

MAGNET – Mutable Arithmetic Expressions with Generators and Exceptions

1 INTRODUCTION

MAGNET is a toy language for the [COSE212](#) course at Korea University. MAGNET stands for the **M**utable **A**rithmetic **E**xpressions with **G**enerators and **E**xceptions, and it supports the following features:

- **undefined value** (undefined):
- **number values** (0, 1, -1, 2, -2, 3, -3, ...)
- **boolean values** (true and false)
- **arithmetic operators**: negation (-), addition (+), subtraction (-), multiplication (*), division (/), and modulo (%)
- **logical operators**: conjunction (&&), disjunction (||), and negation (!)
- **comparison operators**: equality (== and !=) and relational (<, >, <=, and >=)
- **mutable variable definitions** (var) and **identifier lookup** (x)
- **variable assignment** (=) and **sequences** (;)
- **augmented assignment** (+=, -=, *=, /=, and %=) and **increment/decrement** (++ and --)
- **conditionals** (if-else), **while loops** (while), **break** (break), and **continue** (continue)
- **first-class functions** (=> or function)
- **function applications** and **return** (return)
- **try-catch** (try-catch) and **throw** (throw)
- **generators** (=>* or function*) and **yield** (yield)
- **iterator next** (_.next) and **iterator result accessors** (_.value and _.done)
- **for-of loops** (for-of)

This document is the specification of MAGNET. First, Section 2 describes the concrete syntax, and Section 3 describes the abstract syntax with the desugaring rules. Then, Section 4 describes the small-step operational (reduction) semantics of MAGNET.

2 CONCRETE SYNTAX

The concrete syntax of MAGNET is written in a variant of the extended Backus–Naur form (EBNF). The notation `<nt>` denotes a nonterminal, and `"t"` denotes a terminal. We use `?` to denote an optional element and `+` (or `*`) to denote one or more (or zero or more) repetitions of the preceding element. We use **butnot** to denote a set difference to exclude some strings from a producible set of strings. We omit some obvious terminals using the ellipsis (`...`) notation.

```
// basic elements
<digit>      ::= "0" | "1" | "2" | ... | "9"
<number>     ::= "-"? <digit>+
<alphabet>   ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>    ::= <alphabet> | "_"
<idcont>     ::= <alphabet> | "_" | <digit>
<keyword>    ::= "break" | "catch" | "continue" | "else" | "false"
              | "for" | "function" | "if" | "of" | "return" | "throw"
              | "true" | "try" | "undefined" | "var" | "while" | "yield"
<id>         ::= <idstart> <idcont>* butnot <keyword>
```

```

// expressions
<expr> ::= "undefined" | <number> | "true" | "false"
        // unary and binary operators
        | <uop> <expr> | <expr> <bop> <expr>
        // parentheses
        | "(" <expr> ")" | "{" <expr> "}"
        // mutable variable definitions
        "var" <id> "=" <expr> ";" <expr> | <id>
        // variable (augmented) assignment and sequence
        | <id> <aop> <expr> | <expr> ";" <expr>
        // increment and decrement
        | "++" <id> | "--" <id> | <id> "++" | <id> "--"
        // conditionals and loops
        | "if" "(" <expr> ")" <expr> "else" <expr>
        | "while" "(" <expr> ")" <expr>
        // first-class functions
        | <params> "=>" <expr> | "function" <params> "{" <expr> "}" <expr>
        // function applications and returns
        | <expr> "(" <expr> ")" | "return" <expr>
        // try-catch and throw
        | "try" <expr> "catch" "(" <id> ")" <expr> | "throw" <expr>
        // generators and yields
        | <params> "=>" "*" <expr>
        | "function" "*" <params> "{" <expr> "}" <expr> | "yield" <expr>
        // iterator next and iterator result accessors
        | <expr> "." "next" "(" <expr>? ")"
        | <expr> "." "value" | <expr> "." "done"

// operators
<aop> ::= "=" | "+=" | "-=" | "*=" | "/=" | "%="
<uop> ::= "-" | "!"
<bop> ::= "+" | "-" | "*" | "/" | "%" | "&&" | "||"
        | "==" | "!=" | "<" | "<=" | ">" | ">="

// parameters
<params> ::= "(" ")" | "(" <id> ")" | "(" <id> ["", <id>]* ")"

