Midterm Exam

COSE212: Programming Languages
2024 Fall

Instructor: Jihyeok Park

October 23, 2024. 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	5	10	15	100
Score:												

1. 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-name
call-by-need
call-by-reference
call-by-value
closure
combined
desugaring
dynamic
eager
first-class
first-order
pure
static
syntactic sugar

- To support ________ scoping, we need to capture the environment where a function is defined. A function value is a pair of the function and the captured environment, which is called a(n) ________.

- If the semantic of a syntactic element is defined as a combination of other syntactic elements, we call it ________.

- There are two different evaluation strategies to support lazy evaluation. In ________ evaluation, the evaluation of expressions is delayed until their values are used the first time and memoized for future reuse.

- A function is said to be ________ if it does not have side effects and always returns the same value for the same input.

2. 10 points Consider the following FACE expression:

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

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 $i \to j$ where i and j are the indices of the bound and binding occurrences, respectively. (e.g., $2 \to 1$, $5 \to 3$, etc.)

(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.)

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

```
// basic elements
<alphabet> ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>  ::= <alphabet> | "_"
<idcont>   ::= <alphabet> | "_" | <digit>
<keyword>  ::= "true" | "false"
<id>       ::= <idstart> <idcont>* butnot <keyword>
// expressions
<expr> ::= <expr> "&&" <expr> | <id> "=" <expr> | "true" | "false" | <id>
```

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) `x = true && y` □

(b) `(true && false) && true` □

(c) `false = x` □

(d) `false && x = y && true` □

(e) `x = y = false` □

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 y = 3;
val f = x => y => x + y;
f(4)(5)
```

(a) 2 points Static Scoping: □

(b) 3 points Dynamic Scoping: □

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

(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:

$$\texttt{App}\ \frac{\boxed{\texttt{(A)}}\qquad \boxed{\texttt{(B)}}\qquad \boxed{\texttt{(C)}}}{\varnothing \vdash (\lambda \texttt{x}.\lambda \texttt{y}.(\texttt{x + y}))(2)(3) \Rightarrow 5}$$

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

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

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

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

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

7. $\boxed{\text{10 points}}$ In this question, you will write the result of **reference counting** garbage collection algorithm.

```
case class Cons(var head: Int, var tail: Cons)
var x = Cons(1, Cons(2, null))
var y = Cons(5, x)
x = Cons(7, x)
```

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

$$\text{Environment} = [\ \ \texttt{x} \mapsto 16, \quad \texttt{y} \mapsto 17\ \ ]$$

Stack =

8	1	X	X
16	17	18	19

Heap =

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

- An *environment* is a mapping from variable names to addresses in the stack.
- A *memory* layout is a sequence of memory cells indexed by integer addresses and consists of two parts: stack and heap.
- If a memory cell is *not allocated* with a garbage value, it is marked with X (e.g., address 4 in the heap).
- A *value* stored in a memory cell is either an integer or an address. The `null` value is the address 0.
- A *data structure* (e.g., `Cons`) is sequentially stored in the heap with its reference counting value as the first element. The sequential memory cells are deallocated together in the garbage collection process.

For example, the value of the variable `y` is stored in the stack at address 17, and it points to the address 1 in the heap. The sequential memory cells from address 1 to 3 represent the following `Cons` data structure:

- reference counting value is 1 (at address 1)
- `head` is an integer 5 (at address 2)
- `tail` is an address 11 (at address 3)

Assume that the subsequent assignments were executed following the previous program.

```
y = x                    /* (A) */
x.tail = x.tail.tail     /* (B) */
```

Fill in the blanks in the following table representing the updated memory layout after executing each line of the program. For clarity, the second line is executed after the first line. (Note that deallocated memory cell should be marked with X.)

(A)

Stack =

16	17	18	19

Heap =

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

(B)

Stack =

16	17	18	19

Heap =

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

8. 10 points This question extends FACE to support **pairs** and **pattern matching** as **syntactic sugar**.

The followings are the extended concrete and abstract syntax:

```
<expr> ::= ...
         | "(" <expr> "," <expr> ")"
         | "val" "(" <id> "," <id> ")" "=" <expr> ";" <expr>
```

$$
\begin{array}{llll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & & \\
 & \mid (e, e) & & (\texttt{Pair}) \\
 & \mid \texttt{val}\ (x, x) \texttt{ = } e; e & & (\texttt{Match})
\end{array}
$$

and the desugaring rules for pairs and pattern matching are defined as:

$$
\begin{array}{lll}
\mathcal{D}[\![(e_0, e_1)]\!] & = & \lambda x.(\texttt{if}\ (x)\ \mathcal{D}[\![e_0]\!]\ \texttt{else}\ \mathcal{D}[\![e_1]\!]) \\
 & & \text{where } x \text{ is not a free identifier in } e_0 \text{ or } e_1 \\
 & & \\
\mathcal{D}[\![\texttt{val}\ (x, y) \texttt{ = } e_0;\ e_1]\!] & = & \mathcal{D}[\![\texttt{val}\ z \texttt{ = } e_0;\ \texttt{val}\ x \texttt{ = } z(\texttt{true});\ \texttt{val}\ y \texttt{ = } z(\texttt{false});\ e_1]\!] \\
 & & \text{where } z \text{ is not a free identifier in } e_1
\end{array}
$$

The omitted cases recursively apply the desugaring rule to sub-expressions.

(a) 3 points See the following FACE expression using pairs and pattern matching.

```
/* FACE + pairs + pattern matching */
val (x, y) = (1, (2, 3));
val (a, b) = y;
x + a * b
```

Desugar the expression using the above desugaring rules.

(b) 3 points Another way to support pairs and pattern matching in FACE is to directly extend the semantics with **new inference rules**:

$$\texttt{Pair}\ \frac{\sigma \vdash e_0 \Rightarrow v_0 \qquad \sigma \vdash e_1 \Rightarrow v_1}{\sigma \vdash (e_0, e_1) \Rightarrow (v_0, v_1)}$$

$$\texttt{Match}\ \frac{x_0 \neq x_1 \qquad \sigma \vdash e_0 \Rightarrow (v_0, v_1) \qquad \sigma[x_0 \mapsto v_0, x_1 \mapsto v_1] \vdash e_1 \Rightarrow v_2}{\sigma \vdash \texttt{val}\ (x_0, x_1)\ \texttt{=}\ e_0;\ e_1 \Rightarrow v_2}$$

with a new kind of values called *pair values*:

$$\text{Values}\mathbb{V} \ni v ::= \ \ldots \ \mid (v, v) \qquad (\texttt{PairV})$$

Write the evaluation result of the given expression in 8(a) using the new inference rules.

(c) 4 points Answer whether two different extensions of FACE using 1) **desugaring rules** and 2) new **inference rules** always produce the same result for all valid expressions according to the extended syntax of FACE with pairs and pattern matching. If it is always the same, explain why. If not, provide a **counterexample** and explain why the results are different.

9. 5 points Define a **desugaring rule** for the sequence of expressions in MFAE using function definitions and applications. (Every desugared expression should be evaluated to the same value as the original expression even though the memory layout is different.)

$\mathcal{D}[\![e_0;\ e_1]\!] =$

10. 10 points The original semantics of MFAE has a memory leak problem. This question modifies the semantics to **automatically deallocate** memory cells for mutable variables when going out of their scope.

For example, the following two examples show the expected behavior of the **modified** semantics compared to the **original** semantics of MFAE:

In the comments, the following information is provided:

- the `order` column represents the order of the execution of expressions.
- the `original` column represents the number of allocated memory cells in the memory when using the **original** semantics.
- `modified` represents the number of allocated memory cells in the memory when using the **modified** semantics.

```
/* MFAE */                 // order |  original   modified
                           // ------|----------------------
{                          //   #1  |     0          0
  var x = 1;               //   #2  |     1          1
  x                        //   #3  |     1          1
} + 2;                     //   #4  |     1          0
{                          //   #5  |     1          0
  var y = 3;               //   #6  |     2          1
  y                        //   #7  |     2          1
} + 4                      //   #8  |     2          0
```

In the first example, two memory cells are allocated for the mutable variables `x` and `y` when they are defined (`x` at `#2` and `y` at `#6`) in the original semantics. While these memory cells are not deallocated in the original semantics, they should be deallocated when going out of their scope (`x` at `#4` and `y` at `#8`) in the modified semantics.

