NOTE: I am using colours in this document to ensure that character styles are applied consistently. They can be removed by changing Word’s character styles and will be removed for the final draft.

## Table of Contents

1 Scope ..................................................................................3
2 Conformance ........................................................................3
3 Normative References ...........................................................3
4 Overview .............................................................................3
5 Notational Conventions ........................................................3
   5.1 Text.............................................................................3
   5.2 Semantic Domains .........................................................3
   5.3 Tags ............................................................................4
   5.4 Booleans ......................................................................4
   5.5 Sets .............................................................................4
   5.6 Real Numbers ................................................................6
   5.6.1 Bitwise Integer Operators ........................................6
   5.7 Characters ....................................................................7
   5.8 Lists ............................................................................7
   5.9 Strings .........................................................................8
   5.10 Tuples ........................................................................9
   5.11 Records ......................................................................9
   5.12 ECMAScript Numeric Types ........................................10
   5.12.1 Signed Long Integers ..........................................10
   5.12.2 Unsigned Long Integers ........................................10
   5.12.3 Single-Precision Floating-Point Numbers ............10
   5.12.3.1 Shorthand Notation .......................................11
   5.12.3.2 Conversion ....................................................11
   5.12.3.3 Arithmetic ....................................................12
   5.12.4 Double-Precision Floating-Point Numbers ........12
   5.12.4.1 Shorthand Notation .......................................13
   5.12.4.2 Conversion ....................................................13
   5.12.4.3 Arithmetic ....................................................13
   5.13 Procedures ..................................................................15
   5.13.1 Operations ...........................................................16
   5.13.2 Semantic Domains of Procedures ......................16
   5.13.3 Steps ....................................................................16
   5.13.4 Nested Procedures ..............................................18
   5.14 Grammars ..................................................................18
   5.14.1 Grammar Notation ...............................................18
   5.14.2 Lookahead Constraints ..........................................19
   5.14.3 Line Break Constraints ..........................................19
   5.14.4 Parameterised Rules ............................................19
   5.14.5 Special Lexical Rules ..........................................20
   5.15 Semantic Actions .......................................................20
   5.15.1 Example ..............................................................21
   5.15.2 Abbreviated Actions ............................................22
   5.15.3 Action Notation Summary ..................................23
   5.16 Other Semantic Definitions ....................................24
6 Source Text ..........................................................................25
   6.1 Unicode Format-Control Characters .....................25
7 Lexical Grammar .................................................................25
   7.1 Input Elements ............................................................27
   7.2 White space ..............................................................28
   7.3 Line Breaks ..................................................................28
8 Program Structure ................................................................38
   8.1 Packages .....................................................................38
   8.2 Scopes .........................................................................38
9 Data Model ..........................................................................39
   9.1 Objects .......................................................................39
   9.1.1 Undefined ..............................................................39
   9.1.2 Null .......................................................................39
   9.1.3 Boolean ...................................................................40
   9.1.4 Number .................................................................40
   9.1.5 String ......................................................................40
   9.1.6 Namespaces ...........................................................40
   9.1.6.1 Qualified Names ............................................40
   9.1.7 Compound attributes ............................................40
   9.1.8 Classes .................................................................41
   9.1.9 Method Closures ...................................................42
   9.1.10 Prototype Instances .........................................42
   9.1.11 Class Instances ..................................................42
   9.1.11.1 Open Instances ...........................................44
   9.1.11.2 Slots ..............................................................44
   9.1.12 Packages ............................................................45
   9.1.13 Global Objects ...................................................45
   9.2 Objects with Limits ....................................................45
   9.3 References ..................................................................45
   9.4 Function Support ..........................................................46
   9.5 Argument Lists .............................................................47
   9.6 Modes of expression evaluation ..............................47
   9.7 Contexts .....................................................................47
   9.8 Labels .........................................................................47
   9.9 Environments ..............................................................48
   9.9.1 Frames ....................................................................48
   9.9.1.1 System Frame ................................................48
   9.9.1.2 Function Parameter Frames ............................48
   9.9.1.3 Block Frames ..................................................49
   9.9.2 Static Bindings ......................................................49
   9.9.3 Instance Bindings ................................................50
10 Data Operations ..................................................................51
   10.1 Numeric Utilities .......................................................51
   10.2 Object Utilities ..........................................................53
   10.2.1 objectType ...........................................................53
   10.2.2 hasType ...............................................................54
   10.2.3 toBoolean ............................................................54
   10.2.4 toGeneralNumber .............................................55
   10.2.5 toString ...............................................................55
   10.2.6 toPrimitive ..........................................................57
7.4 Comments ........................................................................28
7.5 Keywords and Identifiers ...............................................29
7.6 Punctuators ....................................................................32
7.7 Numeric literals .............................................................32
7.8 String literals ...............................................................34
7.9 Regular expression literals ..........................................37
1 Scope

This Standard defines the ECMAScript Edition 4 scripting language.

2 Conformance

3 Normative References

4 Overview

5 Notational Conventions

This specification uses the notation below to represent algorithms and concepts. These concepts are used as notation only and are not necessarily represented or visible in the ECMAScript language.

5.1 Text

Throughout this document, the phrase code point and the word character is used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of Unicode text in the UTF-16 transformation format. The phrase Unicode character is used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code point). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual Unicode characters, even though a user might think of the whole sequence as a single character.

When denoted in this specification, characters with values between 20 and 7E hexadecimal inclusive are in a fixed width font. Other characters are denoted by enclosing their four-digit hexadecimal Unicode value between «u and ». For example, the non-breakable space character would be denoted in this document as «u00A0». A few of the common control characters are represented by name:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>«NUL»</td>
<td>«u0000»</td>
</tr>
<tr>
<td>«BS»</td>
<td>«u0008»</td>
</tr>
<tr>
<td>«TAB»</td>
<td>«u0009»</td>
</tr>
<tr>
<td>«LF»</td>
<td>«u000A»</td>
</tr>
<tr>
<td>«VT»</td>
<td>«u000B»</td>
</tr>
<tr>
<td>«FF»</td>
<td>«u000C»</td>
</tr>
<tr>
<td>«CR»</td>
<td>«u000D»</td>
</tr>
<tr>
<td>«SP»</td>
<td>«u0020»</td>
</tr>
</tbody>
</table>

A space character is denoted in this document either by a blank space where it’s obvious from the context or by «SP» where the space might be confused with some other notation.

5.2 Semantic Domains

Semantic domains describe the possible values that a variable might take on in an algorithm. The algorithms are constructed in a way that ensures that these constraints are always met, regardless of any valid or invalid programmer or user input or actions.
A semantic domain can be intuitively thought of as a set of possible values, and, in fact, any set of values explicitly described in this document is also a semantic domain. Nevertheless, semantic domains have a more precise mathematical definition in domain theory (see for example David Schmidt, Denotational Semantics: A Methodology for Language Development; Allyn and Bacon 1986) that allows one to define semantic domains recursively without encountering paradoxes such as trying to define a set $A$ whose members include all functions mapping values from $A$ to INTEGER. The problem with an ordinary definition of such a set $A$ is that the cardinality of the set of all functions mapping $A$ to INTEGER is always strictly greater than the cardinality of $A$, leading to a contradiction. Domain theory uses a least fixed point construction to allow $A$ to be defined as a semantic domain without encountering problems.

Semantic domains have names in **CAPITALISED SMALL CAPS**. Such a name is to be considered distinct from a tag or regular variable with the same name, so **UNDEFINED, undefined**, and **undefined** are three different and independent entities.

A variable $v$ is constrained using the notation

$$v : T$$

where $T$ is a semantic domain. This constraint indicates that the value of $v$ will always be a member of the semantic domain $T$. These declarations are informative (they may be dropped without affecting the semantics’ correctness) but useful in understanding the semantics. For example, when the semantics state that $x : INTEGER$ then one does not have to worry about what happens when $x$ has the value $true$ or $+\infty$.

The constraints can be proven statically. The semantics have been machine-checked to ensure that every constraint holds.

### 5.3 Tags

Tags are computational tokens with no internal structure. Tags are written using a **bold sans-serif font**. Two tags are equal if and only if they have the same name. Examples of tags include $true$, $false$, $null$, $NaN$, and identifier.

### 5.4 Booleans

The tags $true$ and $false$ represent **Booleans**. **BOOLEAN** is the two-element semantic domain $\{true, false\}$.

Let $a$ and $b$ be Booleans. In addition to $=$ and $\neq$, the following operations can be done on them:

- $not$ $a$
  - true if $a$ is false, false if $a$ is true
- $a$ $and$ $b$
  - If $a$ is false, returns false without computing $b$; if $a$ is true, returns the value of $b$
- $a$ $or$ $b$
  - If $a$ is false, returns the value of $b$; if $a$ is true, returns true without computing $b$
- $a$ $xor$ $b$
  - true if $a$ is true and $b$ is false or $a$ is false and $b$ is true; false otherwise. $a$ $xor$ $b$ is equivalent to $a \neq b$

Note that the $and$ and $or$ operators short-circuit. These are the only operators that do not always compute all of their operands.

### 5.5 Sets

A set is an unordered, possibly infinite collection of elements. Each element may occur at most once in a set. There must be an equivalence relation $=$ defined on all pairs of the set’s elements. Elements of a set may themselves be sets.

A set is denoted by enclosing a comma-separated list of values inside braces:

$$\{element_1, element_2, ..., element_n\}$$

The empty set is written as $\{\}$. Any duplicate elements are included only once in the set.

For example, the set $\{3, 0, 10, 11, 12, 13, -5\}$ contains seven integers.

Sets of either integers or characters can be abbreviated using the $...$ range operator. For example, the above set can also be written as $\{0, -5, 3 ... 3, 10 ... 13\}$. 
If the beginning of the range is equal to the end of the range, then the range consists of only one element: \{7 \ldots 7\} is the same as \{7\}. If the end of the range is one less than the beginning, then the range contains no elements: \{7 \ldots 6\} is the same as \{}. The end of the range is never more than one less than the beginning.

A set can also be written using the set comprehension notation

\[ \{f(x) \mid x \in A\} \]

which denotes the set of the results of computing expression \(f\) on all elements \(x\) of set \(A\). A predicate can be added:

\[ \{f(x) \mid x \in A \text{ such that } \text{predicate}(x)\} \]

denotes the set of the results of computing expression \(f\) on all elements \(x\) of set \(A\) that satisfy the \text{predicate} expression. There can also be more than one free variable \(x\) and set \(A\), in which case all combinations of free variables’ values are considered. For example,

\[ \{x \mid x \in \text{INTEGER such that } x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\} \]
\[ \{x^2 \mid x \in \{-5, -1, 1, 2, 4\}\} = \{1, 4, 16, 25\} \]
\[ \{x \mid 10 + y \mid x \in \{1, 2, 4\}, y \in \{3, 5\}\} = \{13, 15, 23, 25, 43, 45\} \]

The same notation is used for operations on sets and on semantic domains. Let \(A\) and \(B\) be sets (or semantic domains) and \(x\) and \(y\) be values. The following operations can be done on them:

- \(x \in A \) \hspace{1cm} \text{true} \hspace{1cm} \text{if} \hspace{1cm} \text{\(x\) is an element of \(A\)} \hspace{1cm} \text{and} \hspace{1cm} \text{false} \hspace{1cm} \text{if not} \hspace{1cm} \text{\(x\)}
- \(x \notin A \) \hspace{1cm} \text{false} \hspace{1cm} \text{if} \hspace{1cm} \text{\(x\) is an element of \(A\)} \hspace{1cm} \text{and} \hspace{1cm} \text{true} \hspace{1cm} \text{if not} \hspace{1cm} \text{\(x\)}
- \(|A|\) \hspace{1cm} \text{The number of elements in \(A\)} \hspace{1cm} \text{(only used on finite sets)}
- \(\text{min } A\) \hspace{1cm} \text{The value \(m\) that satisfies both} \hspace{1cm} m \not\in A \hspace{1cm} \text{and for all} \hspace{1cm} x \in A, x \geq m \hspace{1cm} \text{(only used on nonempty, finite sets whose elements have a well-defined order relation)}
- \(\text{max } A\) \hspace{1cm} \text{The value \(m\) that satisfies both} \hspace{1cm} m \not\in A \hspace{1cm} \text{and for all} \hspace{1cm} x \in A, x \leq m \hspace{1cm} \text{(only used on nonempty, finite sets whose elements have a well-defined order relation)}
- \(A \sqcap B\) \hspace{1cm} \text{The intersection of} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{(the set or semantic domain of all values that are present both in} \hspace{1cm} A \hspace{1cm} \text{and in} \hspace{1cm} B\)}
- \(A \sqcup B\) \hspace{1cm} \text{The union of} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{(the set or semantic domain of all values that are present in at least one of} \hspace{1cm} A \hspace{1cm} \text{or} \hspace{1cm} B\)}
- \(A - B\) \hspace{1cm} \text{The difference of} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{(the set or semantic domain of all values that are present in} \hspace{1cm} A \hspace{1cm} \text{but not} \hspace{1cm} B\)}
- \(A = B\) \hspace{1cm} \text{true} \hspace{1cm} \text{if} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{are equal and} \hspace{1cm} \text{false} \hspace{1cm} \text{otherwise.} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{are equal if every element of} \hspace{1cm} A \hspace{1cm} \text{is also in} \hspace{1cm} B \hspace{1cm} \text{and every element of} \hspace{1cm} B \hspace{1cm} \text{is also in} \hspace{1cm} A\).
- \(A \neq B\) \hspace{1cm} \text{false} \hspace{1cm} \text{if} \hspace{1cm} A \hspace{1cm} \text{and} \hspace{1cm} B \hspace{1cm} \text{are equal and} \hspace{1cm} \text{true} \hspace{1cm} \text{otherwise}
- \(A \subseteq B\) \hspace{1cm} \text{true} \hspace{1cm} \text{if} \hspace{1cm} A \hspace{1cm} \text{is a subset of} \hspace{1cm} B \hspace{1cm} \text{and} \hspace{1cm} \text{false} \hspace{1cm} \text{otherwise.} \hspace{1cm} A \hspace{1cm} \text{is a subset of} \hspace{1cm} B \hspace{1cm} \text{if every element of} \hspace{1cm} A \hspace{1cm} \text{is also in} \hspace{1cm} B \hspace{1cm} \text{and every element of} \hspace{1cm} B \hspace{1cm} \text{is also in} \hspace{1cm} A\).
- \(A \supset B\) \hspace{1cm} \text{true} \hspace{1cm} \text{if} \hspace{1cm} A \hspace{1cm} \text{is a proper subset of} \hspace{1cm} B \hspace{1cm} \text{and} \hspace{1cm} \text{false} \hspace{1cm} \text{otherwise.} \hspace{1cm} A \subseteq B \hspace{1cm} \text{is equivalent to} \hspace{1cm} A \supset B \hspace{1cm} \text{and} \hspace{1cm} A \neq B\).

If \(T\) is a semantic domain, then \(T\{\}\) is the semantic domain of all sets whose elements are members of \(T\). For example, if \(T = \{1, 2, 3\}\) then:

\[ T\{\} = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\} \]

The empty set \(\{\}\) is a member of \(T\{\}\) for any semantic domain \(T\).

In addition to the above, the \text{some} and \text{every} quantifiers can be used on sets. The quantifier

\[ \text{some } x \in A \text{ satisfies } \text{predicate}(x) \]

returns \text{true} if there exists at least one element \(x\) in set \(A\) such that \(\text{predicate}(x)\) computes to \text{true}. If there is no such element \(x\), then the \text{some} quantifier’s result is \text{false}. If the \text{some} quantifier returns \text{true}, then variable \(x\) is left bound to any element of \(A\) for which \(\text{predicate}(x)\) computes to \text{true}; if there is more than one such element \(x\), then one of them is chosen arbitrarily. For example,

\[ \text{some } x \in \{3, 16, 19, 26\} \text{ satisfies } x \mod 10 = 6 \]

evaluates to \text{true} and leaves \(x\) set to either 16 or 26. Other examples include:
The unique sequence corresponding to an integer in infinite-precision two's complement binary notation, with each 1 bit representing being cascaded, so Real example of a real number slightly larger than 3.14.

(\text{some } x \notnull \{3, 16, 19, 26\} \text{ satisfies } x \text{ mod } 10 = 7) = \mathbf{false};
(\text{some } x \notnull \{\} \text{ satisfies } x \text{ mod } 10 = 7) = \mathbf{false};
(\text{some } x \notnull \{\"Hello\}\} \text{ satisfies } \mathbf{true}) = \mathbf{true} \text{ and leaves } x \text{ set to the string \"Hello\"};
(\text{some } x \notnull \{\} \text{ satisfies } \mathbf{true}) = \mathbf{false}.

The quantifier
\text{every } x \notnull A \text{ satisfies } \mathbf{predicate}(x)
returns \mathbf{true} if there exists no element \( x \) in set \( A \) such that \( \mathbf{predicate}(x) \) computes to \( \mathbf{false} \). If there is at least one such element \( x \), then the \text{every} quantifier’s result is \( \mathbf{false} \). As a degenerate case, the \text{every} quantifier is always \( \mathbf{true} \) if the set \( A \) is empty. For example,
(\text{every } x \notnull \{3, 16, 19, 26\} \text{ satisfies } x \text{ mod } 10 = 6) = \mathbf{false};
(\text{every } x \notnull \{6, 26, 96, 106\} \text{ satisfies } x \text{ mod } 10 = 6) = \mathbf{true};
(\text{every } x \notnull \{\} \text{ satisfies } x \text{ mod } 10 = 6) = \mathbf{true}.

5.6 Real Numbers

Numbers written in this specification are to be understood to be exact mathematical real numbers, which include integers and rational numbers as subsets. Examples of numbers include -3, 0, 17, 10^{1000}, and \[\]. Hexadecimal numbers are written by preceding them with “0x”, so 4294967296, 0x100000000, and 2^{32} are all the same integer.

\text{INTEGER} is the semantic domain of all integers \{... –3, –2, –1, 0, 1, 2, 3 ...\}. 3.0, 3, 0xFF, and –10^{100} are all integers.

\text{RATIONAL} is the semantic domain of all rational numbers. Every integer is also a rational number: \text{INTEGER} \notnull \text{RATIONAL}. 3, 1/3, 7.5, –12/7, and 2^5 are examples of rational numbers.

\text{REAL} is the semantic domain of all real numbers. Every rational number is also a real number: \text{RATIONAL} \notnull \text{REAL}. \[] is an example of a real number slightly larger than 3.14.

Let \( x \) and \( y \) be real numbers. The following operations can be done on them and always produce exact results:

\begin{align*}
\text{Negation:} & \quad \neg x \\
\text{Sum:} & \quad x + y \\
\text{Difference:} & \quad x - y \\
\text{Product:} & \quad x \cdot y \\
\text{Quotient (y must not be zero):} & \quad x / y \\
\text{Exponentiation (y must be an integer):} & \quad x^y \\
\text{The absolute value of x, which is x if x\geq0 and -x otherwise:} & \quad |x| \\
\text{Floor of x, which is the unique integer i such that i\leq x < i+1:} & \quad \lfloor x \rfloor = 3, \lfloor -3.5 \rfloor = -4, \text{ and } \lceil 7 \rceil = 7. \\
\text{Ceiling of x, which is the unique integer i such that i-1 < x \leq i:} & \quad \lceil x \rceil = 4, \lceil -3.5 \rceil = -3, \text{ and } \lfloor 7 \rfloor = 7. \\
\text{x modulo y, which is defined as x - y\lfloor x / y \rfloor (y must not be zero):} & \quad 10 \mod 7 = 3, \text{ and } -1 \mod 7 = 6.
\end{align*}

Real numbers can be compared using =, \neq, <, \leq, >, and \geq. The result is either \mathbf{true} or \mathbf{false}. Multiple relational operators can be cascaded, so \( x < y < z \) is \mathbf{true} only if both \( x \) is less than \( y \) and \( y \) is less than \( z \).

5.6.1 Bitwise Integer Operators

The four procedures below perform bitwise operations on integers. The integers are treated as though they were written in infinite-precision two’s complement binary notation, with each 1 bit representing \mathbf{true} and 0 bit representing \mathbf{false}.

More precisely, any integer \( x \) can be represented as an infinite sequence of bits \( a_i \), where the index \( i \) ranges over the nonnegative integers and every \( a_i \notnull \{0, 1\} \). The sequence is traditionally written in reverse order:
\(..., a_4, a_3, a_2, a_1, a_0\)

The unique sequence corresponding to an integer \( x \) is generated by the formula

The results are listed in the same order as the elements denoted by the corresponding element of list which generates the sequence $c_i$. The result is another infinite sequence of bits $c_i$. The result of the operation is the unique integer $z$ that generates the sequence $c_i$. For example, ANDing corresponding elements of the sequences generated by 6 and $-6$ yields the sequence ...0000010, which is the sequence generated by the integer 2. Thus, $\text{bitwiseAnd}(6, -6) = 2$.

### 5.7 Characters

Characters enclosed in single quotes ‘ and ’ represent single Unicode 16-bit code points. Examples of characters include ‘A’, ‘b’, ‘«LF»’, and ‘«uFFFF»’ (see also section 5.1). Unicode surrogates are considered to be pairs of characters for the purpose of this specification.

**Character** is the semantic domain of all 65536 characters {‘«u0000» ... ‘«uFFFF»’}. Characters can be compared using =, ≠, <, ≤, >, and ≥. These operators compare code point values, so ‘A’ = ‘A’, ‘A’ < ‘B’, and ‘a’ < ‘a’ are all true.

The procedures `characterToCode` and `codeToCharacter` convert between characters and their integer Unicode values.

- `characterToCode(c: CHARACTER): [0 ... 65535]` Return character c’s Unicode code point as an integer
- `codeToCharacter(i: [0 ... 65535]): CHARACTER` Return the character whose Unicode code point is i

### 5.8 Lists

A finite ordered list of zero or more elements is written by listing the elements inside bold brackets:

```
[element₀, element₁, ..., elementₙ]
```

For example, the following list contains four strings:

```
[“parsley”, “sage”, “rosemary”, “thyme”]
```

The empty list is written as $. Unlike a set, the elements of a list are indexed by integers starting from 0. A list can contain duplicate elements.

A list can also be written using the list comprehension notation

```
[f(x) | x ∈ u]
```

which denotes the list $[f(u[0]), f(u[1]), ..., f(u[|u|−1])]$ whose elements consist of the results of applying expression $f$ to each corresponding element of list $u$. $x$ is the name of the parameter in expression $f$. A predicate can be added:

```
[f(x) | x ∈ u such that predicate(x)]
```

denotes the list of the results of computing expression $f$ on all elements $x$ of list $u$ that satisfy the $\text{predicate}$ expression. The results are listed in the same order as the elements $x$ of list $u$. For example,
In The empty string is usually written as "".

5.9 Strings

A list of characters is called a string. In addition to the normal list notation, for notational convenience a string can also be written as zero or more characters enclosed in double quotes (see also the notation for non-ASCII characters). Thus,

"Wonder\r\n"

is equivalent to:

['W', 'o', ' ', 'n', ' ', 'e', ' ', 'r', ' ', '\n']

The empty string is usually written as "".

In addition to the other list operations, <, ≤, >, and ≥ are defined on strings. A string x is less than string y when y is not the empty string and either x is the empty string, the first character of x is less than the first character of y, or the first character of x is equal to the first character of y and the rest of string x is less than the rest of string y.

STRING is the semantic domain of all strings, STRING = CHARACTER[].
5.10 Tuples

A tuple is an immutable aggregate of values comprised of a name `NAME` and zero or more labelled fields.

The fields of each kind of tuple used in this specification are described in tables such as:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>label₁</td>
<td>T₁</td>
<td>Informative note about this field</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>labelₙ</td>
<td>Tₙ</td>
<td>Informative note about this field</td>
</tr>
</tbody>
</table>

The notation

\[ \text{NAME}[\text{label₁}: v₁, \ldots, \text{labelₙ}: vₙ] \]

represents a tuple with name `NAME` and values `v₁` through `vₙ` for fields labelled `label₁` through `labelₙ` respectively. Each value `vᵢ` is a member of the corresponding semantic domain `Tᵢ`. When most of the fields are copied from an existing tuple `a`, this notation can be abbreviated as

\[ \text{NAME}[\text{labelᵢ}: vᵢ₁, \ldots, \text{labelₙ}: vₙ, \text{other fields from } a] \]

which represents a tuple with name `NAME` and values `vᵢ₁` through `vₙ` for fields labelled `labelᵢ` through `labelₙ` respectively and the values of correspondingly labelled fields from `a` for all other fields.

