_0 e_1 e_2 folds the list e_0 with initial value e_1 and binary function e_2 from the right.
- e_0 `++` e_1 appends the list e_1 to the end of the list e_0.

The followings are the extended concrete and abstract syntax:

```
<expr> ::= ...
         | nil | <expr> "::" <expr>
         | "foldr" <expr> <expr> <expr> | <expr> "++" <expr>
```

$$\text{Expressions}\quad \mathbb{E} \ni e ::= \ldots \quad \begin{array}{lllll} \mid \texttt{nil} & (\texttt{Nil}) & \mid \texttt{foldr}\ e\ e\ e & (\texttt{Foldr}) \\ \mid e \mathbin{\texttt{::}} e & (\texttt{Cons}) & \mid e \mathbin{\texttt{++}} e & (\texttt{Append}) \end{array}$$

The followings are the examples and expected results of the new list operations:

```
foldr (1 :: 2 :: 3 :: 4 :: nil) 0 (x => y => x + y)
// evaluates to 10 (i.e., 1 + (2 + (3 + (4 + 0))))
```

```
foldr ((2 :: 3 :: nil) ++ (4 :: 5 :: nil)) 1 (x => y => x * y)
// evaluates to 120 (i.e., 2 * (3 * (4 * (5 * 1))))
```

We can define semantics for lists and list operations as **syntactic sugar** with the following desugaring rules. The omitted cases recursively apply the desugaring rule to sub-expressions. Please fill in the blanks to complete the desugaring rules.

$$\mathcal{D}[\![\texttt{nil}]\!] = \lambda x.\lambda y.y$$

$$\mathcal{D}[\![e_0 \mathbin{\texttt{::}} e_1]\!] = \lambda x.\lambda y.x(\mathcal{D}[\![e_0]\!])(\mathcal{D}[\![e_1]\!](x)(y))$$

where x and y are not free identifiers in e_0 and e_1

$$\mathcal{D}[\![\texttt{foldr}\ e_0\ e_1\ e_2]\!] = \boxed{}$$

$$\mathcal{D}[\![e_0 \mathbin{\texttt{++}} e_1]\!] = \boxed{}$$

8. 10 points [★☆☆] In this question, you will write the result of **copying garbage collection** algorithm.

```
case class Node(var data: Int, var next: Node)
var x = Node(5, Node(4, Node(8, Node(3, null))))
x.next.next.next = x.next
x = x.next
x.next = Node(7, x.next)
```

After executing the above Scala program, the register and memory layout are as follows:

Register = 5

From-Space =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	3	0	8	5	4	9	5	5	7	3	2	9	11	2	5

To-Space =

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- The **register** stores the value of the variable `x`.
- The **memory** layout is a sequence of memory cells indexed by integer addresses and consists of two parts: the **from-space** for allocated memory cells and the **to-space** for copying garbage collection.
- Each memory cell stores either an integer or an address. The `null` value is represented by address 0.
- Each `Node` data structure occupies two consecutive memory cells: the first cell stores the `data` field, and the second stores the `next` field.

For example, the value of the variable `x` is stored in the register, which is an address 5 that represents a `Node` data structure:

- the `data` field is an integer 4 (at address 5)
- the `next` field is an address 9 (at address 6)

The copying garbage collection algorithm:

- copies all **reachable** objects from the from-space to the to-space sequentially, starting at address 16 and traversing in a breadth-first order from the root (i.e., the value stored in the register); and
- updates the original cells in the from-space to store a **forwarding pointer** that records the new address in the to-space, along with the tag value `99`.

For example, if a `Node` data structure originally located at address 5 is copied to address 16 in the to-space, then the cells at addresses 5 and 6 in the from-space are updated as follows:

- 99 as the tag value (at address 5)
- 16 as the forwarding pointer (at address 6)

Fill in the blanks in the following table representing the updated register and memory layout after performing copying garbage collection. Note that there is no explicit deallocation step in the copying garbage collection. Instead, the entire from-space is reclaimed as free memory once all live objects have been copied to the to-space.

Register = $\boxed{}$

From-Space =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

To-Space =

16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

9. $\boxed{\text{10 points}}$ [★★☆] This question modifies the semantics of FACE to support **lazy evaluation**:

$$\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}$$

$$\texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 \times n_2}$$

$$\texttt{Lt}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 \texttt{ < } e_2 \Rightarrow n_1 < n_2}$$

$$\texttt{App}\ \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad v_0 \Downarrow \langle \lambda x.e_2, \sigma' \rangle \qquad \sigma'[x \mapsto \langle\langle e_1, \sigma \rangle\rangle] \vdash e_2 \Rightarrow v_2}{\sigma \vdash e_0(e_1) \Rightarrow v_2}$$

$$\texttt{If}_T\ \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad v_0 \Downarrow \texttt{true} \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash \texttt{if } (e_0)\ e_1 \texttt{ else } e_2 \Rightarrow v_1} \qquad \texttt{If}_F\ \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad v_0 \Downarrow \texttt{false} \qquad \sigma \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{if } (e_0)\ e_1 \texttt{ else } e_2 \Rightarrow v_2}$$

with the following extended values

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

and the **strict evaluation** for values:

$$\boxed{v \Downarrow v}$$

$$\frac{}{n \Downarrow n} \qquad \frac{}{b \Downarrow b} \qquad \frac{}{\langle \lambda x.e, \sigma \rangle \Downarrow \langle \lambda x.e, \sigma \rangle} \qquad \frac{\sigma \vdash e \Rightarrow v \qquad v \Downarrow v'}{\langle\langle e, \sigma \rangle\rangle \Downarrow v'}$$

