# MITScript Language Specification

## 1 Preliminaries

We denote MITScript expressions by $e \in E$ and MITScript statements by $s \in S$. A program variable or field name $x \in X$ is an element of the set of all valid program variables and field names $X$.

**Values.** A value $v \in V$ in the program can be one of the following six categories: Bool, Integer, String,Record, Function, or None. We denote a boolean value by the constructor Bool($b$) where $b \in \{\mathit{True}, \mathit{False}\}$ We denote an integer value by the constructor Integer($n$) where $n \in Z$. We denote a string value by the constructor String($st$) where $st$ is a string in the language of valid MITScript strings. We denote a record by the constructor Record($m$) where $m \in M = X \to A$ is a *map*, mapping a program variable or field name to an *address* $a \in A = (N \cup \cdot)$. An address is a natural number or the distinguished value $\cdot$ which represents an invalid address (e.g., similar to `NULL` in other languages).

We denote a function value by the constructor Function($a, x, s$), where $a$ is an address of the function's *frame* (as described below), $x$ is the function's formal argument, and $s$ is the MITScript code corresponding to the function's body. The value None is a special singleton value, similar to null.

In your implementation you should expect to use objects to represent all the different types of values. Specifically, it is recommended that you define a Value class, and then make each of the different value types subclasses of Value.

**Program State.** We represent the state of an MITScript program by a stack and a heap. Because of closures, however, we model the stack itself as a series of addresses of heap allocated stack frames.

A frame $\sigma \in \Sigma = \{X \cup \{\mathit{parent}, \mathit{global}\}\}A$ is a mapping from program variables and field names to addresses that store in the heap the contents of the program variables. A frame has two additional distinguished fields. The *parent* field stores a pointer to the current 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 frame (by `global` statements) and $a$ the address of the global frame.

The state of the program includes a heap $h : A \to v$ that maps addresses $a$ to heap allocated values $v$ which can either be stack frames or one of the six types of values listed above.

It also includes a stack $\gamma$ of references to stack frames. We will use the notation $\gamma = \gamma'; a$ to indicate 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 indicate 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.

**Native Functions.** Your implementation should support the three native, predefined global functions described in Section 5,

**Execution.** The program starts its execution with a stackframe $\sigma = \{\mathit{parent} : \cdot\}$ with a parent pointer equal to $\cdot$. The initial heap contains only this initial global stackframe and a pre-allocated value for *None*, ($h_0 = \{a_g : \sigma, a_{None} : \mathit{None}\}$). The initial stack contains just the global stack frame in the heap $\gamma_0 = a_g$ as well as the definitions that you need to appropriately support MITScript's native functions.

**Errors.** When a program performs an illegal operation, execution should stop, and your interpreter must report one of the following exceptions:

- **UninitializedVariableException:** when the program attempts to read from a variable in the stack-frame that has not been initialized.
- **IllegalCastException:** when the program attempts to apply an operation to value type for which it does not apply.
- **IllegalArithmeticException:** when the program attempts to divide by zero.
- **RuntimeException:** other error situations, such as passing too many arguments to a function.

In the semantics below we indicate exactly which of these three exceptions your interpreter should throw.

## 2 Statements

Figure 1 presents the semantics of statements. The evaluation relation for statements is of the form $(\gamma, h, s) \to \eta$ which denotes that execution of a statement $s$ from a stack $\gamma$ and a heap $h$ evaluates to a *heap-value* $\eta$. A heap value $\eta$ is either a heap $h \in H$ or a *return-value* denoted by the constructor $\text{Return}(h, v)$ where $h$ is a heap and $v$ is a value. A distinguished return-value enables the semantics specification to appropriately handle cases where `return` statements appear arbitrarily within the body of a function.