If `a` is the tuple `\text{NAME}[\text{label₁}: v₁, \ldots, \text{labelₙ}: vₙ]` then

\[ a.\text{labelᵢ} \]

returns the `ᵢ`<sup>th</sup> field’s value `vᵢ`.

The equality operators `=` and `≠` may be used to compare tuples. Tuples are equal when they have the same name and their corresponding field values are equal.

When used in an expression, the tuple’s name `NAME` itself represents the semantic domain of all tuples with name `NAME`.

5.11 Records

A record is a mutable aggregate of values similar to a tuple but with different equality behaviour.

A record is comprised of a name `NAME` and an `address`. The address points to a mutable data structure comprised of zero or more labelled fields. The address acts as the record’s serial number — every record allocated by `new` (see below) gets a different address, including records created by identical expressions or even the same expression used twice.

The fields of each kind of record used in this specification are described in tables such as:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>label₁</td>
<td>T₁</td>
<td>Informative note about this field</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>labelₙ</td>
<td>Tₙ</td>
<td>Informative note about this field</td>
</tr>
</tbody>
</table>

The expression

\[ \text{new NAME}[\text{label₁}: v₁, \ldots, \text{labelₙ}: vₙ] \]

creates a record with name `NAME` and a new address `[]`. The fields labelled `label₁` through `labelₙ` at address `[]` are initialised with values `v₁` through `vₙ` respectively. Each value `vᵢ` is a member of the corresponding semantic domain `Tᵢ`. A `labelᵢ; vᵢ` pair may be omitted from a `new` expression, which indicates that the initial value of field `labelᵢ` does not matter because the semantics will always explicitly write a value into that field before reading it.
When most of the fields are copied from an existing record \( a \), the \texttt{new} expression can be abbreviated as
\[
\text{new } \texttt{NAME}[] \texttt{label}_i: v_i, \ldots, \texttt{label}_k: v_k, \text{ other fields from } a[]
\]
which represents a record \( b \) with name \texttt{NAME} and a new address \([\texttt{label}_i]\). The fields labeled \texttt{label}_j through \texttt{label}_k at address \([\texttt{label}_i]\) are initialised with values \( v_j \) through \( v_k \) respectively; the other fields at address \([\texttt{label}_i]\) are initialised with the values of correspondingly labeled fields from \( a \)'s address.

If \( a \) is a record with name \texttt{NAME} and address \([\texttt{label}_i]\), then
\[
a.\texttt{label}_i
\]
returns the current value \( v \) of the \( i \)th field at address \([\texttt{label}_i]\). That field may be set to a new value \( w \), which must be a member of the semantic domain \( T_i \), using the assignment
\[
a.\texttt{label}_i \leftarrow w
\]
after which \( a.\texttt{label}_i \) will evaluate to \( w \). Any record with a different address \([\texttt{label}_i]\) is unaffected by the assignment.

The equality operators \( = \) and \( \neq \) may be used to compare records. Records are equal only when they have the same address.

When used in an expression, the record’s name \texttt{NAME} itself represents the semantic domain of all records with name \texttt{NAME}.

### 5.12 ECMAScript Numeric Types

ECMAScript does not support exact real numbers as one of the programmer-visible data types. Instead, ECMAScript numbers have finite range and precision. The semantic domain of all programmer-visible numbers representable in ECMAScript is \texttt{GENERALNUMBER}, defined as the union of four basic numeric semantic domains \texttt{LONG}, \texttt{ULONG}, \texttt{FLOAT32}, and \texttt{FLOAT64}:

\[
\texttt{GENERALNUMBER} = \texttt{LONG} [] \texttt{ULONG} [] \texttt{FLOAT32} [] \texttt{FLOAT64}
\]

The four basic numeric semantic domains are all disjoint from each other and from the semantic domains \texttt{INTEGER}, \texttt{RATIONAL}, and \texttt{REAL}.

The semantic domain \texttt{FINITEGENERALNUMBER} is the subtype of all finite values in \texttt{GENERALNUMBER}:

\[
\texttt{FINITEGENERALNUMBER} = \texttt{LONG} [] \texttt{ULONG} [] \texttt{FLOAT32} [] \texttt{FLOAT64}
\]

#### 5.12.1 Signed Long Integers

Programmer-visible signed 64-bit long integers are represented by the semantic domain \texttt{LONG}. These are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains \texttt{ULONG}, \texttt{FLOAT32}, and \texttt{FLOAT64}. A \texttt{LONG} tuple has the field below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{value}</td>
<td>{(-2^{63} \ldots 2^{63} - 1)}</td>
<td>The signed 64-bit integer</td>
</tr>
</tbody>
</table>

#### 5.12.2 Unsigned Long Integers

Programmer-visible unsigned 64-bit long integers are represented by the semantic domain \texttt{ULONG}. These are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains \texttt{LONG}, \texttt{FLOAT32}, and \texttt{FLOAT64}. A \texttt{ULONG} tuple has the field below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{value}</td>
<td>{0 \ldots 2^{64} - 1}</td>
<td>The unsigned 64-bit integer</td>
</tr>
</tbody>
</table>

#### 5.12.3 Single-Precision Floating-Point Numbers

\texttt{FLOAT32} is the semantic domain of all representable single-precision floating-point IEEE 754 values, with all not-a-number values considered indistinguishable from each other. \texttt{FLOAT32} is the union of the following semantic domains:
The procedure
\[
\text{FLOAT32} = \text{FINITEFLOAT32} \equiv \{ +\infty_{\text{f32}}, -\infty_{\text{f32}}, \text{NaN}_{\text{f32}} \};
\]
\[
\text{FINITEFLOAT32} = \text{NONZEROFINITEFLOAT32} \equiv \{ +\text{zero}_{\text{f32}}, -\text{zero}_{\text{f32}} \}
\]
The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains LONG, ULONG, and FLOAT64. A NONZEROFINITEFLOAT32 tuple has the field below:

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>NORMALISEDFLOAT32VALUES \sqcup \DENORMALISEDFLOAT32VALUES</td>
<td>The value, represented as an exact rational number</td>
</tr>
</tbody>
</table>

There are 4261412864 (that is, $2^{24} - 2^{25}$) normalised values:

\[
\text{NORMALISEDFLOAT32VALUES} = \{ s \} m \{2^{23} \ldots 2^{24}-1\}, \square c \{ -149 \ldots 104 \}
\]

$m$ is called the significand.

There are also 16777214 (that is, $2^{24} - 2$) denormalised non-zero values:

\[
\text{DENORMALISEDFLOAT32VALUES} = \{ s \} m \{2^{-149} \ldots 1\}, \square m \{ 1 \ldots 2^{23} - 1 \}
\]

$m$ is called the significand.

The remaining FLOAT32 values are the tags $+\text{zero}_{\text{f32}}$ (positive zero), $-\text{zero}_{\text{f32}}$ (negative zero), $+\infty_{\text{f32}}$ (positive infinity), $-\infty_{\text{f32}}$ (negative infinity), and NaN$_{\text{f32}}$ (not a number).

Members of the semantic domain NONZEROFINITEFLOAT32 with value greater than zero are called positive finite. The remaining members of NONZEROFINITEFLOAT32 are called negative finite.

Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation $=$ and $\neq$ may be used to compare them. Note that $=$ is false for different tags, so $+\text{zero}_{\text{f32}} \neq -\text{zero}_{\text{f32}}$ but NaN$_{\text{f32}} = \text{NaN}_{\text{f32}}$. The ECMAScript $x == y$ and $x === y$ operators have different behavior for FLOAT32 values, defined by isEqual and isStrictEqual.

5.12.3.1 Shorthand Notation

In this specification, when $x$ is a real number or expression, the notation $x_{\text{f32}}$ indicates the result of realToFloat32($x$), which is the “closest” FLOAT32 value as defined below. Thus, 3.4 is a REAL number, while 3.4$_{\text{f32}}$ is a FLOAT32 value (whose exact value is actually 3.4000000095367431640625). The positive finite FLOAT32 values range from $10^{45}_{\text{f32}}$ to $(3.4028235 \ldots 10^{38})_{\text{f32}}$.

5.12.3.2 Conversion

The procedure realToFloat32 converts a real number $x$ into the applicable element of FLOAT32 as follows:

\[
\text{proc} \quad \text{realToFloat32}(x; \text{REAL}); \text{FLOAT32}
\]
\[
s; \text{RATIONAL} \square \sqcup \text{NORMALISEDFLOAT32VALUES} \square \sqcup \text{DENORMALISEDFLOAT32VALUES} \square \{ -2^{128}, 0, 2^{128} \};
\]

Let $a; \text{RATIONAL}$ be the element of $s$ closest to $x$ (i.e. such that $|a - x|$ is as small as possible). If two elements of $s$ are equally close, let $a$ be the one with an even significand; for this purpose $-2^{128}, 0$, and $2^{128}$ are considered to have even significands.

if $a = 2^{128}$ then return $+\infty_{\text{f32}}$
elsif $a = -2^{128}$ then return $-\infty_{\text{f32}}$
elsif $a \neq 0$ then return NONZEROFINITEFLOAT32$value; a$
elsif $x < 0$ then return $-\text{zero}_{\text{f32}}$
else return $+\text{zero}_{\text{f32}}$
end if

NOTE: This procedure corresponds exactly to the behaviour of the IEEE 754 "round to nearest" mode.

The procedure truncateFiniteFloat32 truncates a FINITEFLOAT32 value to an integer, rounding towards zero:

\[
\text{proc} \quad \text{truncateFiniteFloat32}(x; \text{FINITEFLOAT32}); \text{INTEGER}
\]
\[
\text{if } x \square \{ +\text{zero}_{\text{f32}}, -\text{zero}_{\text{f32}}\} \text{ then return } 0 \text{ end if;}
\]
\[
r; \text{RATIONAL} \sqcup x.\text{value};
\]
\[
\text{if } r > 0 \text{ then return } \square \text{ else return } \square \end if
\end proc
5.12.3.3 Arithmetic

The following tables define procedures that perform common arithmetic operations on `FLOAT32` values using IEEE 754 rules. Note that `(expr)_{32}` is a shorthand for `realToFloat32(expr)`.

<p>| <code>float32Abs(x: FLOAT32): FLOAT32</code> |</p>
<table>
<thead>
<tr>
<th align="left"><code>x</code></th>
<th align="left">Result</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left"><code>+\infty_{32}</code></td>
<td align="left"><code>+\infty_{32}</code></td>
</tr>
<tr>
<td align="left">negative finite</td>
<td align="left"><code>(-x).value\_32</code></td>
</tr>
<tr>
<td align="left"><code>+\text{zero}_{32}</code></td>
<td align="left"><code>+\text{zero}_{32}</code></td>
</tr>
<tr>
<td align="left"><code>+\text{positive finite}</code></td>
<td align="left"><code>(-x).value\_32</code></td>
</tr>
<tr>
<td align="left"><code>+\infty_{32}</code></td>
<td align="left"><code>+\infty_{32}</code></td>
</tr>
<tr>
<td align="left"><code>NaN_{32}</code></td>
<td align="left"><code>NaN_{32}</code></td>
</tr>
</tbody>
</table>

The identity for floating-point addition is `+\text{zero}`, not `+\text{zero}`.

| `float32Subtract(x: FLOAT32, y: FLOAT32): FLOAT32` |
| `float32Multiply(x: FLOAT32, y: FLOAT32): FLOAT32` |
| `float32Divide(x: FLOAT32, y: FLOAT32): FLOAT32` |
| `float32Remainder(x: FLOAT32, y: FLOAT32): FLOAT32` |

Note that `float32Remainder(float32Negate(x), y)` always produces the same result as `float32Negate(float32Remainder(x, y))`. Also, `float32Remainder(x, float32Negate(y))` always produces the same result as `float32Remainder(x, y)`.

5.12.4 Double-Precision Floating-Point Numbers

`FLOAT64` is the semantic domain of all representable double-precision floating-point IEEE 754 values, with all not-a-number values considered indistinguishable from each other. `FLOAT64` is the union of the following semantic domains:

| `FLOAT64 = \text{FiniteFloat64} \uplus \{ +\infty_{64}, -\infty_{64}, NaN_{64}\};` |
| `FiniteFloat64 = \text{NonZeroFiniteFloat64} \uplus \{ +\text{zero}_{64}, -\text{zero}_{64}\}` |

The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains `LONG`, `ULONG`, and `FLOAT32`. A `NonZeroFiniteFloat64` tuple has the field below:

<table>
<thead>
<tr>
<th align="left">Field</th>
<th align="left">Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left"><code>value</code></td>
<td align="left"><code>NormalisedFloat64Values \uplus DenormalisedFloat64Values</code></td>
</tr>
</tbody>
</table>

The value, represented as an exact rational number

There are 18428729675200069632 (that is, $2^{64} \cdot 2^{53}$) normalised values:

| NormalisedFloat64Values | $\{ s \in \{ 0, 1 \}, \, \lfloor m \rfloor \in \{ 2 \cdot 2^{52} \ldots 2^{53-1} \}, \, \lfloor e \rfloor \in \{ -1074 \ldots 971 \} \}$ |

`m` is called the significand.

There are also 9007199254740990 (that is, $2^{53} - 2$) denormalised non-zero values:

| DenormalisedFloat64Values | $\{ s \in \{ 0, 1 \}, \lfloor m \rfloor \in \{ 2 \cdot 2^{52} \ldots 2^{53-1} \} \}$ |

`m` is called the significand.

The remaining `FLOAT64` values are the tags `+\text{zero}_{64}` (positive zero), `-\text{zero}_{64}` (negative zero), `+\infty_{64}` (positive infinity), `-\infty_{64}` (negative infinity), and `NaN_{64}` (not a number).

Members of the semantic domain `NonZeroFiniteFloat64` with `value` greater than zero are called positive finite. The remaining members of `NonZeroFiniteFloat64` are called negative finite.
Since floating-point numbers are either tags or tuples wrapping rational numbers, the notation \(=\) and \(\neq\) may be used to compare them. Note that \(=\) is false for different tags, so \(+\text{zero}_{64}\neq-\text{zero}_{64}\) but \(\text{NaN}_{64}=\text{NaN}_{64}\). The ECMAscript \(x==y\) and \(x===y\) operators have different behavior for \text{FLOAT64} values, defined by \text{isEqual} and \text{isStrictlyEqual}.

5.12.4.1 Shorthand Notation

In this specification, when \(x\) is a real number or expression, the notation \(x_{64}\) indicates the result of \text{realToFloat64}(x), which is the “closest” \text{FLOAT64} value as defined below. Thus, \(3.4\) is a \text{REAL} number, while \(3.4_{64}\) is a \text{FLOAT64} value (whose exact value is actually \(3.399999999999991182158029987476766109466552734375\)). The positive finite \text{FLOAT64} values range from \((5 \cdot 10^{-324})_{64}\) to \((1.7976931348623157 \cdot 10^{308})_{64}\).

5.12.4.2 Conversion

The procedure \text{realToFloat64} converts a real number \(x\) into the applicable element of \text{FLOAT64} as follows:

\[
\text{proc realToFloat64}(x: \text{REAL}) : \text{FLOAT64}
\]

\[
s: \text{RATIONAL} \{\} \rightarrow \text{NORMALISEDFLOAT64VALUES} \rightarrow \text{DENORMALISEDFLOAT64VALUES} \rightarrow \{-2^{1024}, 0, 2^{1024}\};
\]

Let \(a: \text{RATIONAL}\) be the element of \(s\) closest to \(x\) (i.e. such that \(|a-x|\) is as small as possible). If two elements of \(s\) are equally close, let \(a\) be the one with an even significand; for this purpose \(-2^{1024}, 0, 2^{1024}\) are considered to have even significands.

\[
\text{if } a = 2^{1024} \text{ then return } +\text{oo}_{64}
\]

\[
\text{elsif } a = -2^{1024} \text{ then return } -\text{oo}_{64}
\]

\[
\text{elsif } a \neq 0 \text{ then return } \text{NONZEROFINITEFLOAT64}{\text{value}: a}
\]

\[
\text{elsif } x < 0 \text{ then return } -\text{zero}_{64}
\]

\[
\text{else return } +\text{zero}_{64}
\]

\[
\text{end if}
\]

\[
\text{end proc}
\]

NOTE  This procedure corresponds exactly to the behaviour of the IEEE 754 "round to nearest" mode.

The procedure \text{float32ToFloat64} converts a \text{FLOAT32} number \(x\) into the corresponding \text{FLOAT64} number as defined by the following table:

\[
\text{float32ToFloat64}(x: \text{FLOAT32}) : \text{FLOAT64}
\]

<table>
<thead>
<tr>
<th>(x)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\text{oo}_{32})</td>
<td>(-\text{oo}_{64})</td>
</tr>
<tr>
<td>(-\text{zero}_{32})</td>
<td>(-\text{zero}_{64})</td>
</tr>
<tr>
<td>+\text{zero}_{32}</td>
<td>+\text{zero}_{64}</td>
</tr>
<tr>
<td>+\text{oo}_{32}</td>
<td>+\text{oo}_{64}</td>
</tr>
<tr>
<td>\text{NaN}_{32}</td>
<td>\text{NaN}_{64}</td>
</tr>
<tr>
<td>Any \text{NONZEROFINITEFLOAT32} value</td>
<td>\text{NONZEROFINITEFLOAT64}{\text{value}: x.value}</td>
</tr>
</tbody>
</table>

The procedure \text{truncateFiniteFloat64} truncates a \text{FINITEFLOAT64} value to an integer, rounding towards zero:

\[
\text{proc truncateFiniteFloat64}(x: \text{FINITEFLOAT64}) : \text{INTEGER}
\]

\[
\text{if } x \{+\text{zero}_{64}, -\text{zero}_{64}\} \text{ then return } 0 \text{ end if};
\]

\[
r: \text{RATIONAL} \rightarrow x.\text{value};
\]

\[
\text{if } r > 0 \text{ then return } \lfloor \text{else return } \lceil \text{end if}
\]

\[
\text{end proc}
\]

5.12.4.3 Arithmetic

The following tables define procedures that perform common arithmetic on \text{FLOAT64} values using IEEE 754 rules. Note that \((\text{expr})_{64}\) is a shorthand for \text{realToFloat64}(\text{expr}).
The identity for floating-point addition is \( -\text{zero}_{64} \), not \(+\text{zero}_{64}\).
A procedure is denoted as:

```
proc f(param1: T1, ..., paramn: Tn): T
  step1;
  step2;
  ...
  stepn
end proc;
```

### 5.13 Procedures

A procedure is a function that receives zero or more arguments, performs computations, and optionally returns a result. Procedures may perform side effects. In this document the word `procedure` is used to refer to internal algorithms; the word `function` is used to refer to the programmer-visible `function` ECMAScript construct.

Note that `float64Remainder(float64Negate(x), y)` always produces the same result as `float64Negate(float64Remainder(x, y))`. Also, `float64Remainder(x, float64Negate(y))` always produces the same result as `float64Remainder(x, y)`.
If the procedure does not return a value, the : \( T \) on the first line is omitted.

\( f \) is the procedure’s name, \( param_1 \) through \( param_n \) are the procedure’s parameters, \( T_1 \) through \( T_n \) are the parameters’ respective semantic domains, \( T \) is the semantic domain of the procedure’s result, and \( step_1 \) through \( step_m \) describe the procedure’s computation steps, which may produce side effects and/or return a result. If \( T \) is omitted, the procedure does not return a result. When the procedure is called with argument values \( v_1 \) through \( v_n \), the procedure’s steps are performed and the result, if any, returned to the caller.

A procedure’s steps can refer to the parameters \( param_1 \) through \( param_n \); each reference to a parameter \( param \) evaluates to the corresponding argument value \( v \). Procedure parameters are statically scoped. Arguments are passed by value.

5.13.1 Operations

The only operation done on a procedure \( f \) is calling it using the \( (\text{arg}_1, \ldots, \text{arg}_n) \) syntax. \( f \) is computed first, followed by the argument expressions \( \text{arg}_1 \) through \( \text{arg}_n \), in left-to-right order. If the result of computing \( f \) or any of the argument expressions throws an exception \( e \), then the call immediately propagates \( e \) without computing any following argument expressions. Otherwise, \( f \) is invoked using the provided arguments and the resulting value, if any, returned to the caller.

Procedures are never compared using \( =, \neq \), or any of the other comparison operators.

5.13.2 Semantic Domains of Procedures

The semantic domain of procedures that take \( n \) parameters in semantic domains \( T_1 \) through \( T_n \) respectively and produce a result in semantic domain \( T \) is written as \( T_1 \sqcup T_2 \sqcup \ldots \sqcup T_n \sqcup T \). If \( n = 0 \), this semantic domain is written as \( () \sqcup T \). If the procedure does not produce a result, the semantic domain of procedures is written either as \( T_1 \sqcup T_2 \sqcup \ldots \sqcup T_n \sqcup () \) or as \( () \sqcup () \).

5.13.3 Steps

Computation steps in procedures are described using a mixture of English and formal notation. The various kinds of steps are described in this section. Multiple steps are separated by semicolons or periods and performed in order unless an earlier step exits via a \texttt{return} or propagates an exception.

\texttt{nothing}

A \texttt{nothing} step performs no operation.

\texttt{expression}

A computation step may consist of an expression. The expression is computed and its value, if any, ignored.

\( v: T \sqcup \texttt{expression} \)

\( v \sqcup \texttt{expression} \)

An assignment step is indicated using the assignment operator \( \sqcup \). This step computes the value of \texttt{expression} and assigns the result to the temporary variable or mutable global (see \texttt{*****}) \( v \). If this is the first time the temporary variable is referenced in a procedure, the variable’s semantic domain \( T \) is listed; any value stored in \( v \) is guaranteed to be a member of the semantic domain \( T \).

\( v: T \)

This step declares \( v \) to be a temporary variable with semantic domain \( T \) without assigning anything to the variable. \( v \) will not be read unless some other step first assigns a value to it.

Temporary variables are local to the procedures that define them (including any nested procedures). Each time a procedure is called it gets a new set of temporary variables.

\texttt{a.label \sqcup \texttt{expression}}

This form of assignment sets the value of field \texttt{label} of record \texttt{a} to the value of \texttt{expression}. 
if expression, then step; step; ...; step
elsif expression; then step; step; ...; step
...
elif expression, then step; step; ...; step
else step; step; ...; step
end if

An if step computes expression₁, which will evaluate to either true or false. If it is true, the first list of steps is performed. Otherwise, expression₂ is computed and tested, and so on. If no expression evaluates to true, the list of steps following the else is performed. The else clause may be omitted, in which case no action is taken when no expression evaluates to true.

case expression of
   T₁ do step; step; ...; step;
   T₂ do step; step; ...; step;
   ...
   Tₙ do step; step; ...; step
else step; step; ...; step
end case

A case step computes expression, which will evaluate to a value v. If v ∈ T₁, then the first list of steps is performed. Otherwise, if v ∈ T₂, then the second list of steps is performed, and so on. If v is not a member of any Tₙ, the list of steps following the else is performed. The else clause may be omitted, in which case v will always be a member of some Tₙ.

while expression do
   step;
   step;
   ...
   step
end while

A while step computes expression, which will evaluate to either true or false. If it is false, no action is taken. If it is true, the list of steps is performed and then expression is computed and tested again. This repeats until expression returns true (or until the procedure exits via a return or an exception is propagated out).

for each x ∈ expression do
   step;
   step;
   ...
   step
end for each

A for each step computes expression, which will evaluate to either a set or a list A. The list of steps is performed repeatedly with variable x bound to each element of A. If A is a list, x is bound to each of its elements in order; if A is a set, the order in which x is bound to its elements is arbitrary. The repetition ends after x has been bound to all elements of A (or when either the procedure exits via a return or an exception is propagated out).

return expression

A return step computes expression to obtain a value v and returns from the enclosing procedure with the result v. No further steps in the enclosing procedure are performed. The expression may be omitted, in which case the enclosing procedure returns with no result.

invariant expression

An invariant step is an informative note that states that computing expression at this point will always produce the value true.

throw expression

A throw step computes expression to obtain a value v and begins propagating exception v outwards, exiting partially performed steps and procedure calls until the exception is caught by a catch step. Unless the enclosing procedure catches this exception, no further steps in the enclosing procedure are performed.
try
  step;
  step;
  ...
  step
catch v: T do
  step;
  step;
  ...
  step
end try

A try step performs the first list of steps. If they complete normally (or if they return out of the current procedure), then the try step is done. If any of the steps propagates out an exception e, then if e \(\square\) T, then exception e stops propagating, variable v is bound to the value e, and the second list of steps is performed. If e \(\square\) T, then exception e keeps propagating out.

