**Massachusetts Institute of Technology**
**Department of Electrical Engineering and Computer Science**



# 1 Concrete Syntax

## 1.1 Lexical Considerations

Keywords and identifiers are case-sensitive. For example, `if` is a keyword, but `IF` is an identifier; `foo` and `Foo` are two different identifiers referring to two distinct variables.

The keywords are: `global` `if` `else` `while` `return` `fun` `true` `false` `None`

Comments are started by `//` and are terminated by the end of the line.

White space may appear between any lexical tokens (but not within a single token). White space is defined as one or more spaces, tabs, line-break characters (carriage return, line feed, form feed), and comments.

Keywords and identifiers must be separated by white space, or a token that is neither a keyword nor an identifier. For example, `intfortrue` is a single identifier, not three distinct keywords. If a sequence begins with an alphabetic character or an underscore, then it and the longest sequence of alphanumeric characters following it forms a token.

String literals are composed of ⟨char⟩s enclosed in double quotes.

If a sequence begins with a decimal digit, then the longest following sequence of decimal digits forms a decimal integer literal. A long sequence of digits (e.g., `12345678912345678912 3`) is scanned as a single token.

A ⟨char⟩ is any printable ASCII character (ASCII values between decimal value 32 and 126 inclusive) other than quote ( `"` ) or backslash ( `\` ), plus the 2-character sequences `\"` to denote quote, `\\` to denote backslash, `\t` to denote a literal tab, or `\n` to denote newline.

## 1.2 Reference Grammar

| Notation | Meaning |
|---|---|
| ⟨foo⟩ | means foo is a nonterminal. |
| **foo** | (in **bold** font) means that **foo** is a terminal. |
| $\left[ x \right]$ | means zero or one occurrence of $x$, i.e., $x$ is optional; note that brackets in quotes `'['` `']'` are terminals. |
| $x^*$ | means zero or more occurrences of $x$. |
| $x^+$**,** | a comma-separated list of one or more $x$. note that there is no comma following the last of $x$. |
| $\left\{ \right.$ $\left. \right\}$ | large braces are used for grouping; note that braces in quotes `'{'` `'}'` are terminals. |
| \| | separates alternatives. |

⟨program⟩ → ⟨statement⟩*

⟨statement⟩ → ⟨location⟩ `=` ⟨expr⟩ `;`
| ⟨call⟩ `;`
| **global** ⟨id⟩ `;`
| **if** `(` ⟨expr⟩ `)` ⟨block⟩ [**else** ⟨block⟩]
| **while** `(` ⟨expr⟩ `)` ⟨block⟩
| **return** ⟨expr⟩ `;`

⟨block⟩ → `'{'` ⟨statement⟩* `'}'`

⟨expr⟩ → **fun** `(` [⟨id⟩+`,`] `)` ⟨block⟩
| `'{'` {⟨id⟩ `:` ⟨expr⟩ `;`}* `'}'`
| ⟨simple_expr⟩

⟨simple_expr⟩ → ⟨location⟩
| ⟨call⟩
| ⟨literal⟩
| `-` ⟨simple_expr⟩
| `!` ⟨simple_expr⟩
| ⟨simple_expr⟩ ⟨bin_op⟩ ⟨simple_expr⟩
| `(` ⟨simple_expr⟩ `)`

⟨bin_op⟩ → ⟨arith_op⟩ | ⟨comp_op⟩ | ⟨cond_op⟩

⟨arith_op⟩ → `+` | `-` | `*` | `/`

⟨comp_op⟩ → `<` | `>` | `<=` | `>=` | `==`

⟨cond_op⟩ → `&` | `|`

⟨call⟩ → ⟨location⟩ `(` [⟨expr⟩+`,`] `)`

⟨location⟩ → ⟨id⟩
| ⟨location⟩ `.` ⟨id⟩
| ⟨location⟩ `'['` ⟨expr⟩ `']'`

⟨literal⟩ → ⟨int_literal⟩ | ⟨string_literal⟩ | ⟨bool_literal⟩ | **None**

⟨int_literal⟩ → ⟨digit⟩ ⟨digit⟩*

⟨string_literal⟩ → `"` ⟨char⟩* `"`

⟨bool_literal⟩ → `true` | `false`

⟨id⟩ → ⟨alpha⟩ ⟨alpha_num⟩*

⟨alpha_num⟩ → ⟨alpha⟩ | ⟨digit⟩

⟨alpha⟩ → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z` | `_`

⟨digit⟩ → `0` | `1` | `2` | `...` | `9`

### 1.3 Operator Precedence and Associativity

Operator precedence is a set of rules that determine the priority with which operators are evaluated in expressions containing multiple operators. Specifically, operators with higher precedence are grouped and applied before those with lower precedence, allowing accurate translation of written formulas into executable behavior without requiring explicit parentheses.