```

The precedence and associativity of operators are defined as follows:

Description	Operator	Precedence	Associativity
Postfix Unary	++, --, _.next, _.value, _.done	1	left
Prefix Unary	-, !, ++, --	2	right
Multiplicative	*, /, %	3	left
Additive	+, -	4	
Relational	<, <=, >, >=	5	
Equality	==, !=	6	
Logical Conjunction	&&	7	
Logical Disjunction		8	
Assignment	=, +=, -=, *=, /=, %=, var	9	right
Sequence	;	10	left

3 ABSTRACT SYNTAX

The abstract syntax of MAGNET is defined as follows:

Expressions	$\mathbb{E} \ni e ::= \text{undefined}$	(EUnDef)	$\text{if } (e) \ e \text{ else } e$	(EIf)
	$ n$	(ENum)	$\text{while } (e) \ e$	(EWhile)
	$ b$	(EBool)	break	(EBreak)
	$ e + e$	(Add)	continue	(EContinue)
	$ e * e$	(Mul)	$\lambda(x, \dots, x).e$	(EFun)
	$ e / e$	(EDiv)	$e(e, \dots, e)$	(EApp)
	$ e \% e$	(EMod)	$\text{return } e$	(EReturn)
	$ e == e$	(EEq)	$\text{try } e \text{ catch } (x) \ e$	(ETry)
	$ e < e$	(ELt)	$\text{throw } e$	(EThrow)
	$ \text{var } x = e; \ e$	(EVar)	$\lambda^*(x, \dots, x).e$	(EGen)
	$ x$	(EId)	$e.\text{next}(e^?)$	(EIterNext)
	$ x = e$	(EAssign)	$\text{yield } e$	(EYield)
	$ e; \ e$	(ESeq)	$e.\text{value}$	(EValueField)
			$e.\text{done}$	(EDoneField)

where $\left\{ \begin{array}{ll} \text{Identifier} & x \in \mathbb{X} \quad (\text{String}) \\ \text{Number} & n \in \mathbb{Z} \quad (\text{BigInt}) \end{array} \right.$ Boolean $\mathbb{B} \ni b ::= \text{true} \mid \text{false} \quad (\text{Boolean})$

The semantics of the remaining cases are defined with the following desugaring rules:

$$\begin{aligned}
\mathcal{D}[-e] &= \mathcal{D}[e] * (-1) & \mathcal{D}[x += e] &= x = x + \mathcal{D}[e] \\
\mathcal{D}[e_1 - e_2] &= \mathcal{D}[e_1] + \mathcal{D}[-e_2] & \mathcal{D}[x -= e] &= x = x - \mathcal{D}[e] \\
\mathcal{D}[e_1 \ \&\& \ e_2] &= \text{if } (\mathcal{D}[e_1]) \ \mathcal{D}[e_2] \text{ else false} & \mathcal{D}[x *= e] &= x = x * \mathcal{D}[e] \\
\mathcal{D}[e_1 \ || \ e_2] &= \text{if } (\mathcal{D}[e_1]) \ \text{true else } \mathcal{D}[e_2] & \mathcal{D}[x /= e] &= x = x / \mathcal{D}[e] \\
\mathcal{D}[\text{! } e] &= \text{if } (\mathcal{D}[e]) \ \text{false else true} & \mathcal{D}[x \% = e] &= x = x \% \mathcal{D}[e] \\
\mathcal{D}[e_1 \ != \ e_2] &= \mathcal{D}[\text{! } (e_1 == e_2)] & \mathcal{D}[++x] &= \mathcal{D}[x += 1] \\
\mathcal{D}[e_1 \ <= \ e_2] &= \text{var } \underline{x_1} = \mathcal{D}[e_1]; & \mathcal{D}[--x] &= \mathcal{D}[x -= 1] \\
&\quad \text{var } \underline{x_2} = \mathcal{D}[e_2]; & \mathcal{D}[x++] &= \text{var } \underline{x_1} = x; \ \mathcal{D}[x += 1]; \ \underline{x_1} \\
&\quad \mathcal{D}[(\underline{x_1} == \underline{x_2}) \ || \ (\underline{x_1} < \underline{x_2})] & \mathcal{D}[x--] &= \text{var } \underline{x_1} = x; \ \mathcal{D}[x -= 1]; \ \underline{x_1} \\
\mathcal{D}[e_1 > e_2] &= \mathcal{D}[\text{! } (e_1 <= e_2)] \\
\mathcal{D}[e_1 >= e_2] &= \mathcal{D}[\text{! } (e_1 < e_2)]
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}[\text{function } x \ (x_1, \dots, x_n) \{ e_1 \} e_2] &= \text{var } x = (x_1, \dots, x_n) \Rightarrow e_1; e_2 \\
\mathcal{D}[\text{function}^* x \ (x_1, \dots, x_n) \{ e_1 \} e_2] &= \text{var } x = (x_1, \dots, x_n) \Rightarrow^* e_1; e_2
\end{aligned}$$