A try step does not intercept exceptions that may be propagated out of its second list of steps.

### 5.13.4 Nested Procedures

An inner proc may be nested as a step inside an outer proc. In this case the inner procedure is a closure and can access the parameters and temporaries of the outer procedure.

### 5.14 Grammars

The lexical and syntactic structure of ECMAScript programs is described in terms of context-free grammars. A context-free grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side, and a sequence of zero or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet. A grammar symbol is either a terminal or a nonterminal.

Each grammar contains at least one distinguished nonterminal called the goal symbol. If there is more than one goal symbol, the grammar specifies which one is to be used. A sentential form is a possibly empty sequence of grammar symbols that satisfies the following recursive constraints:

- The sequence consisting of only the goal symbol is a sentential form.
- Given any sentential form \(\square\) that contains a nonterminal \(N\), one may replace an occurrence of \(N\) in \(\square\) with the right-hand side of any production for which \(N\) is the left-hand side. The resulting sequence of grammar symbols is also a sentential form.

A derivation is a record, usually expressed as a tree, of which production was applied to expand each intermediate nonterminal to obtain a sentential form starting from the goal symbol. The grammars in this document are unambiguous, so each sentential form has exactly one derivation.

A sentence is a sentential form that contains only terminals. A sentence prefix is any prefix of a sentence, including the empty prefix consisting of no terminals and the complete prefix consisting of the entire sentence.

A language is the (perhaps infinite) set of a grammar’s sentences.

### 5.14.1 Grammar Notation

Terminal symbols are either literal characters (section 5.1), sequences of literal characters (syntactic grammar only), or other terminals such as Identifier defined by the grammar. These other terminals are denoted in bold.

Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a \(\square\) and one or more expansions of the nonterminal separated by vertical bars (|). The expansions are usually listed on separate lines but may be listed on the same line if they are short. An empty expansion is denoted as «empty».

To aid in reading the grammar, some rules contain informative cross-references to sections where nonterminals used in the rule are defined. These cross-references appear in parentheses in the right margin.
For example, the syntactic definition

```
SampleList []
  «empty»
  | . . . Identifier
  | SampleListPrefix
  | SampleListPrefix, . . . Identifier
```

states that the nonterminal `SampleList` can represent one of four kinds of sequences of input tokens:

- It can represent nothing (indicated by the «empty» alternative).
- It can represent the terminal `...` followed by any expansion of the nonterminal `Identifier`.
- It can represent any expansion of the nonterminal `SampleListPrefix`.
- It can represent any expansion of the nonterminal `SampleListPrefix` followed by the terminals `,` and `...` and any expansion of the nonterminal `Identifier`.

### 5.14.2 Lookahead Constraints

If the phrase “[lookahead [] set]” appears in the expansion of a nonterminal, it indicates that that expansion may not be used if the immediately following terminal is a member of the given `set`. That `set` can be written as a list of terminals enclosed in curly braces. For convenience, `set` can also be written as a nonterminal, in which case it represents the set of all terminals to which that nonterminal could expand.

For example, given the rules

```
DecimalDigit [] 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Decimals []
  DecimalDigit
  | Decimals DecimalDigit
```

the rule

```
LookaheadExample []
  n [lookahead [] {1, 3, 5, 7, 9}] Decimals
  | DecimalDigit [lookahead [] {DecimalDigit}]
```

matches either the letter `n` followed by one or more decimal digits the first of which is even, or a decimal digit not followed by another decimal digit.

### 5.14.3 Line Break Constraints

If the phrase “[no line break]” appears in the expansion of a production, it indicates that that production cannot be used if there is a line break in the input stream at the indicated position. Line break constraints are only present in the syntactic grammar. For example, the rule

```
ReturnStatement []
  return
  | return [no line break] ListExpression
```

indicates that the second production may not be used if a line break occurs in the program between the `return` token and the `ListExpression`. Unless the presence of a line break is forbidden by a constraint, any number of line breaks may occur between any two consecutive terminals in the input to the syntactic grammar without affecting the syntactic acceptability of the program.

### 5.14.4 Parameterised Rules

Many rules in the grammars occur in groups of analogous rules. Rather than list them individually, these groups have been summarised using the shorthand illustrated by the example below:

Metadefinitions such as

```
[] {normal, initial}
```
introduce grammar arguments \( a \) and \( b \). If these arguments later parameterise the nonterminal on the left side of a rule, that rule is implicitly replicated into a set of rules in each of which a grammar argument is consistently substituted by one of its variants. For example, the sample rule

\[
\begin{align*}
\text{AssignmentExpression}^a & \allowbreak \text{ConditionalExpression}^a \\
| \text{LeftSideExpression}^a & = \text{AssignmentExpression}^\text{normal} \\
| \text{LeftSideExpression}^a & \text{CompoundAssignment} \text{AssignmentExpression}^\text{normal}
\end{align*}
\]

expands into the following four rules:

\[
\begin{align*}
\text{AssignmentExpression}^\text{normal,allowIn} & \allowbreak \text{ConditionalExpression}^\text{normal,allowIn} \\
| \text{LeftSideExpression}^\text{normal} & = \text{AssignmentExpression}^\text{normal,allowIn} \\
| \text{LeftSideExpression}^\text{normal} & \text{CompoundAssignment} \text{AssignmentExpression}^\text{normal,allowIn}
\end{align*}
\]

Some of the expanded rules (such as the fourth one in the example above) may be unreachable from the grammar’s starting nonterminal; these are ignored.

### 5.14.5 Special Lexical Rules

A few lexical rules have too many expansions to be practically listed. These are specified by descriptive text instead of a list of expansions after the \( \mid \) .

Some lexical rules contain the metaword \texttt{except}. These rules match any expansion that is listed before the \texttt{except} but that does not match any expansion after the \texttt{except}; if multiple expansions are listed after the \texttt{except}, then they are separated by vertical bars (\( | \)). All of these rules ultimately expand into single characters. For example, the rule below matches any single \texttt{UnicodeCharacter} except the * and / characters:

\texttt{NonAsteriskOrSlash} \texttt{UnicodeCharacter except * | /}

### 5.15 Semantic Actions

Semantic actions tie the grammar and the semantics together. A semantic action ascribes semantic meaning to a grammar production.

Two examples illustrates the use of semantic actions. A description of the notation for specifying semantic actions follows the examples.
5.15.1 Example

Consider the following sample grammar, with the start nonterminal `Numeral`:

- `Digit {} 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
- `Digits {} Digit`
  - `Digits Digit`
- `Numeral {} Digits`
  - `Digits # Digits`

This grammar defines the syntax of an acceptable input: “37”, “33#4” and “30#2” are acceptable syntactically, while “1a” is not. However, the grammar does not indicate what these various inputs mean. That is the function of the semantics, which are defined in terms of actions on the parse tree of grammar rule expansions. Consider the following sample set of actions defined on this grammar, with a starting `Numeral` action called (in this example) `Value`:

- `Value[Digit] INTEGER = Digit’s decimal value (an integer between 0 and 9).`
- `DecimalValue[Digits] INTEGER;`
  - `DecimalValue[Digits {} Digit] = Value[Digit];`
  - `DecimalValue[Digits {} Digits1 Digit] = 10 * DecimalValue[Digits1] + Value[Digit];`
- `proc BaseValue[Digits] (base: INTEGER) INTEGER {
  \[Digits {} Digit\] do
  \[d: INTEGER \] Value[Digit];
  if d < base then return d else throw syntaxError end if;
  \[Digits2 {} Digits1 Digit\] do
  \[d: INTEGER \] Value[Digit];
  if d < base then return base * BaseValue[Digits1](base) + d else throw syntaxError end if
  end proc;
- `Value[Numeral] INTEGER;`
  - `Value[Numeral {} Digits] = DecimalValue[Digits];`
  - `Value[Numeral {} Digits1 # Digits2] begin
    base INTEGER INTEGER DecimalValue[Digits2];
    if base > 2 and base <= 10 then return BaseValue[Digits1](base) else throw syntaxError end if
  end;

Action names are written in cursive type. The definition

- `Value[Numeral] INTEGER;`
states that the action `Value` can be applied to any expansion of the nonterminal `Numeral`, and the result is an `INTEGER`. This action either maps an input to an integer or throws an exception. The code above throws the exception `syntaxError` when presented with the input “30#2”.

There are two definitions of the `Value` action on `Numeral`, one for each grammar production that expands `Numeral`:
Given the sample grammar rule
the example below.

In 5.15.2

The semantics can be evaluated on
character into which the

Digits
production
applied
applies
expansion.

22

30#2
33#4
37

Input
Semantic Result
37
15
throw syntaxError

Each definition of an action is allowed to perform actions on the terminals and nonterminals on the right side of the expansion. For example, Value applied to the first Numeral production (the one that expands Numeral into Digits) simply applies the DecimalValue action to the expansion of the nonterminal Digits and returns the result. On the other hand, Value applied to the second Numeral production (the one that expands Numeral into Digits # Digits) performs a computation using the results of the DecimalValue and BaseValue applied to the two expansions of the Digits nonterminals. In this case there are two identical nonterminals Digits on the right side of the expansion, so subscripts are used to indicate on which the actions DecimalValue and BaseValue are performed.

The definition

```
proc BaseValue[Digit] (base: INTEGER): INTEGER
  [Digits [] Digit] do
    d: INTEGER [] Value[Digit];
    if d < base then return d else throw syntaxError end if;
  end proc;
```

states that the action BaseValue can be applied to any expansion of the nonterminal Digits, and the result is a procedure that takes one INTEGER argument base and returns an INTEGER. The procedure’s body is comprised of independent cases for each production that expands Digits. When the procedure is called, the case corresponding to the expansion of the nonterminal Digits is evaluated.

The Value action on Digit

```
Value[Digit]: INTEGER = Digit’s decimal value (an integer between 0 and 9)
```

illustrates the direct use of a nonterminal Digit in a semantic expression. Using the nonterminal Digit in this way refers to the character into which the Digit grammar rule expands.

The semantics can be evaluated on the sample inputs to get the following results:

<table>
<thead>
<tr>
<th>Input</th>
<th>Semantic Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>33#4</td>
<td>15</td>
</tr>
<tr>
<td>30#2</td>
<td>throw syntaxError</td>
</tr>
</tbody>
</table>

5.15.2 Abbreviated Actions

In some cases the all actions named A for a nonterminal N’s rule are repetitive, merely calling A on the nonterminals on the right side of the expansions of N in the grammar. In these cases the semantics of action A are abbreviated, as illustrated by the example below.

Given the sample grammar rule
Expression []
  Subexpression
  Expression * Subexpression
  Subexpression + Subexpression
  this

the notation

Validate[Expression] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of Expression.

is an abbreviation for the following:

proc Validate[Expression] (cxt: CONTEXT, env: ENVIRONMENT)
  [Expression [] Subexpression] do Validate[Subexpression](cxt, env);
  [Expression [] Expression; * Subexpression] do
    Validate[Expression](cxt, env);
    Validate[Subexpression](cxt, env);
  [Expression [] Subexpression; + Subexpression;] do
    Validate[Subexpression](cxt, env);
    Validate[Subexpression](cxt, env);
  [Expression [] this] do nothing
end proc;

Note that:

• The expanded calls to Validate get the same arguments cxt and env passed in to the call to Validate on Expression.
• When an expansion of Expression has more than one nonterminal on its right side, Validate is called on all of the nonterminals in left-to-right order.
• When an expansion of Expression has no nonterminals on its right side, Validate does nothing.

5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

Action[nonterminal]; T;

This notation states that action Action can be performed on nonterminal nonterminal and returns a value that is a member of the semantic domain T. The action’s value is either defined using the notation Action[nonterminal [] expansion] = expression below or set as a side effect of computing another action via an action assignment.

Action[nonterminal [] expansion] = expression;

This notation specifies the value that action Action on nonterminal nonterminal computes in the case where nonterminal nonterminal expands to the given expansion. expansion can contain zero or more terminals and nonterminals (as well as other notations allowed on the right side of a grammar production). Furthermore, the terminals and nonterminals of expansion can be subscripted to allow them to be unambiguously referenced by action references or nonterminal references inside expression.

Action[nonterminal [] expansion]; T = expression;

This notation combines the above two — it specifies the semantic domain of the action as well as its value.

Action[nonterminal [] expansion]
begin
  step1;
  step2;
  ...
  stepn
end;
This notation is used when the computation of the action is too complex for an expression. Here the steps to compute the action are listed as step$_1$ through step$_m$. A return step produces the value of the action.

```
proc Action[nonterminal [] expansion] (param$_1$: T$_1$, ..., param$_n$: T$_n$): T
  step$_1$;
  step$_2$;
  ...
  step$_m$
end proc;
```

This notation is used only when Action returns a procedure when applied to nonterminal nonterminal with a single expansion expansion. Here the steps of the procedure are listed as step$_1$ through step$_m$.

```
proc Action[nonterminal] (param$_1$: T$_1$, ..., param$_n$: T$_n$): T
  [nonterminal [] expansion$_1$] do
    step;
  end;
  [nonterminal [] expansion$_2$] do
    step;
  end;
  ...
  [nonterminal [] expansion$_n$] do
    step;
end proc;
```

This notation is used only when Action returns a procedure when applied to nonterminal nonterminal with several expansions expansion$_1$, through expansion$_n$. The procedure is comprised of a series of cases, one for each expansion. Only the steps corresponding to the expansion found by the grammar parser used are evaluated.

```
Action[nonterminal] (param$_1$: T$_1$, ..., param$_n$: T$_n$) propagates the call to Action to every nonterminal in the expansion of nonterminal.
```

This notation is an abbreviation stating that calling Action on nonterminal causes Action to be called with the same arguments on every nonterminal on the right side of the appropriate expansion of nonterminal. See section 5.15.2.

### 5.16 Other Semantic Definitions

In addition to actions (section 5.15.3), the semantics sometimes define supporting top-level procedures and variables. The following notation is used for these definitions:

```
name: T = expression;
```

This notation defines name to be a constant value given by the result of computing expression. The value is guaranteed to be a member of the semantic domain T.

```
name: T = expression;
```

This notation defines name to be a mutable global value. Its initial value is the result of computing expression, but it may be subsequently altered using an assignment. The value is guaranteed to be a member of the semantic domain T.

```
proc f(param$_1$: T$_1$, ..., param$_n$: T$_n$): T
  step$_1$;
  step$_2$;
  ...
  step$_m$
end proc;
```

This notation defines f to be a procedure (section 5.13).
6 Source Text

ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 2.1 or later, using the UTF-16 transformation format. The text is expected to have been normalised to Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

ECMAScript source text can contain any of the Unicode characters. All Unicode white space characters are treated as white space, and all Unicode line/paragraph separators are treated as line separators. Non-Latin Unicode characters are allowed in identifiers, string literals, regular expression literals and comments.

In string literals, regular expression literals and identifiers, any character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely \u plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

**NOTE** Although this document sometimes refers to a “transformation” between a “character” within a “string” and the 16-bit unsigned integer that is the UTF-16 encoding of that character, there is actually no transformation because a “character” within a “string” is actually represented using that 16-bit unsigned value.

**NOTE** ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence \000A, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character 000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \000A occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write \n instead of \000A to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

6.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category Cf in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a Unicode escape sequence (see section *****) to include a Unicode format-control character inside a string or regular expression literal.

7 Lexical Grammar

This section defines ECMAScript’s lexical grammar. This grammar translates the source text into a sequence of input elements, which are either tokens or the special markers LineBreak and EndOfInput.

A token is one of the following:

- A keyword token, which is either:
  - One of the reserved words currently used by ECMAScript as break, case, catch, class, const, continue, default, delete, do, else, export, extends, false, final, finally, for, function, if, import, in, instanceof, is, namespace, new, null, package, private, public, return, static, super, switch, this, throw, true, try, typeof, use, var, void, while, with.
  - One of the reserved words reserved for future use abstract, debugger, enum, goto, implements, interface, native, protected, synchronized, throws, transient, volatile.
The sequence of input elements \textit{inputElements} is obtained as follows:

Let \textit{inputElements} be an empty sequence of input elements.

Let \textit{input} be the input sequence of characters. Append a special placeholder \texttt{End} to the end of \textit{input}.

Let \textit{state} be a variable that holds one of the constants \texttt{re}, \texttt{div}, or \texttt{num}. Initialise it to \texttt{re}.

Repeat the following steps until exited:

1. Find the longest possible prefix \( P \) of \textit{input} that is a member of the lexical grammar’s language (see section 5.14).
   
   Use the start symbol \textit{NextInputElement}*, \textit{NextInputElement}^\texttt{aw}, or \textit{NextInputElement}^\texttt{sw} depending on whether \textit{state} is \texttt{re}, \texttt{div}, or \texttt{num}, respectively. If the parse failed, signal a syntax error.

2. Compute the action \texttt{Lex} on the derivation of \( P \) to obtain an input element \( e \).

   If \( e \) is \texttt{EndOfInput}, then exit the repeat loop.

   Remove the prefix \( P \) from \textit{input}, leaving only the yet-unprocessed suffix of \textit{input}.

3. Append \( e \) to the end of the \textit{inputElements} sequence.

   If the \textit{inputElements} sequence does not form a valid sentence prefix of the language defined by the syntactic grammar, then:

   - If \( e \) is not \texttt{LineBreak}, but the next-to-last element of \textit{inputElements} is \texttt{LineBreak}, then insert a \texttt{VirtualSemicolon} terminal between the next-to-last element and \( e \) in \textit{inputElements}.
   - If \textit{inputElements} still does not form a valid sentence prefix of the language defined by the syntactic grammar, signal a syntax error.

   End if

   - If \( e \) is a \texttt{Number} token, then set \textit{state} to \texttt{num}. Otherwise, if the \textit{inputElements} sequence followed by the terminal / forms a valid sentence prefix of the language defined by the syntactic grammar, then set \textit{state} to \texttt{div}; otherwise, set \textit{state} to \texttt{re}.

   End if

4. End repeat

   If the \textit{inputElements} sequence does not form a valid sentence of the context-free language defined by the syntactic grammar, signal a syntax error and stop.

Return \textit{inputElements}.
7.1 Input Elements

Syntax

\[ \text{NextInputElement}^\text{re} \] \text{WhiteSpace InputElement}^\text{re} \quad (\text{WhiteSpace}: \text{7.2})

\[ \text{NextInputElement}^\text{div} \] \text{WhiteSpace InputElement}^\text{div}

\[ \text{NextInputElement}^\text{sum} \] \{\text{lookahead}\{\text{ContinuingIdentifierCharacter}, \}\} \text{WhiteSpace InputElement}^\text{div}

\text{InputElement}^\text{re} \quad \begin{align*}
& \text{LineBreaks} \quad (\text{LineBreaks}: \text{7.3}) \\
& \quad | \text{IdentifierOrKeyword} \quad (\text{IdentifierOrKeyword}: \text{7.5}) \\
& \quad | \text{Punctuator} \quad (\text{Punctuator}: \text{7.6}) \\
& \quad | \text{NumericLiteral} \quad (\text{NumericLiteral}: \text{7.7}) \\
& \quad | \text{StringLiteral} \quad (\text{StringLiteral}: \text{7.8}) \\
& \quad | \text{RegExpLiteral} \quad (\text{RegExpLiteral}: \text{7.9}) \\
& \quad | \text{EndOfInput}
\end{align*}

\text{InputElement}^\text{div} \quad \begin{align*}
& \text{LineBreaks} \quad (\text{DivisionPunctuator}: \text{7.6}) \\
& \quad | \text{IdentifierOrKeyword} \\
& \quad | \text{Punctuator} \\
& \quad | \text{DivisionPunctuator} \\
& \quad | \text{NumericLiteral} \\
& \quad | \text{StringLiteral} \\
& \quad | \text{EndOfInput}
\end{align*}

\text{EndOfInput} \quad \begin{align*}
& \text{End} \\
& \quad | \text{LineComment End} \quad (\text{LineComment}: \text{7.4})
\end{align*}

Semantics

The grammar parameter \( \nabla \) can be either \text{re} or \text{div}.

\text{Lex}[\text{NextInputElement}^\text{re}] := \text{INPUTELEMENT};
\text{Lex}[\text{NextInputElement}^\text{re} \text{WhiteSpace InputElement}^\text{re}] = \text{Lex}[\text{InputElement}^\text{re}];
\text{Lex}[\text{NextInputElement}^\text{div} \text{WhiteSpace InputElement}^\text{div}] = \text{Lex}[\text{InputElement}^\text{div}];
\text{Lex}[\text{NextInputElement}^\text{sum} \text{WhiteSpace InputElement}^\text{div}] = \text{Lex}[\text{InputElement}^\text{div}];
\text{Lex}[\text{InputElement}^\text{re}] := \text{INPUTELEMENT};
\text{Lex}[\text{InputElement}^\text{re} \text{LineBreaks}] = \text{LineBreak};
\text{Lex}[\text{InputElement}^\text{re} \text{IdentifierOrKeyword}] = \text{Lex}[\text{IdentifierOrKeyword}];
\text{Lex}[\text{InputElement}^\text{re} \text{Punctuator}] = \text{Lex}[\text{Punctuator}];
\text{Lex}[\text{InputElement}^\text{div} \text{DivisionPunctuator}] = \text{Lex}[\text{DivisionPunctuator}];
\text{Lex}[\text{InputElement}^\text{re} \text{NumericLiteral}] = \text{Lex}[\text{NumericLiteral}];
\text{Lex}[\text{InputElement}^\text{re} \text{StringLiteral}] = \text{Lex}[\text{StringLiteral}];
\text{Lex}[\text{InputElement}^\text{sum} \text{RegExpLiteral}] = \text{Lex}[\text{RegExpLiteral}];
\text{Lex}[\text{InputElement}^\text{div} \text{EndOfInput}] = \text{EndOfInput};
7.2 White space

Syntax

\[ \text{WhiteSpace} \begin{array}{l} 
\text{«empty»} \\
\mid \text{WhiteSpace WhiteSpaceCharacter} \\
\mid \text{WhiteSpace SingleLineBlockComment} 
\end{array} \]  
\[ \text{(SingleLineBlockComment: 7.4)} \]

\[ \text{WhiteSpaceCharacter} \begin{array}{l} 
\text{«TAB»} \mid \text{«VT»} \mid \text{«FF»} \mid \text{«SP»} \mid \text{«u00A0»} \\
\mid \text{Any other character in category Zs in the Unicode Character Database} 
\end{array} \]

NOTE  White space characters are used to improve source text readability and to separate tokens from each other, but are otherwise insignificant. White space may occur between any two tokens.

7.3 Line Breaks

Syntax

\[ \text{LineBreak} \begin{array}{l} 
\text{LineTerminator} \\
\mid \text{LineComment LineTerminator} \\
\mid \text{MultiLineBlockComment} 
\end{array} \]  
\[ \text{(LineComment: 7.4)} \]

\[ \text{LineBreaks} \begin{array}{l} 
\text{LineBreak} \\
\mid \text{LineComment LineBreak} \\
\mid \text{WhiteSpace LineBreak} 
\end{array} \]  
\[ \text{(WhiteSpace: 7.2)} \]

\[ \text{LineTerminator} \begin{array}{l} 
\text{«LF»} \mid \text{«CR»} \mid \text{«u2028»} \mid \text{«u2029»} 
\end{array} \]

NOTE  Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section *****).

7.4 Comments

Syntax

\[ \text{LineComment} \begin{array}{l} 
\text{/ / LineCommentCharacters} 
\end{array} \]

\[ \text{LineCommentCharacters} \begin{array}{l} 
\text{«empty»} \\
\mid \text{LineCommentCharacters NonTerminator} 
\end{array} \]

\[ \text{SingleLineBlockComment} \begin{array}{l} 
\text{/ * BlockCommentCharacters * /} 
\end{array} \]

\[ \text{BlockCommentCharacters} \begin{array}{l} 
\text{«empty»} \\
\mid \text{BlockCommentCharacters NonTerminatorOrSlash} \\
\mid \text{PreSlashCharacters /} 
\end{array} \]

\[ \text{PreSlashCharacters} \begin{array}{l} 
\text{«empty»} \\
\mid \text{BlockCommentCharacters NonTerminatorOrAsteriskOrSlash} \\
\mid \text{PreSlashCharacters /} 
\end{array} \]
NOTE Comments can be either line comments or block comments. Line comments start with // and continue to the end of the line. Block comments start with /* and end with */. Block comments can span multiple lines but cannot nest.

Except when it is on the last line of input, a line comment is always followed by a Line Terminator. That Line Terminator is not considered to be part of that line comment; it is recognised separately and becomes a Line Break. A block comment that actually spans more than one line is also considered to be a Line Break.