Operator associativity refers to the set of rules that determine the order in which operators of the same precedence are evaluated in an expression. For binary operators, this occurs either left to right or right to left.

Operator precedence is defined in the following order from highest to lowest:

| *Operators* | *Comments* |
|---|---|
| `-` | unary minus |
| `* /` | multiplication, division |
| `+ -` | addition, subtraction |
| `< <= >= > ==` | relational |
| `!` | conditional not |
| `&` | conditional and |
| `\|` | conditional or |

All binary operators are left associative in MITScript.

## 2 Abstract Syntax

An MITScript program consists of a sequence of *statements*, which may contain *expressions* and *operators* as well as *identifiers* $x \in X$ that denote program variables or field names of records. The core syntax of the language is specified by the following abstract grammar:

Expression $e ::= x \mid \texttt{true} \mid \texttt{false} \mid n \mid st \mid \texttt{None} \mid e_1\ lop\ e_2 \mid e_1\ op\ e_2 \mid e_1\ cmp\ e_2 \mid uop\ e$
$\mid \texttt{\{}x_1 : e_1, \ldots, x_n : e_n\texttt{\}} \mid e.x \mid e_1\texttt{[}e_2\texttt{]} \mid \texttt{fun}\ x\ s \mid e_1\texttt{(}e_2\texttt{)}$

Statement $s ::= x = e \mid e_1.x = e_2 \mid e_1\texttt{[}e_2\texttt{]} = e_3 \mid \texttt{if}\ e\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ e\ s \mid s_1; s_2 \mid \texttt{global}\ x \mid \texttt{return}\ e$

Operator $lop ::= \texttt{\&} \mid \texttt{|} \qquad op ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \qquad cmp ::= \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \qquad uop ::= \texttt{-} \mid \texttt{!}$

Note that this abstract syntax, which we use to specify the language semantics, deviates from the concrete syntax you will use in your implementation to parse real MITScript programs. The concrete syntax specifies details such as the delimiting of blocks via curly braces and the precedence of operators. While the abstract syntax assumes for simplicity that functions take exactly one argument, the concrete syntax generalizes them to any number of arguments. The grammar for the concrete syntax can be found in Section 1.2.

## 3 Semantics Preliminaries

**Values.** The semantics of the language is defined in terms of *values* $v \in V$, which correspond to the objects that are created, stored in memory, and bound to variables as the program evaluates:

Value $v ::= \texttt{Bool}(b) \mid \texttt{Integer}(n) \mid \texttt{String}(st) \mid \texttt{Record}(a) \mid \texttt{Function}(a, x, s) \mid \texttt{None}$

A value in the program falls in one of the following six categories:

- A *Boolean* is denoted by the constructor `Bool`$(b)$, where $b$ is either `true` or `false`.
- An *integer* is denoted by the constructor `Integer`$(n)$, where $n \in \mathbb{Z}$.
- A *string* is denoted by the constructor `String`$(st)$, where $st$ is an ASCII string (including standard escape sequences).
- A *record* is denoted by the constructor `Record`$(a)$, where $a \in A = \mathbb{N} \cup \{\cdot\}$ is an *address*, either a natural number or the distinguished value $\cdot$ which represents an invalid address, similar to a null pointer in other languages. For a record, the address $a$ in question always points to a *map* $m \in X \to A$ from field names to addresses of field values.
- A *function* is denoted by the constructor `Function`$(a_f)$, where $a_f$ is the address of a map $m$ of the form $m = \{\mathsf{context} : a, \mathsf{formal\text{-}argument} : x, \mathsf{body} : s\}$. Here, $a$ is the address of the function's *stack frame* (to be defined in Section 3), $x$ is the function's formal argument, and $s$ is the statement corresponding to the function's body.
- A *null* value is denoted by the singleton value `None`.

These explicit constructors do not appear literally in the syntax of MITScript programs, which instead consists of expressions and statements. For example, rather than an instance of `Record`$(a)$, a program would contain an instance of the expression `{`$x_1 : e_1, \ldots, x_n : e_n$`}`. Instead, values represent the physical objects stored in memory as the program executes, including their under-the-hood bookkeeping.

**Program State.** The state of an MITScript program is represented by a *heap* $h$ and a *stack* $\gamma$. The heap $h$ is a mapping from addresses $a$ to objects allocated on the heap, each of which is either a value $v$ or a *stack frame* $\sigma$. The stack $\gamma$ is a sequence of addresses of stack frames allocated on the heap.

**Stack Frames.** A *stack frame* $\sigma \in (X \cup \{\mathsf{parent}, \mathsf{global}\}) \to A$ is a mapping from program variables to addresses that point to the contents of the program variables stored in the heap. A stack frame has two additional distinguished fields. The parent field stores the address of the stack frame's parent frame. The global field stores a pair $(X_g, a)$, where $X_g$ is a set of program variables that have been declared as global within the stack frame by `global` statements and $a$ is the address of a distinguished global stack frame.