VarAssignment
$$\frac{(\gamma, h, e) \to (h', v) \qquad \mathit{lookup\_write}(a_1, h', x) = a_2 \qquad \sigma = h'(a_2) \qquad a_3 \notin \mathit{dom}(h) \qquad \sigma' = \sigma[x : a_3]}{(\gamma; a_1, h, x \texttt{ = } e) \to h'[a_3 : v][a_2 : \sigma]}$$

HeapAssignment
$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \text{Record}(a_1)) \\ (\gamma, h', e_2) \to (h'', v) \qquad h''(a_1) = m \qquad a_2 \notin \mathit{dom}(h'') \qquad m' = m[x : a_2]\end{array}}{(\gamma, h, e_1\texttt{.}x \texttt{ = } e_2) \to h''[a_2 : v][a_1 : m']}$$

HeapIndexAssignment
$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \text{Record}(a_1)) \qquad (\gamma, h', e_2) \to (h'', v_1) \\ x = \text{toString}(v_1) \qquad (\gamma, h'', e_3) \to (h''', v_2) \qquad h'''(a) = m \qquad a_2 \notin \mathit{dom}(h'') \qquad m' = m[x : a_2]\end{array}}{(\gamma, h, e_1\texttt{[}e_2\texttt{]} \texttt{ = } e_3) \to h'''[a_2 : v_2][a_1 : m']}$$

IfTrue
$$\frac{(\gamma, h, e) \to (h', \text{Bool}(\mathit{True})) \qquad (\gamma, h', s_1) \to \eta}{(\gamma, h, \texttt{if } e\ s_1 \texttt{ else } s_2) \to \eta}$$

IfFalse
$$\frac{(\gamma, h, e) \to (h', \text{Bool}(\mathit{False})) \qquad (\gamma, h', s_2) \to \eta}{(\gamma, h, \texttt{if } e\ s_1 \texttt{ else } s_2) \to \eta}$$

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

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

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

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

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

Figure 1: Semantics of statements

**VarAssignment.** The VarAssignment rule evaluates an expression and assigns the resulting value to the address of the variable in the heap. The *lookup_write* function (defined below) returns the address of the frame in which to update $x$ using the appropriate scoping rules.

**HeapAssignment.** In this case, the assignment is not updating a stack frame; it is updating a record in the heap. The statement should throw an IllegalCastException if the result of the evaluation of $e_1$ is not a record.

**HeapIndexAssignment.** This rule is similar to the previous one, except the field name is not hardcoded; instead, the index expression needs to be evaluated and cast to the string according to the toString function defined below. The operation should throw an IllegalCastException if the result of evaluation $e_1$ is not a record.

**If.** If the condition evaluates to true, the result of the if statement is the result of evaluating the then branch. There is a symmetric case for when the branch condition evaluates to false. The statement should throw an IllegalCastException if the result of the evaluation of $e$ is not a boolean.

**While.** The statement should throw an IllegalCastException if the result of $e$ is not a boolean.

**Sequence.** The operator 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.** The statement is a no-op; its semantics is only incorporated later during function calls.

**Return.** The statement evaluates the expression and yields a return-value.

**Lookup.** The *lookup_write* function identifies the address of the appropriate frame into which to write. There are two cases to consider:

- $x \in g$, so $x$ is a global variable. In that case, we need to update the value of x in the global stack frame (the GlobalLookup rule).
- $x \notin g$, so $x$ is a local variable. In that case, we need to update the value of x in the local stack frame. (the LocalLookup rule)

$$\textsc{GlobalLookup}\ \frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathit{global}) = (X_g, a_2) \qquad x \in X_g}{\mathit{lookup\_write}(a_1, h, x) = a_2} \qquad \textsc{LocalLookup}\ \frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathit{global}) = (X_g, a_2) \qquad x \notin X_g}{\mathit{lookup\_write}(a_1, h, x) = a_1}$$

Figure 2: Semantics of the *lookup_write* function

# 3 Expressions

Figures 3, 4, and 5 present the semantics of expressions. The evaluation relation for expressions is of the form $(\gamma, h, e) \to (h', v)$ which denotes that given a stack $\gamma$ and a heap $h$, an expression $e$ evaluates to a new heap $h'$ and a value $v$.

## 3.1 Constants

Figure 3 presents the semantics of constants. For each type of constant value, the interpreter generates a value corresponding to the constant.