7.5 Keywords and Identifiers

Syntax

IdentifierOrKeyword [] IdentifierName

Semantics

Lex[IdentifierOrKeyword [] IdentifierName]: INPUTELEMENT

begin
  id: STRING [] LexName[IdentifierName];
  if id [] ("abstract","as","break","case","catch","class","const","continue","debugger","default","delete","do","else","enum","exclude","export","extends","false","final","finally","for","function","get","goto","if","implements","import","in","include","instanceof","interface","is","named","namespace","native","new","null","package","private","protected","public","return","set","static","super","switch","synchronized","this","throw","throws","transient","true","try","typeof","use","var","void","volatile","while","with")
  and IdentifierName contains no escape sequences (i.e. expansions of the NullEscape or HexEscape nonterminals)
  then return the keyword token id
  else return an Identifier token with the name id
  end if
end.

Lex[IdentifierOrKeyword [] IdentifierName]

Let id be the string LexString[IdentifierName].

If IdentifierName contains no escape sequences (i.e. expansions of the NullEscape or HexEscape nonterminals) and exactly matches one of the keywords abstract, as, break, case, catch, class, const, continue, debugger, default, delete, do, else, enum, exclude, export, extends, false, final, finally, for, function, get, goto, if, implements, import, in, include, instanceof, interface, is, namespace, named, native, new, null, package, private, protected, public, return, set, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, use, var, void, volatile, while, with, then return a keyword token with string contents id.

Return an Identifier token with string contents id.

NOTE Even though the lexical grammar treats exclude, get, include, named, and set as keywords, the syntactic grammar contains productions that permit them to be used as identifier names. The other keywords are reserved and may not be used as
identifier names. However, an IdentifierName can never be a keyword if it contains any escape characters, so, for example, one can use new as the name of an identifier by including an escape sequence in it; \_new is one possibility, and n\x65w is another.

**Syntax**

IdentifierName []
  InitialIdentifierCharacterOrEscape
  | NullEscapes InitialIdentifierCharacterOrEscape
  | IdentifierName ContinuingIdentifierCharacterOrEscape
  | IdentifierName NullEscape

NullEscapes []
  NullEscape
  | NullEscapes NullEscape

NullEscape [] \

InitialIdentifierCharacterOrEscape []
  InitialIdentifierCharacter
  | \ HexEscape

InitialIdentifierCharacter [] UnicodeInitialAlphabetic | $ | _

UnicodeInitialAlphabetic [] Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), or Nl (letter number) in the Unicode Character Database

ContinuingIdentifierCharacterOrEscape []
  ContinuingIdentifierCharacter
  | \ HexEscape

ContinuingIdentifierCharacter [] UnicodeAlphanumeric | $ | _

UnicodeAlphanumeric [] Any character in category Lu (uppercase letter), Ll (lowercase letter), Lt (titlecase letter), Lm (modifier letter), Lo (other letter), Nd (decimal number), Nl (letter number), Mn (non-spacing mark), Mc (combining spacing mark), or Pc (connector punctuation) in the Unicode Character Database

**Semantics**

LexName[IdentifierName]: STRING;
  LexName[IdentifierName] [] InitialIdentifierCharacterOrEscape = [LexChar[InitialIdentifierCharacterOrEscape]];
  LexName[IdentifierName] [] NullEscapes InitialIdentifierCharacterOrEscape
    = [LexChar[InitialIdentifierCharacterOrEscape]];
  LexName[IdentifierName0] [] IdentifierName1 ContinuingIdentifierCharacterOrEscape
    = LexName[IdentifierName0] [] IdentifierName1 ContinuingIdentifierCharacterOrEscape;
  LexName[IdentifierName0] [] IdentifierName1 NullEscape = LexName[IdentifierName1];

LexChar[InitialIdentifierCharacterOrEscape]: CHARACTER;
  LexChar[InitialIdentifierCharacterOrEscape] [] InitialIdentifierCharacter = InitialIdentifierCharacter;
  LexChar[InitialIdentifierCharacterOrEscape] [] \ HexEscape
    begin
      ch: CHARACTER [] LexChar[HexEscape];
      if ch is in the set of characters accepted by the nonterminal InitialIdentifierCharacter then return ch
      else throw syntaxError
    end if
  end;

LexString[IdentifierName [] InitialIdentifierCharacterOrEscape]  
LexString[IdentifierName [] NullEscape InitialIdentifierCharacterOrEscape]  
   Return a one-character string with the character LexChar[InitialIdentifierCharacterOrEscape].

LexString[IdentifierName [] IdentifierName_ContinuingIdentifierCharacterOrEscape]  
   Return a string consisting of the string LexString[IdentifierName] concatenated with the character LexChar[ContinuingIdentifierCharacterOrEscape].

LexString[IdentifierName [] IdentifierName NullEscape]  
   Return the string LexString[IdentifierName].

LexChar[InitialIdentifierCharacterOrEscape [] InitialIdentifierCharacter]  
   Return the character InitialIdentifierCharacter.

LexChar[InitialIdentifierCharacterOrEscape [] \ HexEscape]  
   Let ch be the character LexChar[HexEscape].
   If ch is in the set of characters accepted by the nonterminal InitialIdentifierCharacter, then return ch.
   Signal a syntax error.

LexChar[ContinuingIdentifierCharacterOrEscape]: CHARACTER;  
LexChar[ContinuingIdentifierCharacterOrEscape [] ContinuingIdentifierCharacter]  
   = ContinuingIdentifierCharacter;
LexChar[ContinuingIdentifierCharacterOrEscape [] \ HexEscape]  
   begin  
      ch: CHARACTER [] LexChar[HexEscape];
      if ch is in the set of characters accepted by the nonterminal ContinuingIdentifierCharacter then return ch else throw syntaxError
   end if
   end;

LexChar[ContinuingIdentifierCharacterOrEscape [] ContinuingIdentifierCharacter]  
   Return the character ContinuingIdentifierCharacter.

LexChar[ContinuingIdentifierCharacterOrEscape [] \ HexEscape]  
   Let ch be the character LexChar[HexEscape].
   If ch is in the set of characters accepted by the nonterminal ContinuingIdentifierCharacter, then return ch.
   Signal a syntax error.

The characters in the specified categories in version 2.43.0 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations; however, conforming ECMAScript implementations may allow additional legal identifier characters based on the category assignment from later versions of Unicode.

NOTE  Identifiers are interpreted according to the grammar given in Section 5.16 of version 3.0 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode standard. This standard specifies one departure from the grammar given in the Unicode standard: $ and _ are permitted anywhere in an identifier. $ is intended for use only in mechanically generated code.

   Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier. An escape sequence cannot be used to put a character into an identifier that would otherwise be illegal in that position of the identifier.

   Two identifiers that are canonically equivalent according to the Unicode standard are not equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.
7.6 Punctuators

Syntax

```markdown
Punctuator []

| ! | ! = | ! = = | % | % = | & | & & |
| & = | & = | ( | ) | * | * = | + |
| ++ | += | , | - | -= | -= | - = |
| ... | : | :: | ; | < | <= | < < |
| <= | = | == | == == | > | >= | > > |
| >= | >> | >>> | >> >> | ? | [ | ] | ^ |
| ^ = | ^ ^ | ^ ^ = | { | | | | | | ^ |
| | | = | } | ~ |
```

```markdown
DivisionPunctuator []

/ [lookahead[] {/, *}]

| / = |
```

Semantics

Lex[Punctuator]: `TOKEN` = the punctuator token `Punctuator`.

Lex[DivisionPunctuator]: `TOKEN` = the punctuator token `DivisionPunctuator`.

7.7 Numeric literals

Syntax

```markdown
NumericLiteral []

  DecimalLiteral
  | HexIntegerLiteral
  | DecimalLiteral LetterF
  | IntegerLiteral LetterL.
  | IntegerLiteral LetterU LetterL

IntegerLiteral []

  DecimalIntegerLiteral
  | HexIntegerLiteral

LetterF [] F | f

LetterL [] L | l

LetterU [] U | u

DecimalLiteral []

  Mantissa
  | Mantissa LetterE SignedInteger

LetterE [] E | e

Mantissa []

  DecimalIntegerLiteral
  | DecimalIntegerLiteral .
  | DecimalIntegerLiteral . Fraction
  | . Fraction
```
DecimalIntegerLiteral |
0 |
NonZeroDecimalDigits

NonZeroDecimalDigits |
NonZeroDigit |
NonZeroDecimalDigits ASCIIDigit

Fraction |
DecimalDigits

SignedInteger |
DecimalDigits |
+ DecimalDigits |
- DecimalDigits

DecimalDigits |
ASCIIDigit ASCIIDigit

HexIntegerLiteral |
0 LetterX HexDigit |
HexIntegerLiteral HexDigit

LetterX | X | x

ASCIIDigit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

NonZeroDigit | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexDigit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | a | b | c | d | e | f

Semantics

Lex[NumericLiteral]: TOKEN;
Lex[NumericLiteral DecimalLiteral] = a Number token with the value realToFloat64(LexNumber[DecimalLiteral]);
Lex[NumericLiteral HexIntegerLiteral] = a Number token with the value realToFloat64(LexNumber[HexIntegerLiteral]);
Lex[NumericLiteral DecimalLiteral LetterF] = a Number token with the value realToFloat32(LexNumber[DecimalLiteral]);
Lex[NumericLiteral IntegerLiteral LetterL] begin
  i: INTEGER LexNumber[IntegerLiteral];
  if $i \leq 2^{63} - 1$ then return a Number token with the value LONG(value: i)
  elsif $i = 2^{63}$ then return NegatedMinLong
  else throw rangeError
  end if
end;
Lex[NumericLiteral IntegerLiteral LetterU LetterL] begin
  i: INTEGER LexNumber[IntegerLiteral];
  if $i \leq 2^{64} - 1$ then return a Number token with the value ULONG(value: i)
  else throw rangeError
  end if
end;
LexNumber[IntegerLiteral]: INTEGER;
LexNumber[IntegerLiteral DecimallIntegerLiteral] = LexNumber[DecimalIntegerLiteral];
LexNumber[IntegerLiteral HexIntegerLiteral] = LexNumber[HexIntegerLiteral];
NOTE  Note that all digits of hexadecimal literals are significant.

    LexNumber[DecimalLiteral]: RATIONAL;
    LexNumber[DecimalLiteral  Mantissa] = LexNumber[Mantissa];
    LexNumber[DecimalLiteral  Mantissa LetterE SignedInteger] = LexNumber[Mantissa]10LexNumber[SignedInteger];

LexNumber[Mantissa]: RATIONAL;
    LexNumber[Mantissa  DecimalIntegerLiteral] = LexNumber[DecimalIntegerLiteral];
    LexNumber[Mantissa  DecimalIntegerLiteral .] = LexNumber[DecimalIntegerLiteral];
    LexNumber[Mantissa  DecimalIntegerLiteral . Fraction]
        = LexNumber[DecimalIntegerLiteral] + LexNumber[Fraction];
    LexNumber[Mantissa  . Fraction] = LexNumber[Fraction];

LexNumber[DecimalIntegerLiteral]: INTEGER;
    LexNumber[DecimalIntegerLiteral  0] = 0;
    LexNumber[DecimalIntegerLiteral  NonZeroDecimalDigits] = LexNumber[NonZeroDecimalDigits];

LexNumber[NonZeroDecimalDigits]: INTEGER;
    LexNumber[NonZeroDecimalDigits  NonZeroDigit] = DecimalValue[NonZeroDigit];
    LexNumber[NonZeroDecimalDigits,  NonZeroDecimalDigits, ASCIIDigit]
        = 10LexNumber[NonZeroDecimalDigits,] + DecimalValue[ASCIIDigit];

LexNumber[Fraction  DecimalDigits]: RATIONAL = LexNumber[DecimalDigits]/10^NDigits[DecimalDigits],

LexNumber[SignedInteger]: INTEGER;
    LexNumber[SignedInteger  DecimalDigits] = LexNumber[DecimalDigits];
    LexNumber[SignedInteger  + DecimalDigits] = LexNumber[DecimalDigits];
    LexNumber[SignedInteger  DecimalDigits] = –LexNumber[DecimalDigits];

LexNumber[DecimalDigits]: INTEGER;
    LexNumber[DecimalDigits  ASCIIDigit] = DecimalValue[ASCIIDigit];
    LexNumber[DecimalDigits,  ASCIIDigit]
        = 10LexNumber[DecimalDigits,] + DecimalValue[ASCIIDigit];

NDigits[DecimalDigits]: INTEGER;
    NDigits[DecimalDigits  ASCIIDigit] = 1;
    NDigits[DecimalDigits,  ASCIIDigit] = NDigits[DecimalDigits,] + 1;

LexNumber[HexIntegerLiteral]: INTEGER;
    LexNumber[HexIntegerLiteral  0 LetterX HexDigit] = HexValue[HexDigit];
    LexNumber[HexIntegerLiteral,  HexDigit]
        = 16LexNumber[HexIntegerLiteral,] + HexValue[HexDigit];

DecimalValue[ASCIIDigit]: INTEGER = ASCIIDigit’s decimal value (an integer between 0 and 9).

DecimalValue[NonZeroDigit] = NonZeroDigit’s decimal value (an integer between 1 and 9).

HexValue[HexDigit]: INTEGER = HexDigit’s hexadecimal value (an integer between 0 and 15). The letters A, B, C, D, E, and F, in either upper or lower case, have values 10, 11, 12, 13, 14, and 15, respectively.

7.8 String literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence starting with a backslash.
Syntax

The grammar parameter \( q \) can be either single or double.

```
StringLiteral []
   | StringChars^single ^
   | " StringChars^double "

StringChars^[] []
   «empty»
   | StringChars^[] StringChar^[]
   | StringChars^[] NullEscape

LiteralStringChar^[] ^
   | LiteralStringChar^[]
   | \ StringEscape

LiteralStringChar^[]
   | UnicodeCharacter except ^ \ | \ LineTerminator

LiteralStringChar^[]
   | UnicodeCharacter except " \ | \ LineTerminator

StringEscape []
   | ControlEscape
   | ZeroEscape
   | HexEscape
   | IdentityEscape

IdentityEscape []
   | NonTerminator except _ | UnicodeAlphanumeric

ControlEscape []
   | b | f | n | r | t | v

ZeroEscape []
   | 0 [lookahead{ASCIIDigit}]

HexEscape []
   | x HexDigit HexDigit
   | u HexDigit HexDigit HexDigit HexDigit
```

Semantics

Lex[StringLiteral]: TOKEN:
```
Lex[StringLiteral] ^ \ StringChars^[] \ = a String token with the value Lex[StringChars^[]];
Lex[StringLiteral] ^ " StringChars^[] " = a String token with the value Lex[StringChars^[]];
```

Lex[StringChars^[]]: STRING:
```
Lex[StringChars^[]] ^ «empty» = "";
Lex[StringChars^[]] ^ StringChars^[] StringChar^[] = Lex[StringChars^[]] \ @ \ Lex[Char[StringChar^[]]];
Lex[StringChars^[]] ^ StringChars^[] NullEscape = Lex[StringChars^[]];
```

Lex[Char[StringChar^[]]]: CHARACTER:
```
Lex[Char[StringChar^[]]] ^ LiteralStringChar^[] = LiteralStringChar^[];
Lex[Char[StringChar^[]]] ^ \ StringEscape = Lex[Char[StringEscape]];
```

Lex[Char[StringEscape]]: CHARACTER:
```
Lex[Char[StringEscape]] ^ ControlEscape = Lex[Char[ControlEscape]];
Lex[Char[StringEscape]] ^ ZeroEscape = Lex[Char[ZeroEscape]];
Lex[Char[StringEscape]] ^ HexEscape = Lex[Char[HexEscape]];
Lex[Char[StringEscape]] ^ IdentityEscape = IdentityEscape;
```
Lex[StringLiteral[[StringChars]]]
  Return a string token with string contents Lex[StringChars].

Lex[StringLiteral[" StringChars"]]
  Return a string token with string contents Lex[StringChars].

Lex[StringChars[empty]] = ""

Lex[StringChars\StringChars\LiteralStringChar]
  Return a string consisting of the string Lex[StringChars\LiteralStringChar] concatenated with the character LexChar[LiteralStringChar].

Lex[StringChars\StringChars\NullEscape] = Lex[StringChars\LiteralStringChar]

LexChar[StringChars\LiteralStringChar]
  Return the character LiteralStringChar.

LexChar[ControlEscape]: CHARACTER;
  LexChar[ControlEscape \b] = '«BS»';
  LexChar[ControlEscape \f] = '«FF»';
  LexChar[ControlEscape \n] = '«LF»';
  LexChar[ControlEscape \r] = '«CR»';
  LexChar[ControlEscape \t] = '«TAB»';
  LexChar[ControlEscape \v] = '«VT»';

LexChar[ZeroEscape \0 \lookahead {ASCJIDigit}]: CHARACTER = '«NUL»';

LexChar[HexEscape]: CHARACTER;
  LexChar[HexEscape \x HexDigit\HexDigit] = codeToCharacter(16\HexValue[HexDigit1] + HexValue[HexDigit2]);
  LexChar[HexEscape \x HexDigit\HexDigit\HexDigit\HexDigit] = codeToCharacter(4096\HexValue[HexDigit1] + 256\HexValue[HexDigit2] + 16\HexValue[HexDigit3] + HexValue[HexDigit4]);

LexChar[ControlEscape \b] = '«BS»';
LexChar[ControlEscape \f] = '«FF»';
LexChar[ControlEscape \n] = '«LF»';
LexChar[ControlEscape \r] = '«CR»';
LexChar[ControlEscape \t] = '«TAB»';
LexChar[ControlEscape \v] = '«VT»';

NOTE  A backslash followed by a non-alphanumeric character other than \_ or a line break represents character \c.

LexChar[ControlEscape \b] = '«BS»';
LexChar[ControlEscape \f] = '«FF»';
LexChar[ControlEscape \n] = '«LF»';
LexChar[ControlEscape \r] = '«CR»';
LexChar[ControlEscape \t] = '«TAB»';
LexChar[ControlEscape \v] = '«VT»';
LexChar[ZeroEscape [] «[lookahead[]{ASCII[Digit]}]] = ‘NUL’

LexChar[HexEscape [] = HexDigit, HexDigit,]
Let \( n = 16 \times \text{LexNumber}[\text{HexDigit},] + \text{LexNumber}[\text{HexDigit},] \)
Return the character with code point value \( n \).

LexChar[HexEscape [] = HexDigit, HexDigit, HexDigit, HexDigit,]
Let \( n = 4096 \times \text{LexNumber}[\text{HexDigit},] + 256 \times \text{LexNumber}[\text{HexDigit},] + 16 \times \text{LexNumber}[\text{HexDigit},] + \text{LexNumber}[\text{HexDigit},] \)
Return the character with code point value \( n \).

NOTE A LineTerminator character cannot appear in a string literal, even if preceded by a backslash \. The correct way to cause a line terminator character to be part of the string value of a string literal is to use an escape sequence such as \n or \u000A.

### 7.9 Regular expression literals

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the RegExpBody and the RegExpFlags are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor’s grammar, but it should not extend the RegExpBody and RegExpFlags productions or the productions used by these productions.

#### Syntax

\[
\text{RegExpLiteral} \ [\text{RegExpBody} \ \text{RegExpFlags}] \\
\text{RegExpFlags} \ [\text{«empty»}] \\
| \text{RegExpFlags} \ \text{ContinuingIdentifierCharacterOrEscape} \ \\
| \text{RegExpFlags} \ \text{NullEscape} \ \\
\text{RegExpBody} \ [\text{[lookahead[]{*}] \ RegExpChars} \]/
\]

\[
\text{RegExpChars} \ [\text{RegExpChar} \]
\]

\[
\text{RegExpChar} \ [\text{\ OrdinaryRegExpChar} \]
\]

\[
\text{OrdinaryRegExpChar} \ [\text{\ NonTerminator} \]
\]

#### Semantics

\[
\text{Lex[RegExpLiteral [] \ RegExpBody \ RegExpFlags]}: \text{TOKEN} \\
= \text{A RegularExpression token with the body LexString[RegExpBody] and flags LexString[RegExpFlags]};
\]

\[
\text{LexString[RegExpFlags]}: \text{STRING}; \\
\text{LexString[RegExpFlags]} \ [\text{«empty»}] = “”;
\]

\[
\text{LexString[RegExpFlags] \ RegExpChar} \ [\text{ContinuingIdentifierCharacterOrEscape}] \\
= \text{LexString[RegExpFlags]} \oplus \text{LexChar[ContinuingIdentifierCharacterOrEscape]};
\]

\[
\text{LexString[RegExpFlags] \ NullEscape} = \text{LexString[RegExpFlags]};
\]

\[
\text{LexString[RegExpBody [] \ RegExpChars]}/ \ [\text{[lookahead[]{*}] \ RegExpChars}]/: \text{STRING} = \text{LexString[RegExpChars]};
\]
LexString[RegExpChars]: STRING;
LexString[RegExpChars [] RegExpChar] = LexString[RegExpChar];
LexString[RegExpChars [] RegExpChars: RegExpChar] = LexString[RegExpChars] @ LexString[RegExpChar];

LexString[RegExpChar]: STRING;
LexString[RegExpChar [] OrdinaryRegExpChar] = [OrdinaryRegExpChar];
LexString[RegExpChar [] \ NonTerminator] = [\ NonTerminator]; (Note that the result string has two characters)

Lex[RegExpLiteral [] RegExpBody RegExpFlags]
Return a regularExpression token with the body string LexString[RegExpBody] and flags string LexString[RegExpFlags].

LexString[RegExpFlags [] \empty] = ""

LexString[RegExpFlags [] RegExpFlags, ContinuingIdentifierCharacterOrEscape]
Return a string consisting of the string LexString[RegExpFlags] concatenated with the character LexChar[ContinuingIdentifierCharacterOrEscape].

LexString[RegExpFlags [] RegExpFlags, NullEscape] = LexString[RegExpFlags,]

LexString[RegExpBody [] / [lookahead[] (\+)] RegExpChars /] = LexString[RegExpChars]

LexString[RegExpChars [] RegExpChar] = LexString[RegExpChar]

LexString[RegExpChars [] RegExpChars, RegExpChar]
Return a string consisting of the string LexString[RegExpChars] concatenated with the string LexString[RegExpChar].

LexString[RegExpChar [] OrdinaryRegExpChar]
Return a string consisting of the single character OrdinaryRegExpChar.

LexString[RegExpChar [] \ NonTerminator]
Return a string consisting of the two characters '\\' and NonTerminator.

NOTE A regular expression literal is an input element that is converted to a RegExp object (section ****) when it is scanned. The object is created before evaluation of the containing program or function begins. Evaluation of the literal produces a reference to that object; it does not create a new object. Two regular expression literals in a program evaluate to regular expression objects that never compare as === to each other even if the two literals' contents are identical. A RegExp object may also be created at runtime by new RegExp (section ****) or calling the RegExp constructor as a function (section ****).

NOTE Regular expression literals may not be empty; instead of representing an empty regular expression literal, the characters // start a single-line comment. To specify an empty regular expression, use / (?: ) /.

8 Program Structure

8.1 Packages

8.2 Scopes

9 Data Model

This chapter describes the essential state held in various ECMAScript objects. This state is presented abstractly using the formalisms from chapter 5. Much of the state held in these objects is observable by ECMAScript programmers only
indirectly, and implementations are encouraged to implement these objects in more efficient ways as long as the observable behaviour is the same as described here.

9.1 Objects

An object is a first-class data value visible to ECMAScript programmers. Every object is either \texttt{undefined}, \texttt{null}, a Boolean, a signed or unsigned 64-bit integer, a single or double-precision floating-point number, a character, a string, a namespace, a compound attribute, a class, a method closure, a prototype instance, a class instance, a package object, or the global object. These kinds of objects are described in the subsections below.