$$\mathcal{D}[\text{for } (x \text{ of } e_1) \ e_2] = \mathcal{D} \left[\begin{array}{l} \text{var } \underline{x_1} = e_1; \\ \text{var } \underline{x_2} = \underline{x_1}.\text{next}(); \\ \text{while } (\text{! } \underline{x_2}.\text{done}) \{ \\ \quad \text{var } x = \underline{x_2}.\text{value}; \\ \quad e_2; \ \underline{x_2} = \underline{x_1}.\text{next}() \\ \} \end{array} \right]$$

where $\underline{x_k}$ denotes a fresh temporary variable. All the omitted cases recursively apply the desugaring rule to their sub-expressions. For example, $\mathcal{D}[e_1 + e_2] = \mathcal{D}[e_1] + \mathcal{D}[e_2]$.

4 SEMANTICS

We use the following notations in the semantics:

States	$\langle \kappa \parallel s \parallel H \parallel M \rangle \in \mathbb{K} \times \mathbb{S} \times \mathbb{H} \times \mathbb{M}$ (State)		
Continuations	$\kappa \in \mathbb{K}$	(Cont)	
	$\kappa ::= \square \mid i :: \kappa$		
Instructions	$i \in \mathbb{I}$	(Inst)	
	$i ::= (\sigma \vdash e)$	(IEval)	$\mid \text{jmp-if}[\psi]$ (IJmpIf)
	$\mid (+)$	(IAdd)	$\mid \text{jmp}[c]$ (IJmp)
	$\mid (*)$	(IMul)	$\mid \text{call}[n]$ (ICall)
	$\mid (/)$	(IDiv)	$\mid \text{return}$ (IReturn)
	$\mid (\%)$	(IMod)	$\mid \text{next}$ (INext)
	$\mid (==)$	(IEq)	$\mid \text{yield}$ (IYield)
	$\mid (<)$	(ILt)	$\mid \text{value}$ (IValueField)
	$\mid \text{def}[x, \dots, x][\sigma \vdash e]$	(IDef)	$\mid \text{done}$ (IDoneField)
	$\mid \text{write}[a]$	(IWrite)	$\mid \text{pop}$ (IPop)
Value Stacks	$s \in \mathbb{S}$	(Stack)	
	$s ::= \blacksquare \mid v :: s$		
Values	$v \in \mathbb{V}$	(Value)	
	$v ::= \text{undefined}$	(UndefV)	$\mid \langle \kappa \parallel s \parallel H \rangle$ (ContV)
	$\mid n$	(NumV)	$\mid \langle \lambda^*(x, \dots, x).e, \sigma \rangle$ (GenV)
	$\mid b$	(BoolV)	$\mid \text{iter}[a]$ (IterV)
	$\mid \langle \lambda(x, \dots, x).e, \sigma \rangle$	(CloV)	$\mid \{\text{value} : v, \text{done} : b\}$ (ResultV)
Control Handlers	$H \in \mathbb{H} = \mathbb{C} \xrightarrow{\text{fin}} \Psi$	(Handler)	
Control Operators	$c \in \mathbb{C}$	(Control)	
	$c ::= \text{return}$	(Return)	$\mid \text{throw}$ (Throw)
	$\mid \text{break}$	(Break)	$\mid \text{throw}$ (Finally)
	$\mid \text{continue}$	(Continue)	$\mid \text{yield}$ (Yield)
Continuation Values	$\psi, \langle \kappa \parallel s \parallel H \rangle \in \Psi = \mathbb{K} \times \mathbb{S} \times \mathbb{H}$	(KValue)	
Memories	$M \in \mathbb{M} = \mathbb{A} \xrightarrow{\text{fin}} \mathbb{V}$	(Mem)	
Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{A}$	(Env)	
Addresses	$a \in \mathbb{A}$	(Addr)	