We use the notation $\gamma = \gamma'; a$ to denote that the stack $\gamma$ is equal to another stack $\gamma'$ with an address $a$ added at the top. Similarly, we use the notation $\gamma = a_1; \gamma'; a_2$ to denote that $\gamma$ is a stack that has $a_1$ at the bottom, followed by another stack $\gamma'$ followed by an address $a_2$ at the top.

**Program Execution.** The program starts its execution with a distinguished global stack frame $\sigma_g = \{\mathsf{parent} : \cdot, \mathsf{global} : (\{\}, \cdot)\}$. The initial heap $h_0$ contains only this initial global stack frame and a pre-allocated value for `None`, i.e. $h_0 = \{a_g : \sigma_g, a_{\texttt{None}} : \texttt{None}\}$. The initial stack $\gamma_0$ contains the address of the global stack frame, i.e. $\gamma_0 = a_g$. Additionally, it should contain any definitions that are needed to appropriately support MITScript's native functions (Section 7).

**Errors.** When a program performs an illegal operation, execution must stop, and the interpreter must report one of the following exceptions by printing the type of the exception to console via standard output:

- `UninitializedVariableException`: when the program attempts to read from a variable that is not present in any appropriate stack frame.
- `IllegalCastException`: when the program attempts to apply an operation to a value for which it does not apply.
- `IllegalArithmeticException`: when the program attempts to divide by zero.
- `RuntimeException`: other illegal operations, such as passing too many arguments to a function.

The semantics in the following sections indicates exactly when each exception must be reported.

**Integer Representation.** You should implement MITScript's integers under the standard semantics of 32-bit two's-complement signed integers. You do not need to report errors for integer overflow or underflow.

**Native Functions.** In addition to the core syntax and semantics of MITScript, your implementation must support the three native, predefined global functions described in Section 7.

**Scoping.** In MITScript, a program variable $x$ may reside in different stack frames depending on whether it is global. To identify the appropriate stack frame for a newly declared variable, we define in Figure 1 a function named lookup-write that takes the address $a_1$ of the current stack frame, the current heap $h$, and the variable $x$, and returns the address of the stack frame into which $x$ should be written. This address is either the address $a_2$ of the global frame if $x$ is global, or the address $a_1$ of the current frame otherwise.

Similarly, we define in Figure 2 a function named lookup-read that determines the address of the stack frame in which an existing variable $x$ resides. This address is either that of the global frame, the current frame, or some parent of the current frame. If the variable has not been defined in any appropriate stack frame, then the interpreter must raise an `UninitializedVariableException`.

# 4 Statement Semantics

Figure 3 presents the semantics of statements. The evaluation relation for statements $(\gamma, h, s) \to \eta$ denotes that execution of a statement $s$ from a stack $\gamma$ and a heap $h$ yields a *heap-value* $\eta$. A heap value $\eta$ is either a heap $h$ or a *return-value* denoted by the constructor $\texttt{Return}(h, v)$, where $h$ is a heap and $v$ is a value. The purpose of the distinguished return-value is to enable the semantics specification to appropriately handle cases where `return` statements appear arbitrarily within the body of a function.

GlobalLookupWrite

$$\frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathsf{global}) = (X_g, a_2) \qquad x \in X_g}{\mathsf{lookup\text{-}write}(a_1, h, x) = a_2}$$

LocalLookupWrite

$$\frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathsf{global}) = (X_g, a_2) \qquad x \notin X_g}{\mathsf{lookup\text{-}write}(a_1, h, x) = a_1}$$

Figure 1: Semantics of the lookup-write function.

GlobalLookupRead

$$\frac{\begin{array}{c} h(a_1) = \sigma_1 \\ \sigma_1(\mathsf{global}) = (X_g, a_2) \qquad x \in X_g \qquad h(a_2) = \sigma_2 \qquad x \in \mathrm{dom}(\sigma_2) \qquad \sigma_2(x) = a_3 \end{array}}{\mathsf{lookup\text{-}read}(a_1, h, x) = a_3}$$

LocalLookupRead

$$\frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathsf{global}) = (X_g, a_2) \qquad x \notin X_g \qquad x \in \mathrm{dom}(\sigma_1) \qquad \sigma_1(x) = a_3}{\mathsf{lookup\text{-}read}(a_1, h, x) = a_3}$$

ScopedLookupRead

$$\frac{\begin{array}{c} h(a_1) = \sigma_1 \qquad \sigma_1(\mathsf{global}) = (X_g, a_2) \qquad x \notin X_g \qquad x \notin \mathrm{dom}(\sigma_1) \\ \mathsf{lookup\text{-}read}(\sigma(\mathsf{parent}), h, x) = a_3 \end{array}}{\mathsf{lookup\text{-}read}(a_1, h, x) = a_3}$$

Figure 2: Semantics of the lookup-read function.