\texttt{OBJECT} is the semantic domain of all possible objects and is defined as:

\begin{verbatim}
OBJECT = UNDEFINED \ Bool \ Null \ Boolean \ Long \ ULONG \ FLOAT32 \ FLOAT64 \ Character \ String \ Namespace \ CompoundAttribute \ Class \ MethodClosure \ Prototype \ Instance \ Package \ Global;
\end{verbatim}

A \texttt{PrimitiveObject} is either \texttt{undefined}, \texttt{null}, a Boolean, a signed or unsigned 64-bit integer, a single or double-precision floating-point number, a character, or a string:

\begin{verbatim}
PrimitiveObject = UNDEFINED \ Bool \ Null \ Boolean \ Long \ ULONG \ FLOAT32 \ FLOAT64 \ Character \ String;
\end{verbatim}

A \texttt{DynamicObject} is an object that can host dynamic properties:

\begin{verbatim}
DynamicObject = Prototype \ SimpleInstance \ CallableInstance \ DynamicInstance \ Global;
\end{verbatim}

The semantic domain \texttt{ObjectOpt} consists of all objects as well as the tag \texttt{none} which denotes the absence of an object. \texttt{none} is not a value visible to ECMAScript programmers.

\begin{verbatim}
ObjectOpt = Object \ {none};
\end{verbatim}

The semantic domain \texttt{ObjectI} consists of all objects as well as the tag \texttt{inaccessible} which denotes that a variable’s value is not available at this time (for example, a variable whose value is accessible only at run time would hold the value \texttt{inaccessible} at compile time). \texttt{inaccessible} is not a value visible to ECMAScript programmers.

\begin{verbatim}
ObjectI = Object \ {inaccessible};
\end{verbatim}

The semantic domain \texttt{ObjectIOpt} consists of all objects as well as the tags \texttt{none} and \texttt{inaccessible}:

\begin{verbatim}
ObjectIOpt = Object \ {inaccessible, none};
\end{verbatim}

Some of the variables are in an uninitialised state before first being assigned a value. The semantic domain \texttt{ObjectU} describes such a variable, which contains either an object or the tag \texttt{uninitialised}. \texttt{uninitialised} is not a value visible to ECMAScript programmers. The difference between \texttt{uninitialised} and \texttt{Inaccessible} is that a variable holding the value \texttt{uninitialised} can be written but not read, while a variable holding the value \texttt{inaccessible} can be neither read nor written.

\begin{verbatim}
ObjectU = Object \ {uninitialised};
\end{verbatim}

The semantic domain \texttt{BooleanOpt} consists of the tags \texttt{true}, \texttt{false}, and \texttt{none}:

\begin{verbatim}
BooleanOpt = Boolean \ {none};
\end{verbatim}

The semantic domain \texttt{IntegerOpt} consists of all integers as well as \texttt{none}:

\begin{verbatim}
IntegerOpt = Integer \ {none};
\end{verbatim}

9.1.1 Undefined

There is exactly one \texttt{undefined} value. The semantic domain \texttt{Undefined} consists of that one value.

\begin{verbatim}
Undefined = {undefined}
\end{verbatim}

9.1.2 Null

There is exactly one \texttt{null} value. The semantic domain \texttt{Null} consists of that one value.

\begin{verbatim}
Null = {null}
\end{verbatim}
9.1.3 Booleans

There are two Booleans, true and false. The semantic domain Boolean consists of these two values. See section 5.4.

9.1.4 Numbers

The semantic domains Long, ULONG, FLOAT32, and FLOAT64, collectively denoted by the domain GENERALNUMBER, represent the numeric types supported by ECMAScript. See section 5.12.

9.1.5 Strings

The semantic domain STRING consists of all representable strings. See section 5.9. A STRING is considered to be of either the class String if s’s length isn’t 1 or the class Character if s’s length is 1.

The semantic domain STRINGOPT consists of all strings as well as the tag none which denotes the absence of a string. none is not a value visible to ECMAScript programmers.

\[
\text{STRINGOPT} = \text{STRING} \cup \{\text{none}\}
\]

9.1.6 Namespaces

A namespace object is represented by a NAMESPACE record (see section 5.11) with the field below. Each time a namespace is created, the new namespace is different from every other namespace, even if it happens to share the name of an existing namespace.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>The namespace’s name used by toString</td>
</tr>
</tbody>
</table>

9.1.6.1 Qualified Names

A QualifiedName tuple (see section 5.10) has the fields below and represents a name qualified with a namespace.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespace</td>
<td>NAMESPACE</td>
<td>The namespace qualifier</td>
</tr>
<tr>
<td>id</td>
<td>STRING</td>
<td>The name</td>
</tr>
</tbody>
</table>

QUALIFIEDNAMEOPT consists of all qualified names as well as none:

\[
\text{QUALIFIEDNAMEOPT} = \text{QUALIFIEDNAME} \cup \{\text{none}\}
\]

MULTINAME is the semantic domain of sets of qualified names. Multinames are used internally in property lookup.

\[
\text{MULTINAME} = \text{QUALIFIEDNAME}\{}\}
\]

9.1.7 Compound attributes

Compound attribute objects are all values obtained from combining zero or more syntactic attributes (see ******) that are not Booleans or single namespaces. A compound attribute object is represented by a COMPOUNDATTRIBUTE tuple (see section 5.10) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespaces</td>
<td>NAMESPACE{}</td>
<td>The set of namespaces contained in this attribute</td>
</tr>
<tr>
<td>explicit</td>
<td>BOOLEAN</td>
<td>true if the explicit attribute has been given</td>
</tr>
<tr>
<td>dynamic</td>
<td>BOOLEAN</td>
<td>true if the dynamic attribute has been given</td>
</tr>
<tr>
<td>memberMod</td>
<td>MEMBERMODIFIER</td>
<td>static, constructor, abstract, virtual, or final if one of these attributes has been given; none if not. MEMBERMODIFIER = {none, static, constructor, abstract, virtual, final}</td>
</tr>
<tr>
<td>overrideMod</td>
<td>OVERRIDEMODIFIER</td>
<td>true, false, or undefined if the override attribute with one of these arguments was given; true if the attribute override was given without arguments was given</td>
</tr>
</tbody>
</table>
arguments was given; **true** if the attribute **override** without arguments was given; **none** if the **override** attribute was not given. **OVERRIDE** = {**none**, **true**, **false**, **undefined**}

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>staticReadBindings</strong></td>
<td><strong>STATICBINDING{}</strong></td>
<td>Map of qualified names to readable static members defined in this class (see section 5.11)</td>
</tr>
<tr>
<td><strong>staticWriteBindings</strong></td>
<td><strong>STATICBINDING{}</strong></td>
<td>Map of qualified names to writable static members defined in this class</td>
</tr>
<tr>
<td><strong>instanceReadBindings</strong></td>
<td><strong>INSTANCEBINDING{}</strong></td>
<td>Map of qualified names to readable instance members defined in this class</td>
</tr>
<tr>
<td><strong>instanceWriteBindings</strong></td>
<td><strong>INSTANCEBINDING{}</strong></td>
<td>Map of qualified names to writable instance members defined in this class</td>
</tr>
<tr>
<td><strong>instanceInitOrder</strong></td>
<td><strong>INSTANCEVARIABLE[]</strong></td>
<td>List of instance variables defined in this class in the order in which they are initialised</td>
</tr>
<tr>
<td><strong>complete</strong></td>
<td><strong>BOOLEAN</strong></td>
<td><strong>true</strong> after all members of this class have been added to this <strong>CLASS</strong> record</td>
</tr>
<tr>
<td><strong>super</strong></td>
<td><strong>CLASS</strong></td>
<td>This class’s immediate superclass or <strong>null</strong> if none</td>
</tr>
<tr>
<td><strong>prototype</strong></td>
<td><strong>OBJECT</strong></td>
<td>An object that serves as this class’s prototype for compatibility with ECMAScript 3; may be <strong>null</strong></td>
</tr>
<tr>
<td><strong>privateNamespace</strong></td>
<td><strong>NAMESPACE</strong></td>
<td>This class’s <strong>private</strong> namespace</td>
</tr>
<tr>
<td><strong>dynamic</strong></td>
<td><strong>BOOLEAN</strong></td>
<td><strong>true</strong> if this class or any of its ancestors was defined with the <strong>dynamic</strong> attribute</td>
</tr>
<tr>
<td><strong>allowNull</strong></td>
<td><strong>BOOLEAN</strong></td>
<td><strong>true</strong> if <strong>null</strong> is considered to be an instance of this class</td>
</tr>
<tr>
<td><strong>final</strong></td>
<td><strong>BOOLEAN</strong></td>
<td><strong>true</strong> if this class cannot be subclassed</td>
</tr>
<tr>
<td><strong>call</strong></td>
<td><strong>OBJECT</strong> <strong>ARGUMENTLIST</strong> <strong>PHASE</strong> <strong>OBJECT</strong></td>
<td>A procedure to call (see section 9.5) when this class is used in a call expression</td>
</tr>
<tr>
<td><strong>construct</strong></td>
<td><strong>ARGUMENTLIST</strong> <strong>PHASE</strong> <strong>OBJECT</strong></td>
<td>A procedure to call (see section 9.5) when this class is used in a <strong>new</strong> expression</td>
</tr>
</tbody>
</table>
| **implicitCoerce**   | **OBJECT** **OBJECT**   | A procedure to call when a value is assigned to a variable, parameter, or result whose type is this class. The argument to **implicitCoerce** can be any value, which may or may not be an instance of this class; the result must be an instance of this class. If the coercion is not appropriate, **implicitCoerce** should throw an
**9.1.9 Method Closures**

A `MethodClosure` tuple (see section 5.10) has the fields below and describes an instance method with a bound `this` value.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>this</td>
<td><code>OBJECT</code></td>
<td>The bound <code>this</code> value</td>
</tr>
<tr>
<td>method</td>
<td><code>INSTANCEMETHOD</code></td>
<td>The bound method</td>
</tr>
</tbody>
</table>

**9.1.10 Prototype Instances**

Prototype instances are represented as `Prototype` records (see section 5.11) with the fields below. Prototype instances contain no fixed properties.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td><code>PrototypeOpt</code></td>
<td>If this instance was created by calling <code>new</code> on a <code>prototype</code> function, the value of the function’s <code>prototype</code> property at the time of the call; <code>none</code> otherwise.</td>
</tr>
</tbody>
</table>

`DynamicProperty` record (see section 5.11) has the fields below and describes one dynamic property of one (prototype or class) instance.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td><code>STRING</code></td>
<td>This dynamic property’s name</td>
</tr>
<tr>
<td>value</td>
<td><code>OBJECT</code></td>
<td>This dynamic property’s current value</td>
</tr>
</tbody>
</table>

**9.1.11 Class Instances**

Instances of programmer-defined classes as well as of some built-in classes have the semantic domain `INSTANCE`. If the class of an instance or one of its ancestors has the dynamic attribute `instance` responds to the function call or `new` operators, then the instance is a `CallableInstanceDynamicInstance` record; otherwise, it is a `SimpleInstanceFixedInstance` record. An instance can also be an `AliasInstance` that refers to another instance. This specification uses `AliasInstances` to permit but not require an implementation to share function closures with identical behaviour.

```
INSTANCE = SimpleInstance [] CallableInstance [] AliasInstanceInstance = NonAliasInstance [] AliasInstance
```
NOTE  Instances of some built-in classes are represented as described in sections 9.1.1 through 9.1.10 rather than as \texttt{INSTANCE} records. This distinction is made for convenience in specifying the language's behaviour and is invisible to the programmer.

Instances of non-dynamic classes are represented as \texttt{FIXEDINSTANCE} records (see section 5.11) with the fields below. These instances can contain only fixed properties.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>\texttt{CLASS}</td>
<td>This instance's type</td>
</tr>
<tr>
<td>call</td>
<td>\texttt{OBJECT} \texttt{ARGUMENTLIST} \texttt{ENVIRONMENT} \texttt{PHASE} \texttt{OBJECT}</td>
<td>A procedure to call when this instance is used in a call expression. The procedure takes an \texttt{OBJECT} (the \texttt{this} value), an \texttt{ARGUMENTLIST} (see section 9.5), a lexical \texttt{ENVIRONMENT}, and a \texttt{PHASE} (see section 9.6) and produces an \texttt{OBJECT} result</td>
</tr>
<tr>
<td>construct</td>
<td>\texttt{ARGUMENTLIST} \texttt{ENVIRONMENT} \texttt{PHASE} \texttt{OBJECT}</td>
<td>A procedure to call when this instance is used in a new expression. The procedure takes an \texttt{ARGUMENTLIST} (see section 9.5), a lexical \texttt{ENVIRONMENT}, and a \texttt{PHASE} (see section 9.6) and produces an \texttt{OBJECT} result</td>
</tr>
<tr>
<td>env</td>
<td>\texttt{ENVIRONMENT}</td>
<td>The environment to pass to the call or construct procedure</td>
</tr>
<tr>
<td>typeofString</td>
<td>\texttt{STRING}</td>
<td>A string to return if \texttt{typeof} is invoked on this instance</td>
</tr>
<tr>
<td>slots</td>
<td>\texttt{SLOT[]}</td>
<td>A set of slots that hold this instance's fixed property values</td>
</tr>
</tbody>
</table>

Instances which do not respond to the function call or \texttt{new} operators are represented as \texttt{SIMPLEINSTANCE} records (see section 5.11) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>\texttt{CLASS}</td>
<td>This instance’s type</td>
</tr>
<tr>
<td>typeofString</td>
<td>\texttt{STRING}</td>
<td>A string to return if \texttt{typeof} is invoked on this instance</td>
</tr>
<tr>
<td>slots</td>
<td>\texttt{SLOT[]}</td>
<td>A set of slots that hold this instance’s fixed property values</td>
</tr>
<tr>
<td>dynamicProperties</td>
<td>\texttt{DYNAMICPROPERTY[]} \texttt{[fixed]}</td>
<td>A set of this instance’s dynamic properties if this instance’s class is a dynamic class; \texttt{fixed} if not</td>
</tr>
</tbody>
</table>

Instances of dynamic classes which respond to the function call or \texttt{new} operators—are represented as \texttt{CALLABLEINSTANCE} \texttt{DYNAMICINSTANCE} records (see section 5.11) with the fields below. These instances can contain fixed and dynamic properties.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>\texttt{CLASS}</td>
<td>This instance’s type</td>
</tr>
<tr>
<td>typeofString</td>
<td>\texttt{STRING}</td>
<td>A string to return if \texttt{typeof} is invoked on this instance</td>
</tr>
<tr>
<td>slots</td>
<td>\texttt{SLOT[]}</td>
<td>A set of slots that hold this instance’s fixed property values</td>
</tr>
<tr>
<td>dynamicProperties</td>
<td>\texttt{DYNAMICPROPERTY[]} \texttt{[fixed]}</td>
<td>A set of this instance’s dynamic properties if this instance’s class is a dynamic class; \texttt{fixed} if not</td>
</tr>
<tr>
<td>call</td>
<td>\texttt{OBJECT} \texttt{ARGUMENTLIST} \texttt{ENVIRONMENT} \texttt{PHASE} \texttt{OBJECT}</td>
<td>A procedure to call when this instance is used in a call expression. The procedure takes an \texttt{OBJECT} (the \texttt{this} value), an \texttt{ARGUMENTLIST} (see section 9.5), a lexical \texttt{ENVIRONMENT}, and a \texttt{PHASE} (see section 9.6) and produces an \texttt{OBJECT} result</td>
</tr>
<tr>
<td>construct</td>
<td>\texttt{ARGUMENTLIST} \texttt{ENVIRONMENT} \texttt{PHASE} \texttt{OBJECT}</td>
<td>A procedure to call when this instance is used in a \texttt{new} expression. The procedure takes an \texttt{ARGUMENTLIST} (see section 9.5), a lexical \texttt{ENVIRONMENT} and a \texttt{PHASE} (see section 9.6) and produces an \texttt{OBJECT} result</td>
</tr>
</tbody>
</table>
section 9.5), a lexical \texttt{ENVIRONMENT}, and a \texttt{Phase} (see section 9.6) and produces an \texttt{Object} result.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\texttt{env} & \texttt{ENVIRONMENT} & The environment to pass to the \texttt{call} or \texttt{construct} procedure \\
\hline
\end{tabular}
\end{table}

\texttt{AliasInstance} records (see section 5.11) with the fields below represent aliases to existing instances. An \texttt{AliasInstance} behaves just like its original instance except that it supplies a different environment to the \texttt{call} and \texttt{construct} procedures. In practice, an implementation would likely only use \texttt{AliasInstances} if it can prove that supplying the different environment to the \texttt{call} and \texttt{construct} procedures has no visible consequences, so it could optimise out the \texttt{AliasInstance} altogether.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Field} & \textbf{Contents} & \textbf{Note} \\
\hline
\texttt{original} & \texttt{CallableInstanceNonAliasInstance} & This original instance being aliased \\
\texttt{env} & \texttt{ENVIRONMENT} & The environment to pass to the \texttt{call} or \texttt{construct} procedure \\
\hline
\end{tabular}
\end{table}

\subsection{Open Instances}

An \texttt{OpenInstance} record (see section 5.11) has the fields below. It is not an instance in itself but creates an instance when instantiated with an environment. \texttt{OpenInstance} records represent functions with variables inherited from their enclosing environments; supplying the environment turns such a function into a \texttt{CallableInstance\_callable\_instance}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Field} & \textbf{Contents} & \textbf{Note} \\
\hline
\texttt{type} & \texttt{CLASS} & \\
\texttt{typeofString} & \texttt{STRING} & \\
\texttt{defaultSlots\_instantiate} & \texttt{Slot[]}\texttt{ENVIRONMENT}[]\texttt{NonAliasInstance} & A procedure to call to supply an environment and obtain a fresh instance. A list of the default values of the generated \texttt{CallableInstance\_s} slots \\
\texttt{buildPrototype} & \texttt{BOOLEAN} & If true, the generated \texttt{CallableInstance} gets a separate \texttt{prototype} slot with its own \texttt{prototype} object \\
\texttt{call} & \texttt{OBJECT}[]\texttt{ARGUMENTLIST}[]\texttt{ENVIRONMENT}[]\texttt{PHASE}[]\texttt{OBJECT} & Values to be transferred into the generated \texttt{CallableInstance\_s} corresponding fields \\
\texttt{construct} & \texttt{ARGUMENTLIST}[]\texttt{ENVIRONMENT}[]\texttt{PHASE}[]\texttt{OBJECT} & \\
\texttt{cache} & \texttt{CallableInstanceNonAliasInstance}[]\texttt{\{none\}} & Optional cached value of the last instantiation. This cache serves only to precisely specify the closure sharing optimization and would likely not be present in any actual implementation. \\
\hline
\end{tabular}
\end{table}

\subsection{Slots}

A \texttt{Slot} record (see section 5.11) has the fields below and describes the value of one fixed property of one instance.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Field} & \textbf{Contents} & \textbf{Note} \\
\hline
\texttt{id} & \texttt{INSTANCE\_VAR} & The instance variable whose value this slot carries \\
\texttt{value} & \texttt{OBJECT\_U} & This fixed property\_s current value; \texttt{uninitialised} if the fixed property is an uninitialised constant \\
\hline
\end{tabular}
\end{table}
9.1.12 Packages

Programmer-visible packages are represented as PACKAGE records (see section 5.11) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticReadBindings</td>
<td>STATIC_BINDING{}</td>
<td>Map of qualified names to readable members defined in this package</td>
</tr>
<tr>
<td>staticWriteBindings</td>
<td>STATIC_BINDING{}</td>
<td>Map of qualified names to writable members defined in this package</td>
</tr>
<tr>
<td>internalNamespace</td>
<td>NAMESPACE</td>
<td>This package’s internal namespace</td>
</tr>
</tbody>
</table>

9.1.13 Global Objects

Programmer-visible global objects are represented as GLOBAL records (see section 5.11) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticReadBindings</td>
<td>STATIC_BINDING{}</td>
<td>Map of qualified names to readable members defined in this global object</td>
</tr>
<tr>
<td>staticWriteBindings</td>
<td>STATIC_BINDING{}</td>
<td>Map of qualified names to writable members defined in this global object</td>
</tr>
<tr>
<td>internalNamespace</td>
<td>NAMESPACE</td>
<td>This global object’s internal namespace</td>
</tr>
<tr>
<td>dynamicProperties</td>
<td>DYNAMIC_PROPERTY{}</td>
<td>A set of this global object’s dynamic properties</td>
</tr>
</tbody>
</table>

9.2 Objects with Limits

A LIMITEDINSTANCE tuple (see section 5.10) represents an intermediate result of a super or super(expr) subexpression. It has the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance</td>
<td>INSTANCE</td>
<td>The value of expr to which the super subexpression was applied; if expr wasn’t given, defaults to the value of this. The value of instance is always an instance of the limit class or one of its descendants.</td>
</tr>
<tr>
<td>limit</td>
<td>CLASS</td>
<td>The class inside which the super subexpression was applied</td>
</tr>
</tbody>
</table>

Member and operator lookups on a LIMITEDINSTANCE value will only find members and operators defined on proper ancestors of limit.

OBJOPTIONALLIMIT is the result of a subexpression that can produce either an OBJECT or a LIMITEDINSTANCE:

OBJOPTIONALLIMIT = OBJECT □ LIMITEDINSTANCE

9.3 References

A REFERENCE (also known as an lvalue in the computer literature) is a temporary result of evaluating some subexpressions. It is a place where a value may be read or written. A REFERENCE may serve as either the source or destination of an assignment.

REFERENCE = LEXICALREFERENCE □ DOTREFERENCE □ BRACKETREFERENCE;

Some subexpressions evaluate to an OBJORRef, which is either an OBJECT (also known as an rvalue) or a REFERENCE. Attempting to use an OBJORRef that is an rvalue as the destination of an assignment produces an error.

OBJORRef = OBJECT □ REFERENCE

A LEXICALREFERENCE tuple (see section 5.10) has the fields below and represents an lvalue that refers to a variable with one of a given set of qualified names. LEXICALREFERENCE tuples arise from evaluating identifiers a and qualified identifiers q::a.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
</table>

...
9.4 Function Support

There are three kinds of functions: normal functions, getters, and setters. The FunctionKind semantic domain encodes the kind:

\[
\text{FunctionKind} = \{\text{normal, get, set}\}
\]

A Signature tuple (see section 5.10) has the fields below and represents the type signature of a function.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>positional</td>
<td>PARAMETER[,]</td>
<td>List of the required positional parameters</td>
</tr>
<tr>
<td>optionalPositional</td>
<td>PARAMETER[,]</td>
<td>List of the optional positional parameters, which follow the required positional parameters</td>
</tr>
<tr>
<td>optionalNamed</td>
<td>NAMEDPARAMETER{}</td>
<td>Set of the types and names of the optional named parameters</td>
</tr>
<tr>
<td>rest</td>
<td>PARAMETER [none]</td>
<td>The parameter for collecting any extra arguments that may be passed or null if no extra arguments are allowed</td>
</tr>
<tr>
<td>restAllowsNames</td>
<td>BOOLEAN</td>
<td>true if the extra arguments may be named</td>
</tr>
<tr>
<td>returnType</td>
<td>CLASS</td>
<td>The type of this function’s result</td>
</tr>
</tbody>
</table>

A Parameter tuple (see section 5.10) has the fields below and represents the signature of one unnamed parameter.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localName</td>
<td>QUALIFIEDNAMEOPT</td>
<td>Name of the local variable that will hold this parameter’s value</td>
</tr>
</tbody>
</table>
A NamedParameter tuple (see section 5.10) has the fields below and represents the signature of one named parameter.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localName</td>
<td>QualifiedNameOpt</td>
<td>Name of the local variable that will hold this parameter’s value</td>
</tr>
<tr>
<td>type</td>
<td>CLASS</td>
<td>This parameter’s type</td>
</tr>
<tr>
<td>name</td>
<td>STRING</td>
<td>This parameter’s external name</td>
</tr>
</tbody>
</table>

9.5 Argument Lists

An ArgumentList tuple (see section 5.10) has the fields below and describes the arguments (other than this) passed to a function.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>positional</td>
<td>OBJECT[]</td>
<td>Ordered list of positional arguments</td>
</tr>
<tr>
<td>named</td>
<td>NAMEDARGUMENT[]</td>
<td>Set of named arguments</td>
</tr>
</tbody>
</table>

9.6 Modes of expression evaluation

Expressions can be evaluated in either run mode or compile mode. In run mode all operations are allowed. In compile mode, operations are restricted to those that cannot use or produce side effects, access non-constant variables, or call programmer-defined functions.