Note that there is **no change** to the other evaluation rules (`Num`, `Bool`, `Id`, `Fun`, and `Val`).

Write the results of evaluating each expression with the above lazy evaluation semantics. If the result is a closure or an expression value, write its captured environment as well (e.g., $\langle\langle x \texttt{ + } 1, [x \mapsto 2] \rangle\rangle$). If the expression throws a run-time error, write **"error"**.

```
(x => x)(1 + 2)
```

(a) $\boxed{\text{2 points}}$ Result:

```
(x => (y => y)(x))(2 * 3)
```

(b) $\boxed{\text{3 points}}$ Result:

```
val f = x => x;
val y = 2 * 5;
(z => f(z + y))(y + 1)
```

(c) $\boxed{\text{5 points}}$ Result:

10. 10 points [★★☆] This question implements the code transformation `optimize` that takes an expression e in BMFAE and returns an optimized expression. We say `optimize` is correct if it satisfies following properties for any expression e:

- If e evaluates to a number n, then `optimize`(e) also evaluates to the same number n.
- If e evaluates to a box, then `optimize`(e) also evaluates to a box.
- If e evaluates to a function, then `optimize`(e) also evaluates to a function.
- If e does not terminate, then `optimize`(e) also does not terminate.
- If e throws a run-time error, then `optimize`(e) also throws a run-time error.

The basic structure of the code transformation is given as follows and each optimization adds a new case to the pattern matching.

```
def optimize(expr: Expr): Expr = expr match
  // optimization cases will be added here
  case Num(number)           => Num(number)
  case Add(left, right)      => Add(optimize(left), optimize(right))
  case Mul(left, right)      => Mul(optimize(left), optimize(right))
  case Var(name, init, body) => Var(name, optimize(init), optimize(body))
  case Id(name)              => Id(name)
  case Fun(param, body)      => Fun(param, optimize(body))
  case App(fun, arg)         => App(optimize(fun), optimize(arg))
  case NewBox(content)       => NewBox(optimize(content))
  case GetBox(box)           => GetBox(optimize(box))
  case SetBox(box, content)  => SetBox(optimize(box), optimize(content))
  case Assign(name, expr)    => Assign(name, optimize(expr))
  case Seq(left, right)      => Seq(optimize(left), optimize(right))
```

For each implemented case of `optimize`, 1) write the optimization result for a given expression, 2) select either **correct** or **incorrect** for the optimization, and 2) explain whether the reasoning as follows:

- an explanation of **why** the optimization is correct; or
- an expression e as a **counterexample** and its **different evaluation results** for e and `optimize`(e).

For example, consider the following optimization case:

```
def optimize(expr: Expr): Expr = expr match
  case Add(left, right) => (optimize(left), optimize(right)) match
    case (Num(n1), Num(n2)) => Num(n1 + n2)
    case (l, r)             => Add(l, r)
  ...
```

The optimization result of `x + (1 + 2 + 3)` is `x + 6`. This optimization is **correct** because adding two numbers can be computed ahead of time without changing the semantics of the original expression.

However, consider the following optimization case:

```
def optimize(expr: Expr): Expr = expr match
  case Mul(left, right) => (optimize(left), optimize(right)) match
    case (_, Num(0)) => Num(0)
    case (l, r)      => Mul(l, r)
  ...
```

The optimization result of `x * 0` is 0. This optimization is **incorrect** because if `x * 0` throws a run-time error because `x` is a free identifier, but the optimized expression 0 evaluates to the number 0.

(a) 4 points Consider the following optimization:

```
def optimize(expr: Expr): Expr = expr match
  case SetBox(box, content) => (optimize(box), optimize(content)) match
    case (NewBox(e1), e2) => Seq(e1, e2)
    case (b, c)           => SetBox(b, c)
  ...
```

The optimization result of `Box(x = 1).set(f(42))` is

and this optimization is correct / incorrect because:

(b) 6 points Consider the following optimization:

```
def optimize(expr: Expr): Expr = expr match
  case Seq(left, right) => (optimize(left), optimize(right)) match
    case (SetBox(box, content), GetBox(box2)) if box == box2 => SetBox(box, content)
    case (l, r)                                              => Seq(l, r)
  ...
```

The optimization result of `x.set(42); x.get` is

and this optimization is correct / incorrect because:

11. $\boxed{\text{10 points}}$ [★★☆] This question extends the syntax and semantics of BMFAE to support **default arguments** for functions. The followings are the modified concrete and abstract syntax of function definitions and function applications in BMFAE:

```
<expr> ::= ...
        // original syntax for function definitions and applications
        | <id> "=>" <expr>                        | <expr> "(" <expr> ")"
        // newly added cases for default arguments
        | "(" <id> "=" <expr> ")" "=>" <expr>"   | <expr> "(" ")"
```

$$\text{Expressions}\quad \mathbb{E} \ni e ::= \ldots \qquad \mid \lambda x.e \mid \lambda(x{=}e).e \quad (\texttt{Fun}) \qquad \mid e(e) \mid e() \quad (\texttt{App})$$

with a modified definition of closure values:

$$\text{Values}\quad \mathbb{V} \ni v ::= \ldots \qquad \mid \langle \lambda(x{=}\bot).e, \sigma\rangle \mid \langle \lambda(x{=}v).e, \sigma\rangle \quad (\texttt{CloV})$$

where $x{=}\bot$ indicates that the function parameter x has no default argument, while $x{=}v$ indicates that the function parameter x has a default argument value v in the closure.

```
enum Expr:
  ...
  case Fun(param: String, default: Option[Expr], body: Expr)
  case App(fun: Expr, arg: Option[Expr])

enum Value:
  ...
  case CloV(param: String, default: Option[Value], body: Expr, env: Env)

def interp(expr: Expr, env: Env, mem: Mem): (Value, Mem) = expr match
  // omitted cases for other expressions
  ...
  // original cases for function definitions and applications
  case Fun(param, None, body) =>
    (CloV(param, None, body, env), mem)
  case App(fun, Some(arg)) =>
    val (fv, fmem) = interp(fun, env, mem)
    fv match
      case CloV(param, _, body, fenv) =>
        val (av, amem) = interp(arg, env, fmem)
        val addr = malloc(amem)
        interp(body, fenv + (param -> addr), amem + (addr -> av))
      case _ =>
        error("not a function")
  // newly added cases for default arguments
  case Fun(param, Some(default), body) =>
    val (v, newMem) = interp(default, env, mem)
    (CloV(param, Some(v), body, env), newMem)
  case App(fun, None) =>
    val (fv, fmem) = interp(fun, env, mem)
    fv match
      case CloV(param, Some(default), body, fenv) =>
        val addr = malloc(fmem)
        interp(body, fenv + (param -> addr), fmem + (addr -> default))
      case CloV(_, None, _, _) =>
        error("missing argument for function")
      case _ =>
        error("not a function")
```

(a) 6 points Write the inference rules for the **big-step operational semantics** of two extended syntactic cases in BMFAE: 1) function definitions (i.e., `Fun`) and 2) function applications (i.e., `App`). The inference rules should follow the semantics implemented in the given Scala code.

(b) 4 points Write down evaluation results of the following expressions of extended BMFAE.

```
/* BMFAE with default arguments */
var f = (box=Box(0)) => k => box.set(box.get + k);
var x = f();
var y = f();
{ x(1); x(2); x(3) } + { y(10); y(20) }
```

Result:

```
/* BMFAE with default arguments */
var inc = box => box.set(box.get + 1);
var f = (k=0) => (box=Box(k)) => box;
var g = f();
var x = g();
var y = g();
var z = f()();
{ inc(x); inc(x) } * { inc(y); inc(y); inc(y) } * { inc(z); inc(z); inc(z); inc(z) }
```

Result:

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

Appendix

FACE – Arithmetic Expressions with Functions and Conditionals

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

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

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

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

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

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

$$
\texttt{Num}\ \frac{}{\sigma \vdash n \Rightarrow n}
\qquad
\texttt{Bool}\ \frac{}{\sigma \vdash b \Rightarrow b}
\qquad
\texttt{Add}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ + } e_2 \Rightarrow n_1 + n_2}
$$

$$
\texttt{Mul}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ * } e_2 \Rightarrow n_1 \times n_2}
\qquad
\texttt{Lt}\ \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \texttt{ < } e_2 \Rightarrow n_1 < n_2}
\qquad
\texttt{Id}\ \frac{x \in \text{Domain}(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}
$$

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

$$
\texttt{Val}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val } x \texttt{ = } e_1\texttt{; } e_2 \Rightarrow v_2}
$$

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

where

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

BMFAE – and Arithmetic Expressions with Functions, Mutable Variables, and Arrays

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

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

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:

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

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

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

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

$$\texttt{NewBox}\ \frac{\sigma, M \vdash e \Rightarrow v, M_1 \qquad a \notin \text{Domain}(M_1)}{\sigma, M \vdash \texttt{Box}(e) \Rightarrow a, M_1[a \mapsto v]} \qquad \texttt{SetBox}\ \frac{\sigma, M \vdash e_1 \Rightarrow a, M_1 \qquad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1\texttt{.set}(e_2) \Rightarrow v, M_2[a \mapsto v]}$$

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

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

where

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