VarAssignment

$$\frac{(\gamma; a_1, h, e) \to (h', v) \quad \mathsf{lookup\text{-}write}(a_1, h', x) = a_2 \quad \sigma = h'(a_2) \quad a_3 \notin \mathrm{dom}(h') \quad \sigma' = \sigma[x : a_3]}{(\gamma; a_1, h, x = e) \to h'[a_3 : v][a_2 : \sigma']}$$

HeapAssignment

$$\frac{(\gamma, h, e_1) \to (h', \texttt{Record}(a_1)) \quad (\gamma, h', e_2) \to (h'', v) \quad h''(a_1) = m \quad a_2 \notin \mathrm{dom}(h'') \quad m' = m[x : a_2]}{(\gamma, h, e_1.x = e_2) \to h''[a_2 : v][a_1 : m']}$$

HeapIndexAssignment

$$\frac{(\gamma, h, e_1) \to (h', \texttt{Record}(a_1)) \quad (\gamma, h', e_2) \to (h'', v_1) \quad x = \mathsf{str}(v_1) \quad (\gamma, h'', e_3) \to (h''', v_2) \quad h'''(a_1) = m \quad a_2 \notin \mathrm{dom}(h''') \quad m' = m[x : a_2]}{(\gamma, h, e_1\,\texttt{[}e_2\texttt{]} = e_3) \to h'''[a_2 : v_2][a_1 : m']}$$

IfTrue

$$\frac{(\gamma, h, e) \to (h', \texttt{Bool(true)}) \quad (\gamma, h', s_1) \to \eta}{(\gamma, h, \texttt{if}\ e\ s_1\ \texttt{else}\ s_2) \to \eta}$$

IfFalse

$$\frac{(\gamma, h, e) \to (h', \texttt{Bool(false)}) \quad (\gamma, h', s_2) \to \eta}{(\gamma, h, \texttt{if}\ e\ s_1\ \texttt{else}\ s_2) \to \eta}$$

While

$$\frac{(\gamma, h, \texttt{if}\ e\ (s; \texttt{while}\ e\ s)) \to \eta}{(\gamma, h, \texttt{while}\ e\ s) \to \eta}$$

Sequence

$$\frac{(\gamma, h, s_1) \to h' \quad (\gamma, h', s_2) \to \eta}{(\gamma, h, s_1; s_2) \to \eta}$$

SequenceReturn

$$\frac{(\gamma, h, s_1) \to \texttt{Return}(h', v)}{(\gamma, h, s_1; s_2) \to \texttt{Return}(h', v)}$$

Global

$$\frac{}{(\gamma, h, \texttt{global}\ x) \to h}$$

Return

$$\frac{(\gamma, h, e) \to (h', v)}{(\gamma, h, \texttt{return}\ e) \to \texttt{Return}(h', v)}$$

Figure 3: Semantics of statements.

**Variable Assignment.** This rule first evaluates the expression $e$ to a value $v$ and a new heap $h'$. First, it uses lookup-write (Section 3) to determine the address $a_2$ of the stack frame $\sigma$ in which to update the variable $x$, using the appropriate scoping rules. It allocates an new address $a_3$ previously unused in $h'$. It returns a new stack with $x$ set to $a_3$ and a new heap with $a_3$ pointing to $v$ and $a_2$ pointing to the new stack.

**Heap Assignment.** This rule does not not update a stack frame but rather updates a record in the heap. This statement must raise an `IllegalCastException` if the result of the evaluation of $e_1$ is not a record.

**Heap Index Assignment.** This rule is similar to the previous one, except the field name is dynamic. The index expression is evaluated and cast to a string using the str function, which is defined in Section 6. The operation must raise an `IllegalCastException` if the result of evaluation of $e_1$ is not a record.

**If.** These rules execute the first branch if the condition evaluates to true and the second branch otherwise. The statement must raise an `IllegalCastException` if the result of the evaluation of $e$ is not a Boolean.

**While.** This rule repeatedly executes its body while the condition evaluates to true. The statement must raise an `IllegalCastException` if the result of the evaluation of $e$ is not a Boolean.

**Sequence.** This rule evaluates the first statement and then evaluates the second statement. Note that if the first statement yields a return-value, then the program skips the execution of the second statement.

**Global.** This rule is a no-op; its semantics is only incorporated later during function calls.

**Return.** This rule evaluates the expression and yields a return-value.

## 5 Expression Semantics

Figures 4 to 6 present the semantics of expressions. The evaluation relation for expressions $(\gamma, h, e) \rightarrow (h', v)$ denotes that given a stack $\gamma$ and a heap $h$, an expression $e$ evaluates to a new heap $h'$ and a value $v$.