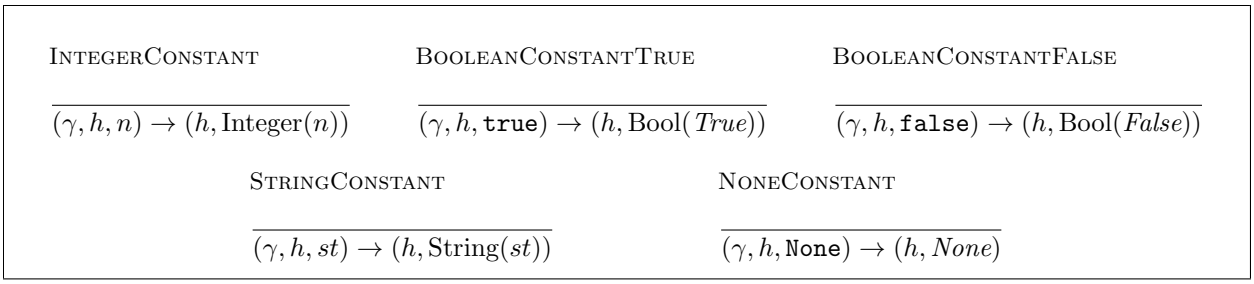


Figure 3: Constant Expressions

## 3.2 Arithmetic and Logical Expressions

Figure 4 presents the semantics of arithmetic and logical expressions. We partition the operators into several categories: logical operations, $lop \in Lop \to \ \& \ | \ |$, arithmetic operations, $op \in Op \to \ + \ | \ - \ | \ \times \ | \ /$, unary operations, $-$ and !, comparison operations, $cmp \in Cmp \to < \ | \ > \ | \ <= \ | \ >=$, and then equality ($==$). We also assume the existence of a function eval that can mathematically evaluate a given operation and its arguments.

**Logical Operations.** Logical operators can only be applied to boolean values.

**Arithmetic Operations.** The arithmetic operators '`-`' '`*`' and '`/`' can only be applied to integer values. 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 causes the non-string value to be cast to a string.

**Unary Operations.** The unary minus operator '-' can only be applied to an integer value. The unary not operator '`!`' can only be applied to a boolean value.

**Comparison Operations.** Comparison operators $<, >, <=, >=$ are only defined for integers.

**Equality Operations.** Equality between two integers, two booleans, two strings, or between two None values performs a value comparison. Equality between two Records checks for pointer equality. Equality between two Functions checks that you have the same frame address and the same function. Equality between two different types of values returns false.

**Errors.** Your interpreter should generate an IllegalCastException for operations that attempt to on types that are not given an explicit semantics. Your interpreter should generate an IllegalArithmeticException for division by zero.

## 3.3 Data and Function Expressions

Figure 5 presents the semantics of data access expressions as well as function expressions.

**Variable Read.** VariableRead rule specifies the semantics of reading from a variable. The *lookup_read* function (defined below) returns the address of $x$ using the appropriate lookup results. If the variable has not been defined in any available stack frame, then your interpreter should generate an UninitializedVariableException.

**Record Constructor** There record constructor just evaluates all its initializer expressions and constructs a Record value (a map) from fields to values. Note that the initializer expressions must be evaluated in order.

LogicalOperation

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

ArithmeticOperation

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

UnaryMinus

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

UnaryNot

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

ComparisonOperation

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

StringConcatentation

$$\frac{(\gamma, h, e_1) \to (h', \text{String}(st_1)) \qquad (\gamma, h', e_2) \to (h'', v) \qquad st_2 = \text{toString}(v)}{(\gamma, h, e_1 \texttt{ + } e_2) \to (h'', \text{String}(st_1 \ . \ st_2))}$$

NoneQuality

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

PrimitiveEquality

$$\frac{(\gamma, h, e_1) \to (h', t_1(v_1)) \qquad (\gamma, h', e_2) \to (h'', t_2(v_2)) \qquad t_1 = t_2}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \text{Bool}(v_1 == v_2))} \{t_1, t_2\} \in \{\text{Integer, Bool, String}\}$$

RecordEquality

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

FunctionEqualityTrue

$$\frac{(\gamma, h, e_1) \to (h', \text{Function}(a_1, x, s)) \qquad (\gamma, h', e_2) \to (h'', \text{Function}(a_2, x, s))}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \text{Bool}(a_1 == a_2 \wedge x_1 == x_2 \wedge s_1 == s_2))}$$

PrimitiveEqualityMismatched

$$\frac{(\gamma, h, e_1) \to (h', t_1(v_1)) \qquad (\gamma, h', e_2) \to (h'', t_2(v_2)) \qquad t_1 \neq t_2}{(\gamma, h, e_1 \texttt{ == } e_2) \to (h'', \text{Bool}(False))}$$

Figure 4: Arithmetic and Logical Expressions

**Field Read.** Evaluating the expression e produces a Record; a pointer to a map from variables to values, and the variable 'x' is mapped to the value v. If the map does not contain a field x, the value should be None. The statement should throw an IllegalCastException if the result of the evaluation of $e$ is not a record.