The semantic domain Phase consists of the tags compile and run representing the two modes of expression evaluation:

\[
\text{Phase} = \{\text{compile, run}\}
\]

9.7 Contexts

A Context tuple (see section 5.10) carries static information about a particular point in the source program and has the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>strict</td>
<td>BOOLEAN</td>
<td>true if strict mode (see *****) is in effect</td>
</tr>
<tr>
<td>openNamespaces</td>
<td>NAMESPACE[]</td>
<td>The set of namespaces that are open at this point. The public namespace is always a member of this set.</td>
</tr>
</tbody>
</table>

9.8 Labels

A Label is a label that can be used in a break or continue statement. The label is either a string or the special tag default. Strings represent labels named by identifiers, while default represents the anonymous label.

\[
\text{Label} = \text{STRING} \{\text{default}\}
\]

A JumpTargets tuple (see section 5.10) describes the sets of labels that are valid destinations for break or continue statements at a point in the source code. A JumpTargets tuple has the fields below.
9.9 Environments

Environments contain the bindings that are visible from a given point in the source code. An Environment is a list of two or more frames. Each frame corresponds to a scope. More specific frames are listed first—each frame’s scope is directly contained in the following frame’s scope. The last frame is always the SystemFrame. The next-to-last frame is always a Package or Global frame.

```
ENVIRONMENT = FRAME[]
```

The semantic domain ENVIRONMENT consists of all environments as well as the tag inaccessible which denotes that an environment is not available at this time:
```
ENVIRONMENT[] = {inaccessible};
```

9.9.1 Frames

A frame contains bindings defined at a particular scope in a program. A frame is either the top-level system frame, a global object, a package, a function parameter frame, a class, or a block frame:

```
FRAME = SYSTEMFRAME GLOBAL PACKAGE PARAMETERFRAME CLASS BLOCKFRAME;
```

Some frames can be marked either singular or plural. A singular frame contains the current values of variables and other definitions. A plural frame is a template for making singular frames — a plural frame contains placeholders for mutable variables and definitions as well as the actual values of compile-time constant definitions. The static analysis done by Validate generates singular frames for the system frame, global object, and any blocks, classes, or packages directly contained inside another singular frame; all other frames are plural during static analysis and are instantiated to make singular frames by Eval.

The system frame, global objects, packages, and classes are always singular. Function and block frames can be either singular or plural.

Plurality is the semantic domain of the two tags singular and plural:
```
PLURALITY = {singular, plural}
```

9.9.1.1 System Frame

The top-level frame containing predefined constants, functions, and classes is represented as a SystemFrame record (see section 5.11) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticReadBindings</td>
<td>STATICBINDING{}</td>
<td>Map of qualified names to readable definitions in this frame</td>
</tr>
<tr>
<td>staticWriteBindings</td>
<td>STATICBINDING{}</td>
<td>Map of qualified names to writable definitions in this frame</td>
</tr>
</tbody>
</table>

9.9.1.2 Function Parameter Frames

Frames holding bindings for invoked functions are represented as ParameterFrame records (see section 5.11) with the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>staticReadBindings</td>
<td>STATICBINDING{}</td>
<td>Map of qualified names to readable definitions in this function</td>
</tr>
<tr>
<td>staticWriteBindings</td>
<td>STATICBINDING{}</td>
<td>Map of qualified names to writable definitions in this function</td>
</tr>
<tr>
<td>plurality</td>
<td>PLURALITY</td>
<td>See section 9.9.1</td>
</tr>
<tr>
<td>this</td>
<td>OBJECTIOP</td>
<td>The value of this; none if this function doesn’t define this; inaccessible if this function defines this, but the value is not available.</td>
</tr>
</tbody>
</table>
9.9.2 Static Bindings

A \texttt{STATICBIND} tuple (see section 5.10) has the fields below and describes the member to which one qualified name is bound in a frame. Multiple qualified names may be bound to the same member in a frame, but a qualified name may not be bound to multiple members in a frame (except when one binding is for reading only and the other binding is for writing only).

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname</td>
<td>\texttt{QUALIFIEDNAME}</td>
<td>The qualified name bound by this binding</td>
</tr>
<tr>
<td>content</td>
<td>\texttt{STATICMEMBER}</td>
<td>The member to which this qualified name was bound</td>
</tr>
<tr>
<td>explicit</td>
<td>\texttt{BOOLEAN}</td>
<td>\texttt{true} if this binding should not be imported into the global scope by an \texttt{import} statement</td>
</tr>
</tbody>
</table>

A static member is either \texttt{forbidden}, a variable, a hoisted variable, a constructor method, a getter, or a setter:

\[
\texttt{STATICMEMBER} = \{\texttt{forbidden}\} \ OR \ \texttt{VARIABLE} \ OR \ \texttt{HOISTEDVAR} \ OR \ \texttt{CONSTRUCTORMETHOD} \ OR \ \texttt{GETTER} \ OR \ \texttt{SETTER};
\]

\[
\texttt{STATICMEMBOPOPT} = \texttt{STATICMEMBER} \ OR \ \{\texttt{none}\};
\]

A \texttt{forbidden} static member is one that must not be accessed because there exists a definition for the same qualified name in a more local block.

A \texttt{VARIABLE} record (see section 5.11) has the fields below and describes one variable or constant definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>\texttt{VARIABLETYPE}</td>
<td>Type of values that may be stored in this variable (see below)</td>
</tr>
<tr>
<td>value</td>
<td>\texttt{VARIABLEVALUE}</td>
<td>This variable’s current value; \texttt{future} if the variable has not been declared yet; \texttt{uninitialised} if the variable must be written before it can be read</td>
</tr>
<tr>
<td>immutable</td>
<td>\texttt{BOOLEAN}</td>
<td>\texttt{true} if this variable’s value may not be changed once set</td>
</tr>
</tbody>
</table>

A variable’s type can be either a class, \texttt{inaccessible}, or a semantic procedure that takes no parameters and will compute a class on demand; such procedures are used instead of \texttt{CLASSes} for types of variables in situations where the type expression can contain forward references and shouldn’t be evaluated until it is needed.

\[
\texttt{VARIABLETYPE} = \texttt{CLASS} \ OR \ \{\texttt{inaccessible}\} \ OR \ \{\texttt{inaccessible, uninitialised}\};
\]

A \texttt{VARIABLEVALUE} record (see section 5.11) has the fields below and describes one hoisted variable.

\[
\texttt{VARIABLEVALUE} = \texttt{OBJECT} \ OR \ \{\texttt{inaccessible, uninitialised}\} \ OR \ \texttt{OPENINSTANCE} \ OR \ \{\texttt{inaccessible, uninitialised}\};
\]

A \texttt{HOISTEDVAR} record (see section 5.11) has the fields below and describes one hoisted variable.
### Field Contents Note

<table>
<thead>
<tr>
<th>value</th>
<th>OBJECT</th>
<th>OPEN_INSTANCE</th>
<th>This variable’s current value; may be an open (unclosed) function at compile time</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasFunctionInitialiser</td>
<td>BOOLEAN</td>
<td>true</td>
<td>if this variable was created by a function statement</td>
</tr>
</tbody>
</table>

A **CONSTRUCTOR METHOD** record (see section 5.11) has the field below and describes one constructor definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>INSTANCE</td>
<td>This constructor itself (a callable object)</td>
</tr>
</tbody>
</table>

A **GETTER** record (see section 5.11) has the fields below and describes one static getter definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>CLASS</td>
<td>The type of the value read from this getter</td>
</tr>
<tr>
<td>call</td>
<td>ENVIRONMENT</td>
<td>PHASE</td>
</tr>
<tr>
<td>env</td>
<td>ENVIRONMENT</td>
<td>I</td>
</tr>
</tbody>
</table>

A **SETTER** record (see section 5.11) has the fields below and describes one static setter definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>CLASS</td>
<td>The type of the value written by this setter</td>
</tr>
<tr>
<td>call</td>
<td>OBJECT</td>
<td>ENVIRONMENT</td>
</tr>
<tr>
<td>env</td>
<td>ENVIRONMENT</td>
<td>I</td>
</tr>
</tbody>
</table>

### 9.9.3 Instance Bindings

An **INSTANCE BINDING** tuple (see section 5.10) has the fields below and describes the binding of one qualified name to an instance member of a class. Multiple qualified names may be bound to the same instance member in a class, but a qualified name may not be bound to multiple instance members in a class (except when one binding is for reading only and the other binding is for writing only).

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>qname</td>
<td>QUALIFIED NAME</td>
<td>The qualified name bound by this binding</td>
</tr>
<tr>
<td>content</td>
<td>INSTANCE MEMBER</td>
<td>The member to which this qualified name was bound</td>
</tr>
</tbody>
</table>

An instance member is either an instance variable, an instance method, or an instance accessor:

```
INSTANCE MEMBER = INSTANCE VARIABLE | INSTANCE METHOD | INSTANCE GETTER | INSTANCE SETTER;
```

```
INSTANCE MEMBER OPT = INSTANCE MEMBER | {none};
```

An **INSTANCE VARIABLE** record (see section 5.11) has the fields below and describes one instance variable or constant definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>CLASS</td>
<td>Type of values that may be stored in this variable</td>
</tr>
<tr>
<td>evallInitialValue</td>
<td>OBJECT OPT</td>
<td>A function that computes this variable’s initial value</td>
</tr>
<tr>
<td>immutable</td>
<td>BOOLEAN</td>
<td>true if this variable’s value may not be changed once set</td>
</tr>
</tbody>
</table>
An `INSTANCE_METHOD` record (see section 5.11) has the fields below and describes one instance method definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td><code>INSTANCE</code> {abstract}</td>
<td>This method itself (a callable object); <code>abstract</code> if this method is abstract</td>
</tr>
<tr>
<td>signature</td>
<td><code>SIGNATURE</code></td>
<td>This method’s signature</td>
</tr>
<tr>
<td>final</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if this member may not be overridden in subclasses</td>
</tr>
</tbody>
</table>

An `INSTANCE_GETTER` record (see section 5.11) has the fields below and describes one instance getter definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td><code>CLASS</code></td>
<td>The type of the value read from this getter</td>
</tr>
<tr>
<td>call</td>
<td><code>OBJECT</code> <code>ENVIRONMENT</code> <code>PHASE</code></td>
<td>A procedure to call to read the value, passing it the <code>this</code> value, the <code>env</code> field below, and the current mode of expression evaluation</td>
</tr>
<tr>
<td>env</td>
<td><code>ENVIRONMENT</code></td>
<td>The environment bound to this getter</td>
</tr>
<tr>
<td>final</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if this member may not be overridden in subclasses</td>
</tr>
</tbody>
</table>

An `INSTANCE_SETTER` record (see section 5.11) has the fields below and describes one instance setter definition.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td><code>CLASS</code></td>
<td>The type of the value written by this setter</td>
</tr>
<tr>
<td>call</td>
<td><code>OBJECT</code> <code>OBJECT</code> <code>ENVIRONMENT</code> <code>PHASE</code> <code>()</code></td>
<td>A procedure to call to write the value, passing it the new value, the <code>this</code> value, the <code>env</code> field below, and the current mode of expression evaluation</td>
</tr>
<tr>
<td>env</td>
<td><code>ENVIRONMENT</code></td>
<td>The environment bound to this setter</td>
</tr>
<tr>
<td>final</td>
<td><code>BOOLEAN</code></td>
<td><code>true</code> if this member may not be overridden in subclasses</td>
</tr>
</tbody>
</table>

## 10 Data Operations

This chapter describes core algorithms defined on the values in chapter 9. The algorithms here are not ECMA Script language constructs themselves; rather, they are called as subroutines in computing the effects of the language constructs presented in later chapters. The algorithms are optimised for ease of presentation and understanding rather than speed, and implementations are encouraged to implement these algorithms more efficiently as long as the observable behaviour is as described here.

### 10.1 Numeric Utilities

`unsignedWrap32(i)` returns `i` converted to a value between 0 and $2^{32} - 1$ inclusive, wrapping around modulo $2^{32}$ if necessary.

```plaintext
proc unsignedWrap32(i: INTEGER): {0 ... 2^32 - 1}
    return bitwiseAnd(i, 0xFFFFFFFF)
end proc;
```

`signedWrap32(i)` returns `i` converted to a value between $-2^{31}$ and $2^{31} - 1$ inclusive, wrapping around modulo $2^{32}$ if necessary.

```plaintext
proc signedWrap32(i: INTEGER): {-2^{31} ... 2^{31} - 1}
    j: INTEGER  □ bitwiseAnd(i, 0xFFFFFFFF);
    if j ≥ 2^{31} then j □ j - 2^{32} end if;
    return j
end proc;
```
unsignedWrap64(i) returns \( i \) converted to a value between 0 and \( 2^{64} - 1 \) inclusive, wrapping around modulo \( 2^{64} \) if necessary.

```plaintext
proc unsignedWrap64(i: INTEGER): \{0 ... 2^{64} - 1\}
    return bitwiseAnd(i, 0xFFFFFFFFFFFFFFFF)
end proc;
```

signedWrap64(i) returns \( i \) converted to a value between \(-2^{63} \) and \( 2^{63} - 1 \) inclusive, wrapping around modulo \( 2^{64} \) if necessary.

```plaintext
proc signedWrap64(i: INTEGER): \{-2^{63} ... 2^{63} - 1\}
    j: INTEGER = bitwiseAnd(i, 0xFFFFFFFFFFFFFFFF);
    if \( j \geq 2^{63} \) then \( j = 2^{64} \) end if;
    return j
end proc;
```

proc truncateToInt(x: GENERALNUMBER): INTEGER
    case x of
        \{+\infty_{32}, +\infty_{64}, -\infty_{32}, -\infty_{64}, NaN32, NaN64\} do return 0;
        FINITEFLOAT32 do return truncateFiniteFloat32(x);
        FINITEFLOAT64 do return truncateFiniteFloat64(x);
        LONG do return x.value
    end case
end proc;

proc checkInteger(x: GENERALNUMBER): INTEGEROPT
    case x of
        \{NaN32, NaN64, +\infty_{32}, +\infty_{64}, -\infty_{32}, -\infty_{64}, zero\} do return none;
        \{+zero, +zero, -zero, -zero\} do return 0;
        LONG do return x.value;
        NONZEROFINITEFLOAT32, NONZEROFINITEFLOAT64 do
            r: RATIONAL = x.value;
            if r = INTEGER then return none end if;
            return r
        end case
end proc;

proc integerToLong(i: INTEGER): GENERALNUMBER
    if \(-2^{63} \leq i \leq 2^{63} - 1\) then return INTEGER value: i
    elif \( 2^{63} \leq i \leq 2^{64} - 1\) then return ULONG value: i
    else return realToFloat64(i)
    end if
end proc;

proc integerToULong(i: INTEGER): GENERALNUMBER
    if \( 0 \leq i \leq 2^{64} - 1\) then return ULONG value: i
    elsif \(-2^{63} \leq i \leq -1\) then return LONG value: i
    else return realToFloat64(i)
    end if
end proc;

proc rationalToLong(q: RATIONAL): GENERALNUMBER
    if q = INTEGER then return integerToLong(q)
    elsif \( |q| \leq 2^{53}\) then return realToFloat64(q)
    elsif \( q < -2^{63} \) or \( q \geq 2^{64} \) then return realToFloat64(q)
    else
        Let \( i \) be the integer closest to \( q \). If \( q \) is halfway between two integers, pick \( i \) so that it is even.
        Note that \(-2^{63} \leq i \leq 2^{64} - 1\)
        if \( i < 2^{63} \) then return LONG value: i else return ULONG value: i end if
    end if
end proc;
proc rationalToULong(q: RATIONAL): GENERALNUMBER
  if q [] INTEGER then return integerToULong(q)
  elsif q ≤ 2\(^{63}\) then return realToFloat64(q)
  elsif q < -2\(^{63}\) - 1/2 or q ≥ 2\(^{64}\) - 1/2 then return realToFloat64(q)
  else
    Let i be the integer closest to q. If q is halfway between two integers, pick i so that it is even.
    Note that -2\(^{63}\) ≤ i ≤ 2\(^{64}\) - 1
    if i ≥ 0 then return ULONG\[value: i] else return LONG\[value: i]\end if
  end if
end proc;

proc toRational(x: FINITEGENERALNUMBER): RATIONAL
  case x of
    \{+zero\_{f32}, +zero\_{f64}, -zero\_{f32}, -zero\_{f64}\} do return 0;
    NONZEROFINITEFLOAT32 \ nonzero\_{finite}FLOAT32 \ nonzero\_{finite}FLOAT64 \ LONG \ ULONG do return x.value
  end case
end proc;

proc toFloat64(x: GENERALNUMBER): FLOAT64
  case x of
    LONG \ ULONG do return realToFloat64(x.value);
    FLOAT32 do return float32ToFloat64(x);
    FLOAT64 do return x
  end case
end proc;

ORDER is the four-element semantic domain of tags representing the possible results of a floating-point comparison:
ORDER = \{less, equal, greater, unordered\};

proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
  if x \ \{NaN\_{f32}, NaN\_{f64}\} or y \ \{NaN\_{f32}, NaN\_{f64}\} then return unordered
  elsif x \ \{+\infty\_{f32}, +\infty\_{f64}\} and y \ \{+\infty\_{f32}, +\infty\_{f64}\} then return equal
  elsif x \ \{-\infty\_{f32}, -\infty\_{f64}\} and y \ \{-\infty\_{f32}, -\infty\_{f64}\} then return equal
  elsif x \ \{+\infty\_{f32}, +\infty\_{f64}\} or y \ \{-\infty\_{f32}, -\infty\_{f64}\} then return greater
  elsif x \ \{-\infty\_{f32}, -\infty\_{f64}\} or y \ \{+\infty\_{f32}, +\infty\_{f64}\} then return less
  else
    xr: RATIONAL \ toRational(x);
    yr: RATIONAL \ toRational(y);
    if xr < yr then return less
    elsif xr > yr then return greater
    else return equal
  end if
end proc;

10.2 Object Utilities

proc resolveAlias(o: INSTANCE): SIMPLEINSTANCE \ CALLABLEINSTANCE
  case o of
    ALIASINSTANCE do return o.original;
    SIMPLEINSTANCE \ CALLABLEINSTANCE do return o
  end case
end proc;

10.2.1 objectType

objectType(o) returns an OBJECT o’s most specific type.
proc objectType(o: Object): Class
    case o of
        UNDEFINED do return undefinedClass;
        NULL do return nullClass;
        BOOLEAN do return booleanClass;
        LONG do return longClass;
        ULONG do return uLongClass;
        FLOAT32 do return floatClass;
        FLOAT64 do return numberClass;
        CHARACTER do return characterClass;
        STRING do return stringClass;
        NAMESPACE do return namespaceClass;
        COMPOUNDATTRIBUTE do return attributeClass;
        CLASS do return classClass;
        METHODCLOSURE do return functionClass;
        PROTOTYPE do return prototypeClass;
        INSTANCE do return resolveAlias(o).type;
    end case
end proc;

10.2.2 hasType

There are two tests for determining whether an object o is an instance of class c. The first, hasType, is used for the purposes of method dispatch and helps determine whether a method of c can be called on o. The second, relaxedHasType, determines whether o can be stored in a variable of type c without conversion.

hasType(o, c) returns true if o is an instance of class c (or one of c’s subclasses). It considers null to be an instance of the classes Null and Object only.

proc hasType(o: Object, c: Class): Boolean
    return isAncestor(c, objectType(o))
end proc;

relaxedHasType(o, c) returns true if o is an instance of class c (or one of c’s subclasses) but considers null to be an instance of the classes Null, Object, and all other non-primitive classes.

proc relaxedHasType(o: Object, c: Class): Boolean
    t: Class = objectType(o);
    return isAncestor(c, t) or (o = null and c allowNull)
end proc;

10.2.3 toBoolean

toBoolean(o, phase) coerces an object o to a Boolean. If phase is compile, only compile-time conversions are permitted.

proc toBoolean(o: Object, phase: Phase): Boolean
    case o of
        UNDEFINED do return false;
        NULL o;
        LONG do return o.value ≠ 0;
        FLOAT32 o do return o.e ∈ [+zero, −zero, NaN] ;
        FLOAT64 o do return o.e ∈ [+zero, −zero, NaN] ;
        STRING do return o ≠ "";
        CHARACTER do return o ≠ "";
        NAMESPACE do return o ≠ "";
        COMPONUDATTRIBUTE do return o ≠ "";
        CLASS do return o ≠ "";
        METHODCLOSURE do return o ≠ "";
        PROTOTYPE do return o ≠ "";
        INSTANCE do return o ≠ "";
        PACKAGE do return o ≠ "";
        GLOBAL do
        return true
    end case
end proc;
10.2.4 toGeneralNumber

toGeneralNumber(o, phase) coerces an object o to a GENERALNUMBER. If phase is compile, only compile-time conversions are permitted.

    proc toGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
    case o of
        UDEFINED do return NaN64;
        NULL ☐ {false} do return +zero64;
        {true} do return 1.064;
        GENERALNUMBER o return o;
        CHARACTER ☐ STRING do ????;
        NAMESPACE ☐ COMPOUNDATEtribute ☐ CLASS ☐ METHODCLOSURE ☐ PACKAGE ☐ GLOBAL do
            throw badValueError;
        PROTOTYPE ☐ INSTANCE do ????
    end case
    end proc;

10.2.5 toString

toString(o, phase) coerces an object o to a string. If phase is compile, only compile-time conversions are permitted.

    proc toString(o: OBJECT, phase: PHASE): STRING
    case o of
        UDEFINED do return “undefined”;
        NULL do return “null”;
        {false} do return “false”;
        {true} do return “true”;
        LONG ☐ ULONG do return integerToString(o.value);
        FLOAT32 do return float32ToString(o);
        FLOAT64 do return float64ToString(o);
        CHARACTER do return [o];
        STRING do return o;
        NAMESPACE do ????;
        COMPOUNDATEtribute do ????;
        CLASS do ????;
        METHODCLOSURE do ????;
        PROTOTYPE ☐ INSTANCE do ????;
        PACKAGE ☐ GLOBAL do ????
    end case
    end proc;

integerToString(i) converts an integer i to a string of one or more decimal digits. If i is negative, the string is preceded by a minus sign.

    proc integerToString(i: INTEGER): STRING
    if i < 0 then return ‘-’ ☐ integerToString(i) end if;
    q: INTEGER ☐ i/10
    r: INTEGER ☐ i % 10;
    c: CHARACTER ☐ codeToCharacter(r + characterToCode(‘0’));
    if q = 0 then return [c] else return integerToString(q) ☐ [c] end if
    end proc;

integerToStringWithSign(i) is the same as integerToString(i) except that the resulting string always begins with a plus or minus sign.

    proc integerToStringWithSign(i: INTEGER): STRING
    if i ≥ 0 then return ‘+’ ☐ integerToString(i)
    else return ‘-’ ☐ integerToString(-i)
    end if
    end proc;
float32ToString(x) converts a FLOAT32 x to a string using fixed-point notation if the absolute value of x is between 10⁻⁶ inclusive and 10²¹ exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a FLOAT32 value would result in the same value x (except that –zero₃₂ would become +zero₃₂).

proc float32ToString(x: FLOAT32): STRING
  case x of
    {NaN₃₂} do return “NaN”;
    {+zero₃₂, –zero₃₂} do return “0”;
    {+∞₃₂} do return “Infinity”;  
    {–∞₃₂} do return “-Infinity”;  
    NONZEROFINITEFLOAT32 do
      r: RATIONAL [] x.value;
      if r < 0 then return “-” ⊕ float32ToString(float32Negate(x))
      else
        Let n, k, and s be integers such that k ≥ 1, 10ᵏ⁻¹ ≤ s ≤ 10ᵏ, realToFloat32(s[1]₀ⁿ⁻ₖ) = x, and k is as small as possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria. When there are multiple possibilities for s according to the rules above, implementations are encouraged but not required to select the one according to the following rules: Select the value of s for which s[1]₀ⁿ⁻ₖ is closest in value to r; if there are two such possible values of s, choose the one that is even.