**Constants.** For each type of constant value, the rule generates a value corresponding to the constant.

**Logical Operators.** Logical operators can only be applied to Boolean values.

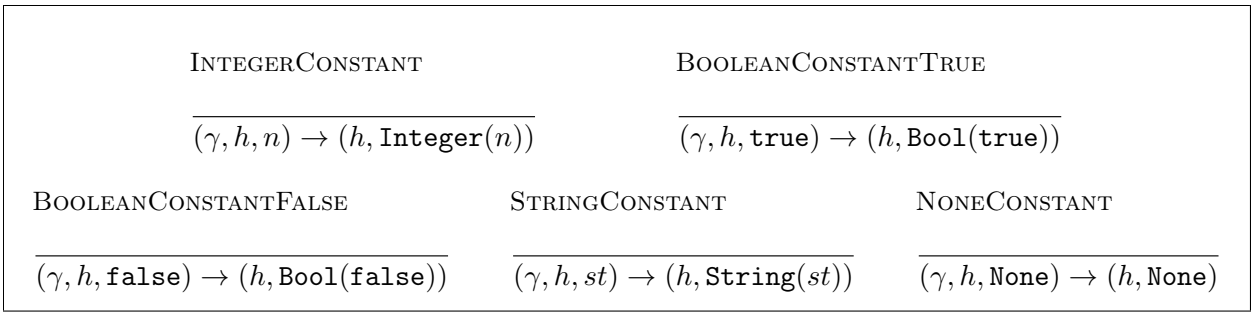


Figure 4: Semantics of constant expressions.

**Arithmetic Operators.** The arithmetic operators `-`, `*`, and `/` can only be applied to integer values. The interpreter must generate an `IllegalArithmeticException` for division by zero.

The operator `+` on two integers corresponds to standard integer addition. The operator `+` on two strings corresponds to string concatenation. The operator `+` on a string and a value that is not a string (where either can appear on either side of the `+`) causes the non-string value to be cast to a string.

**Unary Operators.** The unary minus operator `-` can only be applied to an integer value. The unary negation operator `!` can only be applied to a Boolean value.

**Comparison Operators.** The comparison operators `<`, `>`, `<=`, `>=` are only defined for integers.

Two integers, two Booleans, or two strings are equal if their values are equal. The value `None` is equal to itself. Equality between two records checks that their addresses are identical. Similarly, equality between two functions checks that their addresses are identical. Equality between two values of different type returns `false`.

**Operator Errors.** For the above operators, the interpreter must generate an `IllegalCastException` for any other operation that is not given an explicit semantics by the rules.

**Variable Read.** This rule specifies the semantics of reading from a variable, using lookup-read (Section 3) to obtain the appropriate address of $x$.

**Record Constructor.** This rule evaluates all of the expressions inside the list and constructs a `Record` value from fields to values. Note that the initializer expressions must be evaluated in order.

**Field Read.** These rules evaluate the expression to a `Record` value, from which the field $x$ is looked up. If the record does not contain the named field $x$, the expression evaluates to None. The statement raises an `IllegalCastException` if the result of the evaluation of $e$ is not a record.

LogicalOperation
$$\frac{(\gamma, h, e_1) \to (h', \texttt{Bool}(b_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{Bool}(b_2))}{(\gamma, h, e_1 \ lop \ e_2) \to (h'', \texttt{Bool}(b_1 \ lop \ b_2))}$$

ArithmeticOperation
$$\frac{(\gamma, h, e_1) \to (h', \texttt{Integer}(n_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{Integer}(n_2))}{(\gamma, h, e_1 \ op \ e_2) \to (h'', \texttt{Integer}(n_1 \ op \ n_2))}$$

UnaryMinus
$$\frac{(\gamma, h, e) \to (h', \texttt{Integer}(n))}{(\gamma, h, \texttt{-}e) \to (h', \texttt{Integer}(-n))}$$

UnaryNot
$$\frac{(\gamma, h, e) \to (h', \texttt{Bool}(b))}{(\gamma, h, \texttt{!}e) \to (h', \texttt{Bool}(\neg b))}$$

ComparisonOperation
$$\frac{(\gamma, h, e_1) \to (h', \texttt{Integer}(n_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{Integer}(n_2))}{(\gamma, h, e_1 \ cmp \ e_2) \to (h'', \texttt{Bool}(n_1 \ cmp \ n_2))}$$

StringConcatenation
$$\frac{(\gamma, h, e_1) \to (h', \texttt{String}(st_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{String}(st_2))}{(\gamma, h, e_1 \texttt{ + } e_2) \to (h'', \texttt{String}(st_1 + st_2))}$$

StringConcatenationLeftCast
$$\frac{(\gamma, h, e_1) \to (h', t(\delta)) \qquad (\gamma, h', e_2) \to (h'', \texttt{String}(st_2)) \qquad t \neq \texttt{String} \qquad st_1 = \mathsf{str}(t(\delta))}{(\gamma, h, e_1 \texttt{ + } e_2) \to (h'', \texttt{String}(st_1 + st_2))}$$