```
/* MFAE */                 // order |  original   modified
                           // ------|----------------------
{                          //   #1  |     0          0
  var f = x => {
    {                      //   #3  |     2          2
      var y = x;           //   #4  |     3          3
      y                    //   #5  |     3          3
    } + 1                  //   #6  |     3          2
  };                       //   #2  |     1          1
  f(2)                     //   #7  |     3          1
} + 3                      //   #8  |     3          0
```

Similarly, in the second example, three memory cells for `f`, `x`, and `y` are allocated when they are defined (`f` at `#2`, `x` at `#3`, and `y` at `#4`) in the original semantics. They should be deallocated when going out of their scope (`y` at `#6`, `x` at `#7`, and `f` at `#8`) in the modified semantics.

The goal of this question is to modify semantics of MFAE to deallocate memory cells for mutable variables when going out of their scope. However, the modified semantics always return the exactly same value as the original semantics.

(a) 5 points Modify the semantics of MFAE to satisfy the above requirements. Write only the modified inference rules using the notation $M \setminus a$ to denote the result of deleting address a from the memory M.

(b) 5 points Assume that the following new rule is **added** in the modified semantics of MFAE to support the **call-by-reference** evaluation strategy:

$$\texttt{App}_x \ \frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x'.e_2, \sigma' \rangle, M_1 \qquad x \in \text{Domain}(\sigma) \qquad \sigma'[x' \mapsto \sigma(x)], M_1 \vdash e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash e_1(x) \Rightarrow v_2, M_2 \setminus \sigma(x)}$$

Answer whether it has a problem. If it has a problem, provide a MFAE expression as a **counterexample** and describe the problem. Otherwise, explain why it does not have a problem.

11. $\boxed{\text{15 points}}$ This question extends FACE into LFACE with a **lazy list** data structure and its operations. The following is the extended part of the concrete/abstract syntax of LFACE:

```
<expr> ::= ...
         | "LazyNil" | <expr> "#::" <expr>
         | <expr> "." "isEmpty" | <expr> "." "head" | <expr> "." "tail"
```

$$
\begin{array}{lllllll}
\text{Expressions} & \mathbb{E} \ni e ::= \ldots & & & \mid e.\texttt{isEmpty} & (\texttt{IsEmpty}) \\
 & & \mid \texttt{LazyNil} & (\texttt{LazyNil}) & \mid e.\texttt{head} & (\texttt{Head}) \\
 & & \mid e \; \texttt{\#::} \; e & (\texttt{LazyCons}) & \mid e.\texttt{tail} & (\texttt{Tail})
\end{array}
$$

with three new kinds of values:

$$
\text{Values} \quad \mathbb{V} \ni v ::= \ldots \qquad \mid \langle\langle e, \sigma \rangle\rangle \quad (\texttt{ExprV}) \qquad \mid \texttt{LazyNilV} \quad (\texttt{LazyNilV}) \qquad \mid v \; \texttt{\#::} \; v \quad (\texttt{LazyConsV})
$$

The following Scala code snippet is the only modified or added part of the interpreter for LFACE compared to the original interpreter for FACE:

```
enum Expr:
  ...
  case LazyNil
  case LazyCons(head: Expr, tail: Expr)
  case IsEmpty(list: Expr)
  case Head(list: Expr)
  case Tail(list: Expr)

enum Value:
  ...
  case ExprV(expr: Expr, env: () => Env)
  case LazyNilV
  case LazyConsV(head: Value, tail: Value)

def interp(expr: Expr, env: Env): Value = expr match
  ...
  case Id(x) => strict(env.getOrElse(x, error(s"free identifier: $x")))
  case Val(x, i, b) =>
    lazy val newEnv: Env = env + (x -> ExprV(i, () => newEnv))
    interp(b, newEnv)
  case LazyNil        => LazyNilV
  case LazyCons(h, t) => LazyConsV(interp(h, env), ExprV(t, () => env))
  case IsEmpty(l) => interp(l, env) match
    case LazyNilV        => BoolV(true)
    case LazyConsV(_, _) => BoolV(false)
    case v               => error(s"not a list: ${v.str}")
  case Head(l) => interp(l, env) match
    case LazyNilV        => error(s"empty list")
    case LazyConsV(h, _) => h
    case v               => error(s"not a list: ${v.str}")
  case Tail(l) => interp(l, env) match
    case LazyNilV        => error(s"empty list")
    case LazyConsV(_, t) => strict(t)
    case v               => error(s"not a list: ${v.str}")

def strict(v: Value): Value = v match
  case ExprV(e, env) => strict(interp(e, env()))
  case _             => v
```

(a) 8 points Write the inference rules for the **big-step operational semantics** of the modified or newly added seven syntactic cases (`Id`, `Val`, `LazyNil`, `LazyCons`, `IsEmpty`, `Head`, and `Tail`) in LFACE according to the given Scala code. (You can use $v_0 \Downarrow v_1$ to denote the strict evaluation of a value v_0 to v_1.)

(b) 7 points Fill in the blanks in the following LFACE expression to make the variable `list` a lazy list that consists of **all the non-negative integers** starting from 0 and make the evaluation result 3.

```
/* LFACE */
val map = x => f => if (x.isEmpty) LazyNil else (f(x.head) #:: (A) );
val list = 0 #:: (B) ;
list.tail.tail.tail.head
```

(A) =

(B) =

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:

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

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

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

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

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

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

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

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

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

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

where

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

MFAE – and Arithmetic Expressions with Functions and Mutable Variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where

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