      digits: STRING [] integerToString(s);
      if k ≤ n ≤ 21 then return digits ⊕ repeat(‘0’, n – k)
      elseif 0 < n ≤ 21 then return digits[0 ... n − 1] ⊕ “.” ⊕ digits[n ...]  
      elseif –6 < n ≤ 0 then return “0.” ⊕ repeat(‘0’, –n) ⊕ digits  
      else
        mantissa: STRING;
        if k = 1 then mantissa [] digits
        else mantissa [] digits[0 ... 0] ⊕ “.” ⊕ digits[1 ...]
        end if;
        return mantissa ⊕ “e” ⊕ integerToStringWithSign(n − 1)
      end if;
    end case
  end proc;

float64ToString(x) converts a FLOAT64 x to a string using fixed-point notation if the absolute value of x is between 10⁻⁶ inclusive and 10²¹ exclusive and exponential notation otherwise. The result has the fewest significant digits possible while still ensuring that converting the string back into a FLOAT64 value would result in the same value x (except that –zero₆₄ would become +zero₆₄).
```ecmascript
proc float64ToString(x: FLOAT64): STRING
  case x of
    { NaN64 } do return "NaN";
    {+zero64, -zero64 } do return "0";
    {+∞64 } do return "Infinity";
    {-∞64 } do return "-Infinity";
    NONZEROFINITEFLOAT64 do
      r: RATIONAL = x.value;
      if r < 0 then return "-" ⊕ float64ToString(float64Negate(x))
      else
        Let n, k, and s be integers such that k ≥ 1, 10^{k-1} ≤ s < 10^k, realToFloat64(s[10^{k-k}]) = x, and k is as small as possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria.
        When there are multiple possibilities for s according to the rules above, implementations are encouraged but not required to select the one according to the following rules: Select the value of s for which s[10^{n-k}] is closest in value to r; if there are two such possible values of s, choose the one that is even.
        digits: STRING = integerToString(s);
        if k ≤ n ≤ 21 then return digits ⊕ repeat(‘0’, n - k)
        elsif 0 < n ≤ 21 then return digits[0 ... n - 1] ⊕ “.” ⊕ digits[n ...]
        elsif -6 < n ≤ 0 then return “0.” ⊕ repeat(‘0’, -n) ⊕ digits
        else
          mantissa: STRING =
          if k = 1 then mantissa = digits
          else mantissa = digits[0 ... 0] ⊕ “.” ⊕ digits[1 ...]
          end if;
          return mantissa ⊕ “e” ⊕ integerToStringWithSign(n - 1)
        end if
      end if
    end case
  end proc;

10.2.6 toPrimitive

proc toPrimitive(o: OBJECT, hint: OBJECT, phase: PHASE): PRIMITIVEOBJECT
  case o of
    PRIMITIVEOBJECT do return o;
    NAMESPACE ⊕ COMPOUNDATTRIBUTE ⊕ CLASS ⊕ METHODCLOSURE ⊕ PROTOTYPE ⊕ INSTANCE ⊕ PACKAGE ⊕ GLOBAL do
      return toString(o, phase)
    end case
  end proc;

10.2.7 Attributes

combineAttributes(a, b) returns the attribute that results from concatenating the attributes a and b.
```
proc combineAttributes(a: ATTRIBUTEOPTNOTFALSE, b: ATTRIBUTE): ATTRIBUTE

if b = false then return false
elsif a [] {none, true} then return b
elsif b = true then return a
elsif a [] NAMESPACE then
if a = b then return a
elsif b [] NAMESPACE then
return COMPOUNDATTRIBUTE@namespaces: {a, b}, explicit: false, dynamic: false, memberMod: none,
overrideMod: none, prototype: false, unused: false[]
else return COMPOUNDATTRIBUTE@namespaces: b.namespaces [] {a}, other fields from b[]
end if
elsif b [] NAMESPACE then
return COMPOUNDATTRIBUTE@namespaces: a.namespaces [] {b}, other fields from a[]
else
Both a and b are compound attributes. Ensure that they have no conflicting contents.
if (a.memberMod ≠ none and b.memberMod ≠ none and a.memberMod ≠ b.memberMod) or
(a.overrideMod ≠ none and b.overrideMod ≠ none and a.overrideMod ≠ b.overrideMod) then
throw badValueError
else
return COMPOUNDATTRIBUTE@namespaces: a.namespaces [] b.namespaces,
explicit: a.explicit or b.explicit, dynamic: a.dynamic or b.dynamic,
memberMod: a.memberMod ≠ none ? a.memberMod : b.memberMod,
overrideMod: a.overrideMod ≠ none ? a.overrideMod : b.overrideMod,
prototype: a.prototype or b.prototype, unused: a.unused or b.unused[]
end if
end if
end proc;

toCompoundAttribute(a) returns a converted to a COMPOUNDATTRIBUTE even if it was a simple namespace, true, or none.

proc toCompoundAttribute(a: ATTRIBUTEOPTNOTFALSE): COMPOUNDATTRIBUTE

case a of
{none, true} do
return COMPOUNDATTRIBUTE@namespaces: {}, explicit: false, dynamic: false, memberMod: none,
overrideMod: none, prototype: false, unused: false[]
end case
end proc;

10.3 References

If r is an OBJECT, readReference(r, phase) returns it unchanged. If r is a REFERENCE, this function reads r and returns the result. If phase is compile, only compile-time expressions can be evaluated in the process of reading r.

proc readReference(r: OBJORREF, phase: PHASE): OBJECT

case r of
OBJECT do return r;
LEXICALREFERENCE do return lexicalRead(r.env, r.variableMultiname, phase);
DOTREFERENCE do
result: OBJECTOPT [] readProperty(r.base, r.propertyMultiname, propertyLookup, phase);
if result ≠ none then return result else throw propertyAccessError end if;
BRACKETREFERENCE do return bracketRead(r.base, r.args, phase)
end case
end proc;
proc bracketRead(a: ObjOptionalLimit, args: ArgumentList, phase: Phase): Object
if args.positional ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
name: STRING [] toString(args.positional[0], phase);
result: ObjectOpt [] readProperty(a, {QualifiedName[]namespaces: publicNamespace, id: name[]},
propertyLookup, phase);
if result ≠ none then return result else throw propertyAccessError end if
end proc;

If r is a reference, writeReference(r, newValue) writes newValue into r. An error occurs if r is not a reference. r's limit, if any, is ignored. writeReference is never called from a compile-time expression.

proc writeReference(r: ObjRef, newValue: Object, phase: {run})
result: {none, ok};
case r of
OBJECT do throw referenceError;
LEXICALREFERENCE do
lexicalWrite(r.env, r.variableMultiname, newValue, not r.strict, phase);
return;
DOTREFERENCE do
result [] writeProperty(r.base, r.propertyMultiname, propertyLookup, true, newValue, phase);
BRACKETREFERENCE do result [] bracketWrite(r.base, r.args, newValue, phase)
end case;
if result = none then throw propertyAccessError end if
end proc;

proc bracketWrite(a: ObjOptionalLimit, args: ArgumentList, newValue: Object, phase: Phase): {none, ok}
if phase = compile then throw compileExpressionError end if;
if args.positional ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
name: STRING [] toString(args.positional[0], phase);
return writeProperty(a, {QualifiedName[]namespaces: publicNamespace, id: name[]}, propertyLookup, true, newValue, phase)
end proc;

If r is a Reference, deleteReference(r) deletes it. If r is an Object, this function signals an error in strict mode or returns true in non-strict mode. deleteReference is never called from a compile-time expression.

proc deleteReference(r: ObjRef, strict: BOOLEAN, phase: {run}): BOOLEAN
result: BooleanOpt;
case r of
OBJECT do if strict then throw referenceError else return true end if;
LEXICALREFERENCE do return lexicalDelete(r.env, r.variableMultiname, phase);
DOTREFERENCE do
result [] deleteProperty(r.base, r.propertyMultiname, propertyLookup, phase);
BRACKETREFERENCE do result [] bracketDelete(r.base, r.args, phase)
end case;
if result ≠ none then return result else return true end if
end proc;

proc bracketDelete(a: ObjOptionalLimit, args: ArgumentList, phase: Phase): BooleanOpt
if phase = compile then throw compileExpressionError end if;
if args.positional ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
name: STRING [] toString(args.positional[0], phase);
return deleteProperty(a, {QualifiedName[]namespaces: publicNamespace, id: name[]}, propertyLookup, phase)
end proc;
10.4 Slots

proc findSlot(o: OBJECT, id: INSTANCE_VARIABLE): SLOT
  o must be an INSTANCE;
  matchingSlots: SLOT {} { s | s.x ▶ resolveAlias(o).slots such that s.id = id};
  return the one element of matchingSlots
end proc;

10.5 Environments

If env is from within a class’s body, getEnclosingClass(env) returns the innermost such class; otherwise, it returns none.

proc getEnclosingClass(env: ENVIRONMENT): CLASSOPT
  if some c ▶ env satisfies c ▶ CLASS then
    Let c be the first element of env that is a CLASS.
    return c
  end if;
  return none
end proc;

getRegionalEnvironment(env) returns all frames in env up to and including the first regional frame. A regional frame is either any frame other than a local block frame or a local block frame whose immediate enclosing frame is a class.

proc getRegionalEnvironment(env: ENVIRONMENT): FRAME[]
  i: INTEGER ▶ 0;
  while env[i] ▶ BLOCKFRAME do i ▶ i + 1 end while;
  if i ≠ 0 and env[i] ▶ CLASS then i ▶ i – 1 end if;
  return env[0 ... i]
end proc;

getRegionalFrame(env) returns the most specific regional frame in env.

proc getRegionalFrame(env: ENVIRONMENT): FRAME
  regionalEnv: FRAME[] ▶ getRegionalEnvironment(env);
  return regionalEnv[|regionalEnv| – 1]
end proc;

proc getPackageOrGlobalFrame(env: ENVIRONMENT): PACKAGE OR GLOBAL
  g: FRAME ▶ env[|env| – 2];
  The penultimate frame g is always a PACKAGE or GLOBAL frame.
  return g
end proc;

10.5.1 Access Utilities

tag read;
tag write;
tag readWrite;

ACCESS = {read, write, readWrite};

staticBindingsWithAccess(f, access) returns the set of static bindings in frame f which are used for reading, writing, or either, as selected by access.
proc staticBindingsWithAccess(f: FRAME, access: ACCESS): STATICBINDING{}  
  case access of  
  {read} do return f.staticReadBindings;
  {write} do return f.staticWriteBindings;
  {readWrite} do return f.staticReadBindings ▷ f.staticWriteBindings
end case
end proc;

instanceBindingsWithAccess(c, access) returns the set of instance bindings in class c which are used for reading, writing, or either, as selected by access.

proc instanceBindingsWithAccess(c: CLASS, access: ACCESS): INSTANCEBINDING{}  
  case access of  
  {read} do return c.instanceReadBindings;
  {write} do return c.instanceWriteBindings;
  {readWrite} do return c.instanceReadBindings ▷ c.instanceWriteBindings
end case
end proc;

addStaticBindings(f, access, newBindings) adds newBindings to the set of readable, writable, or both (as selected by access) static bindings in frame f.

proc addStaticBindings(f: FRAME, access: ACCESS, newBindings: STATICBINDING{})  
  if access ▷ {read, readWrite} then  
  f.staticReadBindings ▷ f.staticReadBindings ▷ newBindings
end if;
  if access ▷ {write, readWrite} then  
  f.staticWriteBindings ▷ f.staticWriteBindings ▷ newBindings
end if
end proc;
10.5.2 Adding Static Definitions

```javascript
proc defineStaticMember(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE[]),
    overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: STATICTEMBER): MULTINAME
localFrame: FRAME[] env[0];
if overrideMod ≠ none or (explicit and localFrame [] PACKAGE) then
    throw definitionError
end if;
namespaces2: NAMESPACE[] [] namespaces;
if namespaces2 = {} then namespaces2 [] {publicNamespace} end if;
multiname: MULTINAME[] {QUALIFIEDNAME}[] namespace: ns, id: id[] [] ns [] namespaces2;
regionalEnv: FRAME[] getRegionalEnvironment(env);
regionalFrame: FRAME[] regionalEnv[regionalEnv[regionalEnv[regionalEnv] - 1]];
if some b [] staticBindingsWithAccess(localFrame, access) satisfies b.qname [] multiname then
    throw definitionError
end if;
for each frame [] regionalEnv[1 ...] do
    if some b [] staticBindingsWithAccess(frame, access) satisfies
        b.qname [] multiname and b.content ≠ forbidden then
        throw definitionError
    end if
end for each;
if regionalFrame [] GLOBAL and (some dp [] regionalFrame.dynamicProperties satisfies
    QUALIFIEDNAME}[] namespace: publicNamespace, id: dp.name[] [] multiname) then
    throw definitionError
end if;
newBindings: STATICBINDING[] [] {STATICBINDING}[] name: qname, content: m, explicit: explicit[]
    qname [] multiname;
addStaticBindings(localFrame, access, newBindings);
Mark the bindings of multiname as forbidden in all non-innermost frames in the current region if they haven’t been
marked as such already.
newForbiddenBindings: STATICBINDING[] [] {STATICBINDING}[] name: qname, content: forbidden, explicit: true[]
    qname [] multiname;
for each frame [] regionalEnv[1 ...] do
    addStaticBindings(frame, access, newForbiddenBindings)
end for each;
return multiname
end proc;
```

`defineHoistedVar(env, id, initialValue)` defines a hoisted variable with the name `id` in the environment `env`. Hoisted variables
are hoisted to the global or enclosing function scope. Multiple hoisted variables may be defined in the same scope, but they
may not coexist with non-hoisted variables with the same name. A hoisted variable can be defined using either a `var` or a
`function` statement. If it is defined using `var`, then `initialValue` is always `undefined` (if the `var` statement has an
initialiser, then the variable’s value will be written later when the `var` statement is executed). If it is defined using
`function`, then `initialValue` must be a function instance or open instance. According to rules inherited from ECMAscript
Edition 3, if there are multiple definitions of a hoisted variable, then the initial value of that variable is `undefined` if none of
the definitions is a `function` definition; otherwise, the initial value is the last `function` definition.
proc defineHoistedVar(env: ENVIRONMENT, id: STRING, initialValue: {undefined} » INSTANCE » OPENINSTANCE):

HoistedVar

qname: QUALIFIEDNAME » QUALIFIEDNAME
namespace: publicNamespace, id: id

regionalFrame: FRAME » regionalEnv[ regionalEnv[ regionalEnv[ | – 1];

env is either the GLOBAL frame or a PARAMETERFRAME because hoisting only occurs into global or function scope.

existingBindings: STATICBINDING[] » { b | b » STATICBINDINGS WithAccess(regionalFrame, readWrite) such that b.qname = qname};

if existingBindings = {} then

case regionalFrame of

GLOBAL do

if some dp » regionalFrame.dynamicProperties satisfies dp.name = id then

throw definitionError

end if;

PARAMETERFRAME do

if |regionalEnv| ≥ 2 then

regionalFrame » regionalEnv[ regionalEnv[ | – 2];

existingBindings » { b | b » STATICBINDINGS WithAccess(regionalFrame, readWrite) such that b.qname = qname}

end if

end case

end if;

if existingBindings = {} then

v: HoistedVar » new HoistedVar[] value: initialValue, hasFunctionInitialiser: initialValue ≠ undefined[]

addStaticBindings(regionalFrame, readWrite, { STATICBINDING[] name: qname, content: v, explicit: false[] });

return v

elsif |existingBindings| ≠ 1 then throw definitionError

else

b: STATICBINDING » the one element of existingBindings;

m: STATICMEMBER » b.content;

if m » HoistedVar then throw definitionError end if;

A hoisted binding of the same var already exists, so there is no need to create another one. Overwrite its initial value if the new definition is a function definition.

if initialValue ≠ undefined then

m.value » initialValue;

m.hasFunctionInitialiser » true

end if;

return m

end if

end proc;

10.5.3 Adding Instance Definitions

tuple OVERRIDESTATUSPAIR

readStatus: OVERRIDESTATUS,

writeStatus: OVERRIDESTATUS

dering tuple;

tag potentialConflict;

OVERRIDDENMEMBER = INSTANCEMEMBER » {none, potentialConflict};

tuple OVERRIDESTATUS

overriddenMember: OVERRIDDENMEMBER,

multiname: MULTINAME

dering tuple;
proc searchForOverrides(c: CLASS, id: STRING, namespaces: NAMESPACE{}, access: {read, write}): OVERRIDESTATUS
multiname: MULTINAME[] {}; overriddenMember: INSTANCEMEMBEROPT[] none;
s: CLASSOPT[] c.super;
for each ns[] namespaces do
  qname: QUALIFIEDNAME[] QUALIFIEDNAME[name: ns, id: id[]
m: INSTANCEMEMBEROPT[] findInstanceMember(s, qname, access);
  if m ≠ none then
    multiline[] multiline[] {qname};
    if overriddenMember = none then overriddenMember[] m
    else overriddenMember ≠ m then throw definitionError
  end if
end for each;
return OVERRIDESTATUS[overriddenMember: overriddenMember, multiline: multiline[]]
end proc;

proc resolveOverrides(c: CLASS, ext: CONTEXT, id: STRING, namespaces: NAMESPACE{}, access: {read, write},
expectMethod: BOOLEAN): OVERRIDESTATUS
os: OVERRIDESTATUS;
if namespaces = {} then
  os[] searchForOverrides(c, id, ext.openNamespaces, access);
  if os overriddenMember = none then
    os[] OVERRIDESTATUS[overriddenMember: none, multiline: {QUALIFIEDNAME[name: publicNamespace, id: id[]}
  end if
else
  definedMultiname: MULTINAME[] {QUALIFIEDNAME[name: ns, id: id[] ns[] namespaces};
os2: OVERRIDESTATUS[] searchForOverrides(c, id, namespaces, access);
  if os2 overriddenMember = none then
    os3: OVERRIDESTATUS[] searchForOverrides(c, id, ext.openNamespaces – namespaces, access);
    if os overriddenMember = none then
      os[] OVERRIDESTATUS[overriddenMember: none, multiline: definedMultiname[]
    else
      os[] OVERRIDESTATUS[overriddenMember: os2 overriddenMember, multiline: os2 multiline[] definedMultiname[]
    end if
  else
    os[] OVERRIDESTATUS[overriddenMember: os2 overriddenMember, multiline: os2 multiline[] definedMultiname[]
end if
if some b[] instanceBindingsWithAccess(c, access) satisfies b.qname[] os multiline then
  throw definitionError
end if;
if expectMethod then
  if os overriddenMember [] {none, potentialConflict} INSTANCEMETHOD then
    throw definitionError
  end if
else
  if os overriddenMember [] {none, potentialConflict} INSTANCEVARIABLE INSTANCEGETTER
    INSTANCESETTER then
    throw definitionError
  end if
end if;
return os
end proc;
proc defineInstanceMember(c: CLASS, ext: CONTEXT, id: STRING, namespaces: NAMESPACE{});
  overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, access: ACCESS, m: INSTANCE
  OVERSTATUSPAIR

  if explicit then throw definitionError end if;
  expectMethod: BOOLEAN { m INSTANCEMETHOD;

  readStatus: OVERRIDESTATUS access { read, readWrite } ?
    resolveOverrides(c, ext, id, namespaces, read, expectMethod):
      OVERRIDESTATUS overriddenMember: none, multiname: {}

  writeStatus: OVERRIDESTATUS access { write, readWrite } ?
    resolveOverrides(c, ext, id, namespaces, write, expectMethod):
      OVERRIDESTATUS overriddenMember: none, multiname: {}

  if readStatus.overriddenMember INSTANCEMEMBER or
    writeStatus.overriddenMember INSTANCEMEMBER then
    if overrideMod { true, undefined } then throw definitionError end if
  elsif readStatus.overriddenMember = potentialConflict or
    writeStatus.overriddenMember = potentialConflict then
    if overrideMod { false, undefined } then throw definitionError end if
  else if overrideMod { none, false, undefined } then throw definitionError end if

  end if;
  newReadBindings: INSTANCEBINDING{} []
    { INSTANCEBINDING[] name: qname, content: m[] qname readStatus.multiname };
  c.instanceWriteBindings [] c.instanceReadBindings newReadBindings;
  newWriteBindings: INSTANCEBINDING{} []
    { INSTANCEBINDING[] name: qname, content: m[] qname writeStatus.multiname };
  c.instanceWriteBindings [] c.instanceWriteBindings newWriteBindings;
  return OVERRIDESTATUSPAIR readStatus: readStatus, writeStatus: writeStatus
end proc;

10.5.4 Instantiation

proc instantiateOpenInstance(oi: OPENINSTANCE, env: ENVIRONMENT): INSTANCE
  cache: CALLABLEINSTANCE {} { none } oi.cache;

  if cache = none then
    slots: SLOT{} [] { new SLOT[] id: s.id, value: s.value } s oi.defaultSlots ;
    dynamicProperties: DYNAMICPROPERTY{} [] { fixed };
    if oi.buildPrototype then dynamicProperties {} ;
  else dynamicProperties fixed
  end if;
  i: CALLABLEINSTANCE new CALLABLEINSTANCE[type: oi.type, typeofString: oi.typeOfString, slots: slots,

  reuse: BOOLEAN;

At the implementation’s discretion, either reuse true, or reuse false. An implementation may make different
choices at different times. The intent here is to allow implementations the freedom to reuse a closure object
rather than create a new closure each time a particular OPENINSTANCE is instantiated if the implementation
notices that the resulting closures would be behaviorally indistinguishable from each other.

if reuse then oi.cache / i end if;
  return i
else return new ALIASINSTANCE[original: cache, env: env]
end if
end proc;
proc instantiateMember(m: StaticMember, env: Environment): StaticMember
  case m of
  {forbidden} do return m;
  VARIABLE do
    value: VARIABLE_VALUE m.value;
    if value [] OpenInstance then value [] instantiateOpenInstance(value, env)
  end if;
  return new VARIABLE{type: m.type, value: value, immutable: m.immutable}
  HOISTEDVAR do
    value: OBJECT [] OpenInstance m.value;
    if value [] OpenInstance then value [] instantiateOpenInstance(value, env)
  end if;
  return new HOISTEDVAR{value: value, hasFunctionInitialiser: m.hasFunctionInitialiser}
  CONSTRUCTOR do return m;
  GETTER do
    case m.env of
      Environment do return m;
      {inaccessible} do return new GETTER{type: m.type, call: m.call, env: env}
    end case;
  SETTER do
    case m.env of
      Environment do return m;
      {inaccessible} do return new SETTER{type: m.type, call: m.call, env: env}
    end case
  end case
end proc;

tuple MemberInstantiation
  pluralMember: StaticMember,
  singularMember: StaticMember
end tuple;

proc instantiateFrame(pluralFrame: ParametersFrame [] BlockFrame,
  singularFrame: ParametersFrame [] BlockFrame, env: Environment)
pluralMembers: StaticMember[] [] {b.content | pluralFrame.staticReadBindings [] pluralFrame.staticWriteBindings};
memberInstantiations: MemberInstantiation[] []
{MemberInstantiation{pluralMember: m, singularMember: instantiateMember(m, env)}}
{m [] pluralMembers};
proc instantiateBinding(b: StaticBinding): StaticBinding
  mi: MemberInstantiation [] the one element mi [] memberInstantiations that satisfies mi.pluralMember = b.content;
  return StaticBinding{name: b.qname, content: mi.singularMember, explicit: b.explicit}
end proc;
  singularFrame.staticReadBindings [] {instantiateBinding(b) [] b [] pluralFrame.staticReadBindings};
  singularFrame.staticWriteBindings [] {instantiateBinding(b) [] b [] pluralFrame.staticWriteBindings}
end proc;

### 10.5.5 Environmental Lookup

=findThis(env, allowPrototypeThis) returns the value of this. If allowPrototypeThis is true, allow this to be defined by either an instance member of a class or a prototype function. If allowPrototypeThis is false, allow this to be defined only by an instance member of a class.
proc findThis(env: ENVIRONMENT, allowPrototypeThis: BOOLEAN): OBJECT OPT
    for each frame [ env do
        if frame [ PARAMETERFRAME and frame.this ≠ none then
            if allowPrototypeThis or not frame.prototype then return frame.this end if
        end if
    end for each;
    return none end proc;

dec lexicalRead(env: ENVIRONMENT, multiname: MULTINAME, phase: PHASE): OBJECT
    kind: LOOKUPKIND[].
    [ this: findThis(env, false)]
    i: INTEGER[] 0;
    while i < |env| do
        frame: FRAME[] env[i];
        result: OBJECTOPT[] readProperty(frame, multiname, kind, phase);
        if result ≠ none then return result end if;
        i [] i + 1
    end while;
    throw referenceError end proc;

dec lexicalWrite(env: ENVIRONMENT, multiname: MULTINAME, newValue: OBJECT, createIfMissing: BOOLEAN, phase: [run]): BOOLEAN
    kind: LOOKUPKIND[].
    [ this: findThis(env, false)]
    i: INTEGER[] 0;
    while i < |env| do
        frame: FRAME[] env[i];
        result: [none, ok] [] writeProperty(frame, multiname, kind, false, newValue, phase);
        if result = ok then return end if;
        i [] i + 1
    end while;
    if createIfMissing then
        g: PACKAGE[] GLOBAL[] getPackageOrGlobalFrame(env);
        if g[] GLOBAL then
            Now try to write the variable into g again, this time allowing new dynamic bindings to be created dynamically.
            result: [none, ok] [] writeProperty(g, multiname, kind, true, newValue, phase);
            if result = ok then return end if
        end if
    end if;
    throw referenceError end proc;

dec lexicalDelete(env: ENVIRONMENT, multiname: MULTINAME, phase: [run]): BOOLEAN
    kind: LOOKUPKIND[].
    [ this: findThis(env, false)]
    i: INTEGER[] 0;
    while i < |env| do
        frame: FRAME[] env[i];
        result: BOOLEANOPT[] deleteProperty(frame, multiname, kind, phase);
        if result ≠ none then return result end if;
        i [] i + 1
    end while;
    return true end proc;

10.5.6 Property Lookup
    tag propertyLookup;
tuple \texttt{LEXICALLOOKUP}
  this: \texttt{OBJECTOPT}
end tuple;

\texttt{LOOKUPKIND} = \{\texttt{propertyLookup} \} \cup \texttt{LEXICALLOOKUP};

\texttt{proc selectPublicName(multiname: MULTINAME): STRINGOPT}
  if some qname \in \texttt{multiname} satisfies qname\texttt{.namespace} = \texttt{publicNamespace} then
    return qname.id
  end if;
  return none
end proc;

\texttt{proc findFlatMember(frame: FRAME, multiname: MULTINAME, access: \{read, write\}, phase: PHASE):}
  \texttt{STATICMEMBEROPT}
  matchingBindings: \texttt{STATICBINDING}\{\} \cup
    \{b | \cup \exists b \in \texttt{staticBindingsWithAccess(frame, access)} \texttt{.qname} \in \texttt{multiname}\};
  if matchingBindings = \{\} then return none end if;
  matchingMembers: \texttt{STATICMEMBER}\{\} \cup \{b.\texttt{content} | \cup \exists b \in \texttt{matchingBindings}\};
  Note that if the same member was found via several different bindings \texttt{b}, then it will appear only once in the set
  matchingMembers.
  if |matchingMembers| > 1 then
    This access is ambiguous because the bindings it found belong to several different members in the same class.
    throw \texttt{propertyAccessError}
  end if;
  return the one element of matchingMembers
end proc;
proc findStaticMember(c: CLASSOPT, multiname: MULTINAME, access: {read, write}, phase: PHASE): {none} □ STATICMEMBER □ QUALIFIEDNAME
  s: CLASSOPT □ c;
  while s ≠ none do
    matchingStaticBindings: STATICBINDING{ } □
      {b | □ b □ staticBindingsWithAccess(s, access) such that b.qname □ multiname};
    Note that if the same member was found via several different bindings b, then it will appear only once in the set matchingStaticMembers.
    matchingStaticMembers: STATICMEMBER{ } □ {b.content | □ b □ matchingStaticBindings};
    if matchingStaticMembers ≠ {} then
      if |matchingStaticMembers| = 1 then
        return the one element of matchingStaticMembers
      else
        This access is ambiguous because the bindings it found belong to several different static members in the same class.
        throw propertyAccessError
      end if
    end if;
  end while;
  return none
end proc;
proc `resolveInstanceMemberName(c: CLASS, multiname: MULTINAME, access: {read, write}, phase: PHASE): QUALIFIEDNAMEOPT`

Start from the root class (`Object`) and proceed through more specific classes that are ancestors of `c`.

for each `s` in ancestors(`c`) do

  `matchingInstanceBindings: INSTANCEBINDING[] { b | b in instanceBindingsWithAccess(s, access) such that b.qname = multiname };`

  Note that if the same `INSTANCEMEMBER` was found via several different bindings `b`, then it will appear only once in the set `matchingMembers`.