**Index Read.** When the index evaluates to a string, that string is used as a variable name to index into the map produced by evaluating $e_1$. If $v_1$ is not a string, then it should be converted to a string by the casting rules below. Like before, if there is no field $x$ in $h'(a)$, then Index Read should return *None*.

**Function Creation.** The construction of a function 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 must be evaluated, and it must evaluate to a function value, which consists of the address of a frame, and the code for the function.

The rule is written assuming a function with one parameter, but you should generalize it to support multiple parameters; when there are multiple parameters, you should evaluate them in order from left to right. If the caller passes a different number of parameters from what the callee expects, you should throw an error.

The next step, and the most complex one is to allocate a new stack frame for the function call.

The rule specifies this by creating new stack frame with parent pointer equal to $a$. The rule also traverses the body of the function and for every global declaration *global* $x$ it should add $x$ to the set of global variables for the new stack frame. The function *globals* (specified below) 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. The rule *assigns* (specified below) performs this traversal.

After creating the new stack frame, the rule sets all the parameters in the new stack frame to their correct values, and then evaluate the body of the function.

If $e_1$ does not evaluate to a Function value, then your implementation should return an IllegalCastException. If the caller passes too many or too few parameters to the function, then your implementation should return a RuntimeException.

# 4 Automatic Casting

Some of the rules above involve automatic casting from other values into strings. Your implementation should define a toString function that casts values to strings according to the following rules:

- toString(String($s$)) = String($s$)
- toString(Boolean($b$)) = $b$? "true" : "false"
- toString(Integer($n$)) = The string representing the integer n in base 10
- toString(Function($a, x, s$)) = "FUNCTION"
- toString($\{x_i : v_i\}$) = "{" + $x_i$ + ":" + toString($v_i$) + " " ... "}"
- toString(*None*) = "None"

VariableRead

$$\frac{\mathit{lookup\_read}(a_1, h, x) = a_2 \qquad h(a_2) = v}{(\gamma; a_1, h, x) \to (h, v)}$$

Record

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

FieldRead

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

FieldReadFail

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

IndexRead

$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \text{Record}(a_1)) \\ h'(a_1) = m \qquad (\gamma, h', e_2) \to (h'', v_1) \qquad x = \text{toString}(v_1) \qquad x \in \mathit{dom}(m) \qquad a_2 = m(x)\end{array}}{(\gamma, h, e_1\,\texttt{[}e_2\texttt{]}) \to (h'', h'(a_2))}$$

IndexReadFail

$$\frac{\begin{array}{c}(\gamma, h, e_1) \to (h', \text{Record}(a)) \\ h'(a) = m \qquad (\gamma, h', e_2) \to (h'', v_1) \qquad x = \text{toString}(v_1) \qquad x \notin \mathit{dom}(m)\end{array}}{(\gamma, h, e_1\,\texttt{[}e_2\texttt{]}) \to (h'', \mathit{None})}$$

Function

$$\frac{}{(\gamma; a, h, \texttt{fun}\ x\ s) \to (h, \text{Function}(a, x, s))}$$

FunctionCallBase

$$\frac{\begin{array}{c}(\gamma, h_1, e_1) \to (h_2, \text{Function}(a_c, x, s)) \qquad (\gamma, h_2, e_2) \to (h_3, v) \qquad a_2 \notin \mathit{dom}(h_3) \\ a_3 \notin (\mathit{dom}(h_3) \cup \{a_2\}) \qquad \sigma = \{\mathit{parent} : a_c, \mathit{global} : (\mathit{globals}(s), a_g))\}[x_i \in \mathit{assigns}(s) : a_{\mathit{None}}][x : a_2] \\ h_4 = h_3[a_2 : v][a_3 : \sigma] \qquad (a_g; \gamma; a_3, h_4, s) \to \eta\end{array}}{(a_g; \gamma; a, h_1, e_1\,\texttt{(}e_2\texttt{)}) \to \eta}$$