The small-step operational (reduction) semantics of MAGNET is defined in the following form of the reduction relation (\rightarrow):

$$\langle \kappa \parallel s \parallel H \parallel M \rangle \rightarrow \langle \kappa \parallel s \parallel H \parallel M \rangle$$

4.1 Reduction Relations for IEval

EUnDef	$\langle(\sigma \vdash \text{undefined}) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle\kappa \parallel \text{undefined} :: s \parallel H \parallel M\rangle$
ENum	$\langle(\sigma \vdash n) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle\kappa \parallel n :: s \parallel H \parallel M\rangle$
EBool	$\langle(\sigma \vdash b) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle\kappa \parallel b :: s \parallel H \parallel M\rangle$
EAdd	$\langle(\sigma \vdash e_1 + e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (+) :: \kappa \parallel s \parallel H \parallel M\rangle$
EMul	$\langle(\sigma \vdash e_1 * e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (*) :: \kappa \parallel s \parallel H \parallel M\rangle$
EDiv	$\langle(\sigma \vdash e_1 / e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (/) :: \kappa \parallel s \parallel H \parallel M\rangle$
EMod	$\langle(\sigma \vdash e_1 \% e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (\%) :: \kappa \parallel s \parallel H \parallel M\rangle$
EEq	$\langle(\sigma \vdash e_1 == e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (==) :: \kappa \parallel s \parallel H \parallel M\rangle$
ELt	$\langle(\sigma \vdash e_1 < e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (<) :: \kappa \parallel s \parallel H \parallel M\rangle$
EVar	$\langle(\sigma \vdash \text{var } x = e_1; e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: \text{def}[x][\sigma \vdash e_2] :: \kappa \parallel s \parallel H \parallel M\rangle$
EId	$\langle(\sigma \vdash x) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle\kappa \parallel M(\sigma(x)) :: s \parallel H \parallel M\rangle$
EAssign	$\langle(\sigma \vdash x = e) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e) :: \text{write}(\sigma(x)) :: \kappa \parallel s \parallel H \parallel M\rangle$
ESeq	$\langle(\sigma \vdash e_1; e_2) :: \kappa \parallel s \parallel H \parallel M\rangle \rightarrow \langle(\sigma \vdash e_1) :: \text{pop} :: (\sigma \vdash e_2) :: \kappa \parallel s \parallel H \parallel M\rangle$

4.1.1 Conditionals and While Loops.

EIf	$\langle(\sigma \vdash \text{if } (e_1) e_2 \text{ else } e_3) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle(\sigma \vdash e_1) :: \text{jmp-if}[\langle(\sigma \vdash e_2) :: \kappa \parallel s \parallel H\rangle] :: (\sigma \vdash e_3) :: \kappa \parallel s \parallel H \parallel M\rangle$
EWhile	$\langle(\sigma \vdash \text{while } (e_1) e_2) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle(\sigma \vdash e_1) :: \text{jmp-if}[\psi_{\text{body}}] :: \kappa \parallel \text{undefined} :: s \parallel H \parallel M\rangle$
where	$\begin{cases} \psi_{\text{body}} &= \langle(\sigma \vdash e_2) :: \text{jmp}[\text{continue}] :: \square \parallel s \parallel H_{\text{body}}\rangle \\ H_{\text{body}} &= H[\text{continue} \mapsto \psi_{\text{continue}}, \text{break} \mapsto \psi_{\text{break}}] \\ \psi_{\text{continue}} &= \langle\text{pop} :: (\sigma \vdash \text{while } (e_1) e_2) :: \kappa \parallel s \parallel H\rangle \\ \psi_{\text{break}} &= \langle\kappa \parallel s \parallel H\rangle \end{cases}$
EBreak	$\langle(\sigma \vdash \text{break}) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle\text{jmp}[\text{break}] :: \square \parallel \text{undefined} :: s \parallel H \parallel M\rangle$
EContinue	$\langle(\sigma \vdash \text{continue}) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle\text{jmp}[\text{continue}] :: \square \parallel \text{undefined} :: s \parallel H \parallel M\rangle$