StringConcatenationRightCast
$$\frac{(\gamma, h, e_1) \to (h', \texttt{String}(st_1)) \qquad (\gamma, h', e_2) \to (h'', t(\delta)) \qquad t \neq \texttt{String} \qquad st_2 = \mathsf{str}(t(\delta))}{(\gamma, h, e_1 \texttt{ + } e_2) \to (h'', \texttt{String}(st_1 + st_2))}$$

NoneEquality
$$\frac{(\gamma, h, e_1) \to (h', \texttt{None}) \qquad (\gamma, h', e_2) \to (h'', \texttt{None})}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \texttt{Bool(true)})}$$

PrimitiveEquality
$$\frac{(\gamma, h, e_1) \to (h', t_1(\delta_1)) \qquad (\gamma, h', e_2) \to (h'', t_2(\delta_2)) \qquad t_1 = t_2}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \texttt{Bool}(\delta_1 = \delta_2))} \ t_1, t_2 \in \{\texttt{Integer}, \texttt{Bool}, \texttt{String}\}$$

RecordEquality
$$\frac{(\gamma, h, e_1) \to (h', \texttt{Record}(a_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{Record}(a_2))}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \texttt{Bool}(a_1 = a_2))}$$

FunctionEquality
$$\frac{(\gamma, h, e_1) \to (h', \texttt{Function}(a_1)) \qquad (\gamma, h', e_2) \to (h'', \texttt{Function}(a_2))}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \texttt{Bool}(a_1 = a_2))}$$

PrimitiveEqualityMismatched
$$\frac{(\gamma, h, e_1) \to (h', t_1(\delta_1)) \qquad (\gamma, h', e_2) \to (h'', t_2(\delta_2)) \qquad t_1 \neq t_2}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \texttt{Bool(false)})}$$

Figure 5: Semantics of arithmetic and logical expressions.

**Index Read.** These rules evaluate the dynamic index value. If it evaluates to a string, the field by that name is obtained from the record. If it evaluates to some other value, then that value is first cast to a string (Section 6). As before, if there is no such field present in the record, then the expression evaluates to `None`.

**Function Creation.** This rule creates a new function value that captures the current frame.

**Function Call.** There are multiple steps to this rule. First, the base expression for the function call is evaluated to a function value, which points to a map containing the address of a stack frame and the code for the function.

The next step is to allocate a new stack frame for the function call. The rule creates a new stack frame whose parent is the function's stack frame. The rule traverses the body of the function, and for every global declaration `global` $x$, it adds $x$ to the set of global variables for the new stack frame. A helper function globals (Figure 7) performs this traversal. Then, for every assignment of the form $x = e$ in the body of $f$, the rule adds variable $x$ to the stack frame and initializes it to `None`. A helper function assigns (Figure 8) performs this traversal. Finally, the rule evaluates the body of the function with the new stack and heap.

Note that the rule is written assuming a function with one argument, and should be generalized to support multiple arguments. When there are multiple arguments, they must be evaluated in order from left to right. If $e_1$ does not evaluate to a `Function` value, then `IllegalCastException` must be raised. If the caller passes too many or too few arguments to the function, then `RuntimeException` must be raised.

**Return.** These rules specify that a function call evaluates to the return-value if it exists or else to `None`.

# 6 Cast Semantics

The semantics of MITScript requires that certain values be automatically converted into strings. This conversion is performed via a function called str, defined by the following rules:

- $\mathsf{str}(\texttt{String}(st)) = st$
- $\mathsf{str}(\texttt{Boolean}(b)) =$ "`true`" if $b$ is true, or "`false`" otherwise
- $\mathsf{str}(\texttt{Integer}(n)) =$ base-10 representation of $n$, with no leading zeroes, preceded by sign if negative
- $\mathsf{str}(\texttt{Function}(a_f)) =$ "`FUNCTION`"
- $\mathsf{str}(\{x_i : v_i, \ldots\}) =$ "`{`" $+ x_i +$ "`:`" $+ \mathsf{str}(v_i) +$ "` `" $+ \cdots +$ "`}`"
  - Fields should be ordered lexicographically in the resulting string (i.e., $i < j \implies x_i < x_j$)
- $\mathsf{str}(\texttt{None}) =$ "`None`"

VariableRead

$$\frac{\mathsf{lookup\text{-}read}(a_1, h, x) = a_2 \qquad h(a_2) = v}{(\gamma; a_1, h, x) \to (h, v)}$$

Record

$$\frac{(\gamma, h, e_1) \to (h_1, v_1) \quad \ldots \quad (\gamma, h_{n-1}, e_n) \to (h_n, v_n) \qquad \{a_1, \ldots, a_{n+1}\} \notin \mathrm{dom}(h_n) \qquad m = \{x_1 : a_1, \ldots, x_n : a_n\} \qquad h' = h_n[a_1 : v_1, \ldots, a_n : v_1, a_{n+1} : m]}{(\gamma, h, \{x_1 : e_1, \ldots, x_n : e_n\}) \to (h', \mathtt{Record}(a_{n+1}))}$$