FunctionCallReturn

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

FunctionCallNoReturn

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

Figure 5: Data and Function Expressions

GlobalLookup

$$\frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathit{global}) = (X_g, a_2) \qquad x \in X_g \qquad h(a_2) = \sigma_2 \qquad x \in \mathit{dom}(\sigma_2) \qquad \sigma_2(x) = a_3}{\mathit{lookup\_read}(a_1, h, x) = a_3}$$

LocalLookup

$$\frac{h(a_1) = \sigma_1 \qquad \sigma_1(\mathit{global}) = (X_g, a_2) \qquad x \notin X_g \qquad x \in \mathit{dom}(\sigma_1) \qquad \sigma_1(x) = a_3}{\mathit{lookup\_read}(a_1, h, x) = a_3}$$

ScopedLookup

$$\frac{\begin{array}{c} h(a_1) = \sigma_1 \\ \sigma_1(\mathit{global}) = (X_g, a_2) \qquad x \notin X_g \qquad x \notin \mathit{dom}(\sigma_1) \qquad \mathit{lookup\_read}(\sigma(\mathit{parent}), x) = a_3 \end{array}}{\mathit{lookup\_read}(a_1, h, x) = a_3}$$

Figure 6: Semantics of the *lookup_read* function

Assign

$$\overline{\mathit{assigns}(x \texttt{ = } e) = \{x\}}$$

HeapAssignment

$$\overline{\mathit{assigns}(e_1.x \texttt{ = } e_2) = \{\}}$$

HeapIndexAsignment

$$\overline{\mathit{assigns}(e_1 \texttt{[} e_2 \texttt{]} \texttt{ = } e_3) = \{\}}$$

If

$$\overline{\mathit{assigns}(\texttt{if } e\ s_1 \texttt{ else } s_2) = \mathit{assigns}(s_1) \cup \mathit{assigns}(s_2)}$$

While

$$\overline{\mathit{assigns}(\texttt{while } e\ s) = \mathit{assigns}(s)}$$

Sequence

$$\overline{\mathit{assigns}(s_1 \texttt{ ; } s_2) = \mathit{assigns}(s_1) \cup \mathit{assigns}(s_2)}$$

Global

$$\overline{\mathit{assigns}(\texttt{global } x) = \{\}}$$

Return

$$\overline{\mathit{assigns}(\texttt{return } e) = \{\}}$$

Figure 7: Semantics of the *assigns* function

Assign

$$\overline{\mathit{globals}(x \texttt{ = } e) = \{\}}$$

HeapAssignment

$$\overline{\mathit{globals}(e_1.x \texttt{ = } e_2) = \{\}}$$

HeapIndexAsignment

$$\overline{\mathit{globals}(e_1 \texttt{[} e_2 \texttt{]} \texttt{ = } e_3) = \{\}}$$

If

$$\overline{\mathit{globals}(\texttt{if } e\ s_1 \texttt{ else } s_2) = \mathit{globals}(s_1) \cup \mathit{globals}(s_2)}$$

While

$$\overline{\mathit{globals}(\texttt{while } e\ s) = \mathit{globals}(s)}$$

Sequence

$$\overline{\mathit{globals}(s_1 \texttt{ ; } s_2) = \mathit{globals}(s_1) \cup \mathit{globals}(s_2)}$$

Global

$$\overline{\mathit{globals}(\texttt{global } x) = \{x\}}$$

Return

$$\overline{\mathit{globals}(\texttt{return } e) = \{\}}$$

Figure 8: Semantics of the *globals* function

# 5 Native functions

Your interpreter should support the following native functions:

- `print(s)` Uses the default casting of s to a string and prints it to the console followed by a newline.
- `input()` Reads a line of input from the console and returns it as a string value.
- `intcast(s)` Expects a string and internally uses the c++ function `atoi` to parse the string and return an integer value. If the string does not represent an integer (e.g., the string "hello"), the function should raise an IllegalCastException

These three 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.
```