4.1.2 Functions and Return.

EFun	$\langle(\sigma \vdash \lambda(x, \dots, x).e) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle\kappa \parallel \langle\lambda(x, \dots, x).e, \sigma\rangle :: s \parallel H \parallel M\rangle$
EApp	$\langle(\sigma \vdash e(e_1, \dots, e_n)) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle(\sigma \vdash e) :: (\sigma \vdash e_1) :: \dots :: (\sigma \vdash e_n) :: \text{call}[n] :: \kappa \parallel s \parallel H \parallel M\rangle$
EReturn	$\langle(\sigma \vdash \text{return } e) :: \kappa \parallel s \parallel H \parallel M\rangle$
\rightarrow	$\langle(\sigma \vdash e) :: \text{return} :: \kappa \parallel s \parallel H \parallel M\rangle$

4.1.3 Exceptions.

$$\begin{array}{lcl}
\text{ETry} & \langle (\sigma \vdash \text{try } e_1 \text{ catch } (x) e_2) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e_1) :: \text{jmp}[\text{finally}] :: \square \parallel s \parallel H_{\text{body}} \parallel M \rangle \\
\\
& \text{where } \begin{cases} H_{\text{body}} = H[\text{throw} \mapsto \psi_{\text{throw}}, \text{finally} \mapsto \psi_{\text{finally}}] \\ \psi_{\text{throw}} = \langle \text{def}[x][\sigma \vdash e_2] :: \kappa \parallel s \parallel H \rangle \\ \psi_{\text{finally}} = \langle \kappa \parallel s \parallel H \rangle \end{cases} \\
\\
\text{EThrow} & \langle (\sigma \vdash \text{throw } e) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e) :: \text{jmp}[\text{throw}] :: \square \parallel s \parallel H \parallel M \rangle
\end{array}$$

4.1.4 Generators.

$$\begin{array}{lcl}
\text{EGen} & \langle (\sigma \vdash \lambda^*(x_1, \dots, x_n).e) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle \kappa \parallel \langle \lambda^*(x_1, \dots, x_n).e, \sigma \rangle :: s \parallel H \parallel M \rangle \\
\\
\text{EIterNext}_1 & \langle (\sigma \vdash e_1.\text{next}()) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e_1) :: (\sigma \vdash \text{undefined}) :: \text{next} :: \kappa \parallel s \parallel H \parallel M \rangle \\
\\
\text{EIterNext}_2 & \langle (\sigma \vdash e_1.\text{next}(e_2)) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: \text{next} :: \kappa \parallel s \parallel H \parallel M \rangle \\
\\
\text{EYield} & \langle (\sigma \vdash \text{yield } e) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e) :: \text{yield} :: \square \parallel \text{false} :: \psi_{\text{next}} :: s \parallel H \parallel M \rangle \\
\\
& \text{where } \psi_{\text{next}} = \langle \kappa \parallel s \parallel H \rangle \\
\\
\text{EValueField} & \langle (\sigma \vdash e.\text{value}) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e) :: \text{value} :: \kappa \parallel s \parallel H \parallel M \rangle \\
\\
\text{EDoneField} & \langle (\sigma \vdash e.\text{done}) :: \kappa \parallel s \parallel H \parallel M \rangle \\
\rightarrow & \langle (\sigma \vdash e) :: \text{done} :: \kappa \parallel s \parallel H \parallel M \rangle
\end{array}$$

4.2 Reduction Relations for Other Instructions

$$\begin{array}{lcl}
\text{IAdd} & \langle (+) :: \kappa \parallel n_2 :: n_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel (n_1 + n_2) :: s \parallel H \parallel M \rangle \\
\text{IMul} & \langle (*) :: \kappa \parallel n_2 :: n_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel (n_1 * n_2) :: s \parallel H \parallel M \rangle \\
\text>IDiv} & \langle (/) :: \kappa \parallel n_2 :: n_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel (n_1 / n_2) :: s \parallel H \parallel M \rangle \quad \text{if } n_2 \neq 0 \\
\text{IMod} & \langle (\%) :: \kappa \parallel n_2 :: n_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel (n_1 \% n_2) :: s \parallel H \parallel M \rangle \quad \text{if } n_2 \neq 0 \\
\text{IEq} & \langle (==) :: \kappa \parallel v_2 :: v_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel \text{eq}(v_1, v_2) :: s \parallel H \parallel M \rangle \\
\text{ILt} & \langle (<) :: \kappa \parallel n_2 :: n_1 :: s \parallel H \parallel M \rangle & \rightarrow \langle \kappa \parallel (n_1 < n_2) :: s \parallel H \parallel M \rangle
\end{array}$$