FieldRead

$$\frac{(\gamma, h, e) \to (h', \mathtt{Record}(a_1)) \qquad h'(a_1) = m \qquad x \in \mathrm{dom}(m) \qquad a_2 = m(x)}{(\gamma, h, e.x) \to (h', h'(a_2))}$$

FieldReadFail

$$\frac{(\gamma, h, e) \to (h', \mathtt{Record}(a)) \qquad h'(a) = m \qquad x \notin \mathrm{dom}(m)}{(\gamma, h, e.x) \to (h', \mathtt{None})}$$

IndexRead

$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \mathtt{Record}(a_1)) \\ h'(a_1) = m \qquad (\gamma, h', e_2) \to (h'', v_1) \qquad x = \mathsf{str}(v_1) \qquad x \in \mathrm{dom}(m) \qquad a_2 = m(x)\end{array}}{(\gamma, h, e_1\,\mathtt{[}e_2\mathtt{]}) \to (h'', h'(a_2))}$$

IndexReadFail

$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \mathtt{Record}(a)) \\ h'(a) = m \qquad (\gamma, h', e_2) \to (h'', v_1) \qquad x = \mathsf{str}(v_1) \qquad x \notin \mathrm{dom}(m)\end{array}}{(\gamma, h, e_1\,\mathtt{[}e_2\mathtt{]}) \to (h'', \mathtt{None})}$$

Function

$$\frac{m = \{\mathsf{context} : a, \mathsf{formal\text{-}argument} : x, \mathsf{body} : s\} \qquad a_f \notin \mathrm{dom}(h) \qquad h' = h[a_f : m]}{(\gamma; a, h, \mathtt{fun}\ x\ s) \to (h', \mathtt{Function}(a_f))}$$

FunctionCall

$$\frac{\begin{array}{c}(a_g; \gamma; a, h_1, e_1) \to (h_2, \mathtt{Function}(a_f)) \\ m = h_2(a_f) \qquad a_c = m(\mathsf{context}) \qquad x = m(\mathsf{formal\text{-}argument}) \\ s = m(\mathsf{body}) \qquad (a_g; \gamma; a, h_2, e_2) \to (h_3, v) \qquad a_2 \notin \mathrm{dom}(h_3) \qquad a_3 \notin (\mathrm{dom}(h_3) \cup \{a_2\}) \\ \sigma = \{\mathsf{parent} : a_c, \mathsf{global} : (\mathsf{globals}(s), a_g))\}[x_i : a_{\mathtt{None}} \mid x_i \in \mathsf{assigns}(s)][x : a_2] \\ h_4 = h_3[a_2 : v][a_3 : \sigma] \qquad (a_g; \gamma; a_3, h_4, s) \to \eta\end{array}}{(a_g; \gamma; a, h_1, e_1\mathtt{(}e_2\mathtt{)}) \to \eta}$$

FunctionCallReturn

$$\frac{(\gamma, h, e_1\mathtt{(}e_2\mathtt{)}) \to \mathtt{Return}(h', v)}{(\gamma, h, e_1\mathtt{(}e_2\mathtt{)}) \to (h', v)}$$

FunctionCallNoReturn

$$\frac{(\gamma, h, e_1\mathtt{(}e_2\mathtt{)}) \to h'}{(\gamma, h, e_1\mathtt{(}e_2\mathtt{)}) \to (h', \mathtt{None})}$$

Figure 6: Semantics of data and function expressions.