$$\text{where } \text{eq}(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2 = \text{iter}[a] \\ \text{true} & \text{if } v_1 = v_2 = n \\ \text{true} & \text{if } v_1 = v_2 = b \end{cases} \quad \begin{cases} \text{true} & \text{if } v_1 = v_2 = \text{undefined} \\ \text{true} & \text{if } v_1 = v_2 = \{\text{value} : v, \text{done} : b\} \\ \text{false} & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\text{IDef} \quad & \langle \text{def}[x_1, \dots, x_n][\sigma \vdash e] :: \kappa \parallel v_n :: \dots :: v_1 :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \langle \sigma[x_1 \mapsto a_1, \dots, x_n \mapsto a_n] \vdash e :: \kappa \parallel s \parallel H \parallel M[a_1 \mapsto v_1, \dots, a_n \mapsto v_n] \rangle \\
& \text{where } \forall 1 \leq p \leq n. a_p \notin \text{Domain}(M) \wedge (\forall 1 \leq q < p. a_q \neq a_p) \\
\\
\text{IWrite} \quad & \langle \text{write}[a] :: \kappa \parallel v :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \langle \kappa \parallel v :: s \parallel H \parallel M[a \mapsto v] \rangle \\
\\
\text{IPop} \quad & \langle \text{pop} :: \kappa \parallel v :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \langle \kappa \parallel s \parallel H \parallel M \rangle
\end{aligned}$$

4.2.1 Control Flow Instructions.

$$\begin{aligned}
\text{IJmpIf}_{\text{true}} \quad & \langle \text{jmp-if}[\langle \kappa \parallel s \parallel H \rangle] :: _ \parallel \text{true} :: _ \parallel _ \parallel M \rangle \rightarrow \langle \kappa \parallel s \parallel H \parallel M \rangle \\
\text{IJmpIf}_{\text{false}} \quad & \langle \text{jmp-if}[_] :: \kappa \parallel \text{false} :: s \parallel H \parallel M \rangle \rightarrow \langle \kappa \parallel s \parallel H \parallel M \rangle \\
\text{IJmp} \quad & \langle \text{jmp}[c] :: \kappa \parallel v :: s \parallel H \parallel M \rangle \rightarrow \langle \kappa' \parallel v :: s' \parallel H'' \parallel M \rangle \\
& \text{where } \begin{cases} H(c) = \langle \kappa' \parallel s' \parallel H' \rangle \\ H'' = \begin{cases} H'[\text{yield} \mapsto H(\text{yield})] & \text{if } \text{yield} \in \text{Domain}(H) \\ H' & \text{otherwise} \end{cases} \end{cases}
\end{aligned}$$