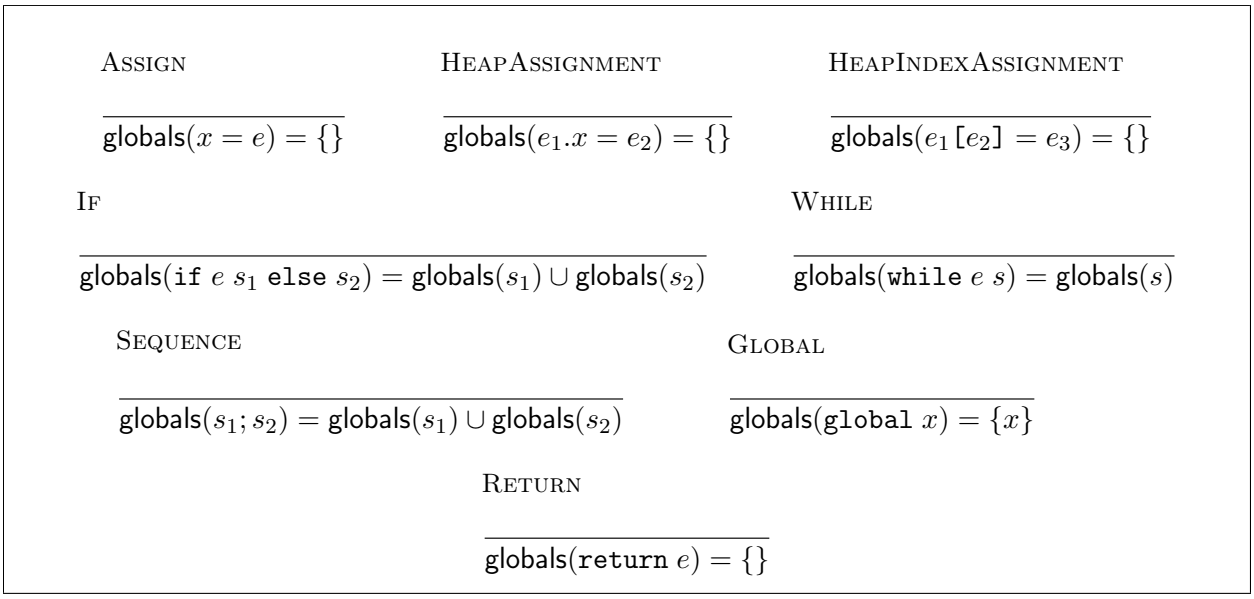


Figure 7: Definition of the globals function.

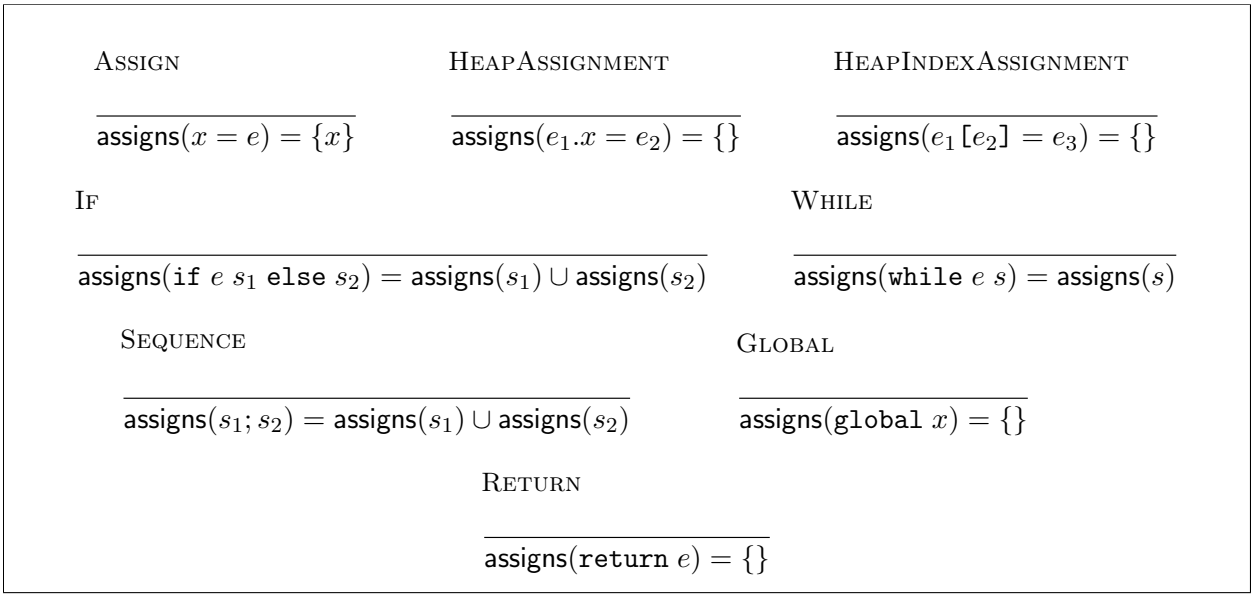


Figure 8: Definition of the assigns function.

# 7 Native Functions

In addition to the core syntax and semantics of MITScript, your interpreter must support the following three native functions:

- `print(s)` Cast `s` to a string and print it to the console (via standard output) followed by a newline.
- `input()` Read a line of input from the console (via standard input) and return it as a string value.
- `intcast(s)` Parse `s` as an integer value. If the string does not represent an integer (e.g., "`hello`"), this function raises an `IllegalCastException`. You may internally utilize the C function `atoi`.

These functions should be initially available in the global scope. However, it should be possible for the program to redefine them. For example, the program below updates `print` to add a prefix to every message:

```
print("Hello"); // this should print ‘Hello’ to the console.
oldprint = print;
print = fun(s){
    oldprint("OUTPUT: " + s);
};
print("Hello"); // this should print ‘OUTPUT: Hello’ to the console.
```

You should also implement equality for native functions. Comparing a native function with $f_1$ with $f_2$ should only return true if $f_1$ and $f_2$ both refer to the same native function. Example behavior is given below.

```
print(input == input); // this should print ‘true’ to the console.
print(input == intcast); // this should print ‘false’ to the console.
also_input = input;
print(also_input == input); // this should print ‘true’ to the console.
```