4.2.2 Function Call/Return Instructions.

$$\begin{aligned}
\text{ICall}_{\lambda} \quad & \langle \text{call}[n] :: \kappa \parallel v_n :: \dots :: v_1 :: \langle \lambda(x_1, \dots, x_m).e, \sigma' \rangle :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \langle \text{def}[x_1, \dots, x_m][\sigma' \vdash \text{return } e] :: \square \parallel s_{\text{body}} \parallel H_{\text{body}} \parallel M \rangle \\
& \text{where } \begin{cases} s_{\text{body}} = \begin{cases} v_m :: \dots :: v_1 :: \blacksquare & \text{if } n \geq m \\ \text{undefined} :: \dots :: \text{undefined} :: v_n :: \dots :: v_1 :: \blacksquare & \text{otherwise} \end{cases} \\ H_{\text{body}} = H[\text{return} \mapsto \psi_{\text{return}}] \setminus \{\text{break, continue, yield}\} \\ \psi_{\text{return}} = \langle \kappa \parallel s \parallel H \rangle \end{cases} \\
\\
\text{ICall}_{\lambda*} \quad & \langle \text{call}[n] :: \kappa \parallel v_n :: \dots :: v_1 :: \langle \lambda^*(x_1, \dots, x_m).e, \sigma' \rangle :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \langle \kappa \parallel \text{iter}[a] :: s \parallel H \parallel M[a \mapsto \psi_{\text{body}}] \rangle \\
& \text{where } \begin{cases} a \notin \text{Domain}(M) \\ \psi_{\text{body}} = \langle \kappa_{\text{body}} \parallel s_{\text{body}} \parallel \emptyset \rangle \\ \kappa_{\text{body}} = \text{pop} :: \text{def}[x_1, \dots, x_m][\sigma' \vdash \text{return } (\text{try } e \text{ catch } (x) x)] :: \square \\ s_{\text{body}} = \begin{cases} v_m :: \dots :: v_1 :: \blacksquare & \text{if } n \geq m \\ \text{undefined} :: \dots :: \text{undefined} :: v_n :: \dots :: v_1 :: \blacksquare & \text{otherwise} \end{cases} \\ x \text{ could be any identifier.} \end{cases} \\
\\
\text{IReturn} \quad & \langle \text{return} :: \kappa \parallel v :: s \parallel H \parallel M \rangle \\
\rightarrow \quad & \begin{cases} \langle \text{yield} :: \square \parallel v :: \text{true} :: \psi_{\text{done}} :: s \parallel H \parallel M \rangle & \text{if } \text{yield} \in \text{Domain}(H) \\ \langle \text{jmp}[\text{return}] :: \square \parallel v :: s \parallel H \parallel M \rangle & \text{otherwise} \end{cases}
\end{aligned}$$

4.2.3 Generator Instructions.

INext	$\langle \text{next} :: \kappa \parallel v :: \text{iter}[a] :: s \parallel H \parallel M \rangle$
\rightarrow	$\langle \kappa' \parallel v :: s' \parallel H_{\text{body}} \parallel M \rangle$
	where $\begin{cases} M(a) = \langle \kappa' \parallel s' \parallel H' \rangle \\ H_{\text{body}} = H'[\text{yield} \mapsto \psi, \text{return} \mapsto \psi] \\ \psi = \langle \kappa \parallel \text{iter}[a] :: s \parallel H \rangle \end{cases}$
IYield	$\langle \text{yield} :: _ \parallel v :: b :: v' :: _ \parallel H \parallel M \rangle$
\rightarrow	$\langle \kappa' \parallel \{\text{value} : v, \text{done} : b\} :: s' \parallel H' \parallel M[a \mapsto v'] \rangle$
	where $H(\text{yield}) = \langle \kappa' \parallel \text{iter}[a] :: s' \parallel H' \rangle$
IValueField	$\langle \text{value} :: \kappa \parallel \{\text{value} : v, \text{done} : _ \} :: s \parallel H \parallel M \rangle$
\rightarrow	$\langle \kappa \parallel v :: s \parallel H \parallel M \rangle$
IDoneField	$\langle \text{done} :: \kappa \parallel \{\text{value} : _, \text{done} : b\} :: s \parallel H \parallel M \rangle$
\rightarrow	$\langle \kappa \parallel b :: s \parallel H \parallel M \rangle$

And \rightarrow^* is the reflexive-transitive closure of \rightarrow and denotes the repeated reduction:

$$\begin{array}{c} \langle \kappa \parallel s \parallel H \parallel M \rangle \rightarrow^* \langle \kappa' \parallel s' \parallel H' \parallel M' \rangle \\ \langle \kappa' \parallel s' \parallel H' \parallel M' \rangle \rightarrow \langle \kappa'' \parallel s'' \parallel H'' \parallel M'' \rangle \\ \hline \langle \kappa \parallel s \parallel H \parallel M \rangle \rightarrow^* \langle \kappa'' \parallel s'' \parallel H'' \parallel M'' \rangle \end{array}$$

The evaluation result of an expression e is the value v if

$$\langle (\emptyset \vdash e) :: \square \parallel \blacksquare \parallel \emptyset \parallel \emptyset \rangle \rightarrow^* \langle \square \parallel v :: \blacksquare \parallel _ \parallel _ \rangle$$