  `matchingInstanceMembers: INSTANCEMEMBER[] { b.content | b in matchingInstanceBindings };`

if `matchingInstanceMembers ≠ {}` then

  if `|matchingInstanceMembers| = 1` then
    Return the qualified name of any matching binding. It doesn’t matter which because they all refer to the same `INSTANCEMEMBER`, and if one is overridden by a subclass then all must be overridden in the same way by that subclass.

    `b: INSTANCEBINDING[] any element of matchingInstanceBindings;`

    return `b.qname`
  else
    This access is ambiguous because the bindings it found belong to several different members in the same class.

    throw `propertyAccessError`
  end if
end if
end for each;
return `none`
end proc;

proc `findInstanceMember(c: CLASSOPT, qname: QUALIFIEDNAMEOPT, access: {read, write}): INSTANCEMEMBEROPT`

if `qname = none` then return `none` end if;

`s: CLASSOPT[] c;`

while `s ≠ none` do

  if some `b in instanceBindingsWithAccess(s, access) satisfies b.qname = qname` then
    return `b.content`
  end if;
  `s = s.super`
end while;
return `none`
end proc;

10.5.7 Reading a Property

tag generic;
proc readProperty(container: ObjectOptionalLimit [FRAME, multiname: Multiname, kind: LookupKind, phase: Phase]: ObjectOpt

case container of
  Undefined [NULL | BOOLEAN | GENERALNUMBER | CHARACTER | STRING | NAMESPACE]
    CompoundAttribute [METHODCLOSURE | INSTANCE] do
      c: CLASS [] objectType(container);
      qname: QualifiedNameOpt [] resolveInstanceMemberName(c, multiname, read, phase);
      if qname = none and container [] INSTANCE then
        return readDynamicProperty(resolveAlias(container), multiname, kind, phase)
      else return readInstanceMember(container, c, qname, phase)
      end if;
    SystemFrame [GLOBAL | PACKAGE | PARAMETERFRAME | BLOCKFRAME] do
      m: StaticMemberOpt [] findFlatMember(container, multiname, read, phase);
      if m = none and container [] GLOBAL then
        return readDynamicProperty(container, multiname, kind, phase)
      else return readStaticMember(m, phase)
      end if;
    Class do
      this: Object [] {inaccessible, none, generic};
      case kind of
        {propertyLookup} do this [] generic;
          lexicalLookup do this [] kind.this
        end case;
      end case;
    end case;
  m2: {none} [] StaticMember [] QualifiedName [] findStaticMember(container, multiname, read, phase);
  if m2 [] QualifiedName then return readStaticMember(m2, phase) end if;
  case this of
    {none} do throw propertyAccessError;
    {inaccessible} do throw compileExpressionError;
    {generic} do ????
    Object do return readInstanceMember(this, objectType(this), m2, phase)
  end case;
end case;
 Prototype do return readDynamicProperty(container, multiname, kind, phase);
 LimitedInstance do
   superclass: CLASSOpt [] container.limit.super;
   if superclass = none then return none end if;
   qname: QualifiedNameOpt [] resolveInstanceMemberName(superclass, multiname, read, phase);
   return readInstanceMember(container, instance, superclass, qname, phase)
 end case
end proc;

proc readInstanceMember(this: Object, c: CLASS, qname: QualifiedNameOpt, phase: Phase): ObjectOpt
m: InstanceMemberOpt [] findInstanceMember(c, qname, read);

end case
end proc;
proc readStaticMember(m: STATICMEMBEROPT, phase: PHASE): OBJECTOPT
    case m of
        {none} do return none;
        {forbidden} do throw propertyAccessError;
        VARIABLE do return readVariable(m, phase);
        HOISTEDVAR do
            if phase = compile then throw compileExpressionError end if;
            value: OBJECT OPT OPENINSTANCE m.value;
            Note that value can be an OPENINSTANCE only during the compile phase, which was ruled out above.
            return value;
        CONCATSTRUCTORMETHOD do return m.code;
        GETTER do
            env: ENVIRONMENT OPT m.env;
            if env = inaccessible then throw compileExpressionError end if;
            return m.call(env, phase);
        SETTER do
            m cannot be a SETTER because these are only represented as write-only members.
        end case
    end proc;

proc readDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME, kind: LOOKUPKIND, phase: PHASE): OBJECTOPT
    name: STRINGOPT OPT selectPublicName(multiname);
    if name = none then return none end if;
    if phase = compile then throw compileExpressionError end if;
    dynamicProperties: DYNAMICPROPERTY OPT {fixed} OPT container.dynamicProperties;
    if dynamicProperties ≠ fixed and (some dp OPT dynamicProperties satisfies dp.name = name) then return dp.value end if;
    end if;
    if container OPT PROTOTYPE then
        parent: PROTOTYPEOPT OPT container.parent;
        if parent ≠ none then return readDynamicProperty(parent, multiname, kind, phase) end if;
    end if;
    if kind = propertyLookup then return undefined end if;
    return none
end proc;
proc readVariable(v: VARIABLE, phase: PHASE): OBJECT
    if phase = compile and not v.immutable then throw compileExpressionError end if;
    value: VARIABLEVALUE  v.value;
    case value of
        OBJECT do return value;
        {inaccessible} do
            if phase = compile then throw compileExpressionError
            else throw uninitialisedError
        end if;
        {uninitialised} do throw uninitialisedError;
        OPENINSTANCE do
            Note that an uninstantiated function can only be found when phase = compile.
            throw compileExpressionError;
        () OBJECT do
            Note that phase = compile because all futures are resolved by the end of the compilation phase.
            v.value  inaccessible;
            type: CLASS  getVariableType(v, phase);
            newValue: OBJECT  value();
            coercedValue: OBJECT  type.implicitCoerce(newValue);
            v.value  coercedValue;
            return newValue
        end case
    end proc;
10.5.8 Writing a Property

```
proc writeProperty(container: OPTIONAL LIMIT Frame, multiname: MULTINAME, kind: LOOKUP KIND,
                    createIfMissing: BOOLEAN, newValue: OBJECT, phase: {run}): {none, ok}
  case container of
    UNDEFINED » NULL » BOOLEAN » GENERAL NUMBER » CHARACTER » STRING »NAMESPACE »
      COMPOUND ATTRIBUTE » METHOD CLOSURE do
      return none;
    SYSTEM FRAME » GLOBAL » PACKAGE » PARAMETER FRAME » BLOCK FRAME do
      m: STATIC MEMBER OPT $ findFlatMember(container, multiname, write, phase);
      if m = none and container » GLOBAL then
        return writeDynamicProperty(container, multiname, createIfMissing, newValue, phase)
      else return writeStaticMember(m, newValue, phase)
      end if;
    CLASS do
      this: OBJECT OPT;
      case kind of
       {propertyLookup} do this $ none;
      LEXICAL LOOKUP do this $ kind . this
      end case;
      m2: {none} » STATIC MEMBER » QUALIFIED NAME $ findStaticMember(container, multiname, write, phase);
      if m2 » QUALIFIED NAME then return writeStaticMember(m2, newValue, phase)
      elsif this = none then throw propertyAccessError
      elsif this = inaccessible then throw compileExpressionError
      else return writeInstanceMember(this, objectType(this), m2, newValue, phase)
      end if;
    PROTOTYPE do
      return writeDynamicProperty(container, multiname, createIfMissing, newValue, phase);
    INSTANCE do
      c: CLASS OPT $ objectType(container);
      qname: QUALIFIED NAME OPT $ resolveInstanceMemberName(objectType(container), multiname, write, phase);
      if qname = none then
        return writeDynamicProperty(resolveAlias(container), multiname, createIfMissing, newValue, phase)
      else return writeInstanceMember(container, c, qname, newValue, phase)
      end if;
    LIMITED INSTANCE do
      superclass: CLASS OPT $ container . limit . super;
      if superclass = none then return none end if;
      qname: QUALIFIED NAME OPT $ resolveInstanceMemberName(superclass, multiname, write, phase);
      return writeInstanceMember(container . instance, superclass, qname, newValue, phase)
    end case
  end proc
```
proc writeInstanceMember(this: OBJECT, c: CLASS, qname: QUALIFIEDNAMEOPT, newValue: OBJECT, phase: {run}): {none, ok}

m: INSTANCEMEMBEROPT [] findInstanceMember(c, qname, write);

case m of

{none} do return none;

INSTANCEVARIABLE do

s: SLOT [] findSlot(this, m);

if m.immutable and s.value ≠ uninitialised then throw propertyAccessError
end if;

coercedValue: OBJECT [] m.type.implicitCoerce(newValue);

s.value [] coercedValue;

return ok;

INSTANCEMETHOD do throw propertyAccessError;

INSTANCEGETTER do

m cannot be an INSTANCEGETTER because these are only represented as read-only members.

INSTANCESETTER do

coercedValue: OBJECT [] m.type.implicitCoerce(newValue);

m.call(this, coercedValue, m.env, phase);

return ok
end case

end proc;

proc writeStaticMember(m: STATICMEMBEROPT, newValue: OBJECT, phase: {run}): {none, ok}

m: INSTANCEMEMBEROPT [] findInstanceMember(m, newValue, phase);

case m of

{none} do return none;

{forbidden} » CONSTRUCTORMETHOD do throw propertyAccessError;

VARIABLE do writeVariable(m, newValue, phase); return ok;

HOISTEDVAR do m.value [] newValue; return ok;

GETTER do

m cannot be a GETTER because these are only represented as read-only members.

SETTER do

coercedValue: OBJECT [] m.type.implicitCoerce(newValue);

env: ENVIRONMENTI [] m.env;

Note that all instances are resolved for the run phase, so env ≠ inaccessible.

m.call(coercedValue, env, phase);

return ok
end case

end proc;
proc writeDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME, createIfMissing: BOOLEAN, newValue: OBJECT, phase: {run}): {none, ok}
name: STRINGOPT [] selectPublicName(multiname);
if name = none then return none end if;
dynamicProperties: DYNAMICPROPERTY [] {fixed} [] container.dynamicProperties;
if dynamicProperties = fixed then return none end if;
if some dp in dynamicProperties satisfies dp.name = name then
  dp.value [] newValue;
  return ok
end if;
if not createIfMissing then return none end if;
Before trying to create a new dynamic property, check that there is no read-only fixed property with the same name.

m: {none} [] STATICMEMBER [] QUALIFIEDNAME;
case container of
  Prototype do m [] none;
  SimpleInstance [] CallableInstance do
    m [] resolveInstanceMemberName(objectType(container), multiname, read, phase);
  GLOBAL do m [] findFlatMember(container, multiname, read, phase)
end case;
if m ≠ none then return none end if;
container.dynamicProperties [] dynamicProperties [] {new DYNAMICPROPERTY[] name: name, value: newValue[]};
return ok
end proc;

proc getVariableType(v: VARIABLE, phase: PHASE): CLASS
  type: VARIABLETYPE [] v.type;
case type of
    CLASS do return type;
    {inaccessible} do
      Note that this can only happen when phase = compile because the compilation phase ensures that all types are valid, so invalid types will not occur during the run phase.
      throw compileExpressionError;
() [] CLASS do
  Note that phase = compile because all futures are resolved by the end of the compilation phase.
  v.type [] inaccessible;
  newType: CLASS [] type();
  v.type [] newType;
return newType
end case
end proc;

proc writeVariable(v: VARIABLE, newValue: OBJECT, phase: {run})
  type: CLASS [] getVariableType(v, phase);
if v.value = inaccessible or (v.immutable and v.value ≠ uninitialised) then
  throw propertyAccessError
end if;
coercedValue: OBJECT [] type.implicitCoerce(newValue);
  v.value [] coercedValue
end proc;
10.5.9 Deleting a Property

**proc** deleteProperty**(container: OPTIONAL LIMIT FRAME, renamedName: MULTINAME, kind: LAYOUT KIND, phase: {run}): BOOLEAN OPT**

```plaintext
case container of
    UNDEFINED | NULL | BOOLEAN | GENERAL NUMBER | CHARACTER | STRING | NAME SPACE |
        | COMPOUND ATTRIBUTE | METHOD CLOSURE | INSTANCE do
    c: CLASS | objectType(container);
        qname: QUALIFIED NAME OPT | resolveInstanceMemberName(c, renamedName, read, phase);
        if qname = none and container ▶ INSTANCE then
            return deleteDynamicProperty(resolveAlias(container), renamedName)
        else return deleteInstanceMember(c, qname)
    end if;

SYSTEM FRAME | GLOBAL | PACKAGE | PARAMETER FRAME | BLOCK FRAME do
    m: STATIC MEMBER OPT | findFlatMember(container, renamedName, read, phase);
    if m = none and container ▶ GLOBAL then
        return deleteDynamicProperty(container, renamedName)
    else return deleteStaticMember(m)
    end if;

CLASS do
    this: OBJECT ▶ {none, generic};
    case kind of
        | propertyLookup do this ▶ generic;
            LEXICAL LOOKUP do
                this ▶ kind . this;
                Note that this cannot be inaccessible during the run phase.
            end case;
        m2: {none} | STATIC MEMBER | QUALIFIED NAME | findStaticMember(container, renamedName, read, phase);
        if m2 ▶ QUALIFIED NAME then return deleteStaticMember(m2) end if;
    end case;

PROTOTYPE do return deleteDynamicProperty(container, renamedName);

LIMITED INSTANCE do
    superclass: CLASS OPT ▶ container . limit . super;
    if superclass = none then return none end if;
    qname: QUALIFIED NAME OPT | resolveInstanceMemberName(superclass, renamedName, read, phase);
    return deleteInstanceMember(superclass, qname)
end case
end proc;
```

**proc** deleteInstanceMember**(c: CLASS, qname: QUALIFIED NAME OPT): BOOLEAN OPT**

```plaintext
m: INSTANCE MEMBER OPT | findInstanceMember(c, qname, read);
if m = none then return none end if;
return false
end proc;
```

**proc** deleteStaticMember**(m: STATIC MEMBER OPT): BOOLEAN OPT**

```plaintext
case m of
    | {none} do return none;
    | {forbidden} do throw propertyAccessError;
        VARIABLE ▶ HOISTED VAR ▶ CONSTRUCTOR METHOD ▶ GETTER ▶ SETTER do return false
end case
end proc;
```
proc deleteDynamicProperty(container: DYNAMICOBJECT, multiname: MULTINAME): BOOLEANOPT
name: STRINGOPT selectPublicName(multiname);
if name = none then return none end if;
dynamicProperties: DYNAMICPROPERTY \{} \{fixed\} \{container.dynamicProperties;
if dynamicProperties = fixed then return none end if;
if some dp \{dynamicProperties satisfies dp.name = name then
    container.dynamicProperties \{dynamicProperties = \{dp\};
    return true
else return none
end if
end proc;

10.6 Invocation
proc badConstruct(args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
    throw propertyAccessError
end proc;

11 Evaluation

11.1 Phases of Evaluation

- Parse using the grammar. If the parse fails, throw a syntax error.
- Call Validate on the goal nonterminal, which will recursively call Validate on some intermediate nonterminals. This checks that the program is well-formed, ensuring for instance that break and continue labels exist, compile-time constant expressions really are compile-time constant expressions, etc. If the check fails, Validate will throw an exception.
- Call Setup on the goal nonterminal, which will recursively call Setup on some intermediate nonterminals.
- Call Eval on the goal nonterminal.

11.2 Constant Expressions

12 Expressions

Some expression grammar productions in this chapter are parameterised (see section 5.14.4) by the grammar argument \[\]\{allowIn, noIn\}

Most expression productions have both the Validate and Eval actions defined. Most of the Eval actions on subexpressions produce an Object or Result result, indicating that the subexpression may evaluate to either a value or a place that can potentially be read, written, or deleted (see section 9.3).

12.1 Identifiers

An Identifier is either a non-keyword Identifier token or one of the non-reserved keywords get, set, exclude, include, or named. In either case, the Name action on the Identifier returns a string comprised of the identifier’s characters after the lexer has processed any escape sequences.
Syntax

Identifier []
  Identifier | get | set | exclude | include | named

Semantics

Name[Identifier]: STRING;
  Name[Identifier] Identifier = Name[Identifier];
  Name[Identifier] get = “get”;
  Name[Identifier] set = “set”;
  Name[Identifier] exclude = “exclude”;
  Name[Identifier] include = “include”;
  Name[Identifier] named = “named”;

12.2 Qualified Identifiers

Syntax

Qualifier []
  Identifier | public | private

SimpleQualifiedIdentifier []
  Identifier | Qualified := Identifier

ExpressionQualifiedIdentifier [] ParenExpression := Identifier

QualifiedIdentifier []
  SimpleQualifiedIdentifier
  ExpressionQualifiedIdentifier

Validation

proc Validate[Qualifier] (ext: CONTEXT, env: ENVIRONMENT): NAMESPACE
[Qualifier [] Identifier] do
  multiline: MULTINAME [] {QUALIFIED.NAME[namespace: ns, id: Name[Identifier]]
    ns [] ext.openNamespaces};
  a: OBJECT lexicalRead(env, multiline, compile);
  if a [] NAMESPACE then throw badValueError end if;
  return a;
[Qualifier [] public] do return publicNamespace;
[Qualifier [] private] do
  c: CLASSOPT getEnclosingClass(env);
  if c = none then throw syntaxError end if;
  return c.privateNamespace
end proc;

Multiname[SimpleQualifiedIdentifier]: MULTINAME;
proc Validate[SimpleQualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
    [SimpleQualifiedIdentifier [] Identifier] do
        multiname: MULTINAME [] {QUALIFIEDNAME} namespace: ns, id: Name[Identifier][]
            ns [] cxt.openNamespaces;
        Multiname[SimpleQualifiedIdentifier] [] multiname;
        [SimpleQualifiedIdentifier [] Qualifier :: Identifier] do
            q: NAMESPACE [] Validate[Qualifier](cxt, env);
        Multiname[SimpleQualifiedIdentifier] [] {QUALIFIEDNAME} namespace: q, id: Name[Identifier][]
    end proc;

Multiname[ExpressionQualifiedIdentifier]: MULTINAME;

proc Validate[ExpressionQualifiedIdentifier [] ParenExpression :: Identifier] (cxt: CONTEXT, env: ENVIRONMENT)
    Validate[ParenExpression](cxt, env);
    Setup[ParenExpression]();
    r: ObjOrRef [] Eval[ParenExpression](env, compile);
    q: OBJECT [] readReference(r, compile);
    if q [] NAMESPACE then throw badValueError end if;
        Multiname[ExpressionQualifiedIdentifier] [] {QUALIFIEDNAME} namespace: q, id: Name[Identifier][]
    end proc;

Multiname[QualifiedIdentifier]: MULTINAME;

proc Validate[QualifiedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
    [QualifiedIdentifier [] SimpleQualifiedIdentifier] do
        Validate[SimpleQualifiedIdentifier](cxt, env);
        Multiname[QualifiedIdentifier] [] Multiname[SimpleQualifiedIdentifier];
        [QualifiedIdentifier [] ExpressionQualifiedIdentifier] do
            Validate[ExpressionQualifiedIdentifier](cxt, env);
            Multiname[QualifiedIdentifier] [] Multiname[ExpressionQualifiedIdentifier]
    end proc;

Setup

proc Setup[SimpleQualifiedIdentifier] ()
    [SimpleQualifiedIdentifier [] Identifier] do nothing;
    [SimpleQualifiedIdentifier [] Qualifier :: Identifier] do nothing
end proc;

proc Setup[ExpressionQualifiedIdentifier [] ParenExpression :: Identifier] ()
end proc;

Setup[QualifiedIdentifier] () propagates the call to Setup to every nonterminal in the expansion of QualifiedIdentifier.
12.3 Primary Expressions

Syntax

```
PrimaryExpression →
    null
  | true
  | false
  | public
  | Number
  | String
  | this
  | RegularExpression
  | ParenListExpression
  | ArrayLiteral
  | ObjectLiteral
  | FunctionExpression

ParenExpression → ( AssignmentExpression )

ParenListExpression → ParenExpression
                      | ( ListExpression , AssignmentExpression )
```

Validation

```
proc Validate[PrimaryExpression] (ctx: CONTEXT, env: ENVIRONMENT)
[PrimaryExpression → null] do nothing;
[PrimaryExpression → true] do nothing;
[PrimaryExpression → false] do nothing;
[PrimaryExpression → public] do nothing;
[PrimaryExpression → Number] do nothing;
[PrimaryExpression → String] do nothing;
[PrimaryExpression → this] do
    if findThis(env, true) = none then throw syntaxError end if;
[PrimaryExpression → RegularExpression] do nothing;
[PrimaryExpression → ParenListExpression] do
    Validate[ParenListExpression](ctx, env);
[PrimaryExpression → ArrayLiteral] do Validate[ArrayLiteral](ctx, env);
[PrimaryExpression → ObjectLiteral] do Validate[ObjectLiteral](ctx, env);
[PrimaryExpression → FunctionExpression] do Validate[FunctionExpression](ctx, env)
end proc;
```

```
Validate[ParenExpression] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ParenExpression.

Validate[ParenListExpression] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ParenListExpression.
```

Setup

```
Setup[PrimaryExpression] () propagates the call to Setup to every nonterminal in the expansion of PrimaryExpression.

Setup[ParenExpression] () propagates the call to Setup to every nonterminal in the expansion of ParenExpression.
```
Setup[ParenListExpression] () propagates the call to Setup to every nonterminal in the expansion of ParenListExpression.

Evaluation

proc Eval[PrimaryExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[PrimaryExpression] null] do return null;
[PrimaryExpression] true] do return true;
[PrimaryExpression] false] do return false;
[PrimaryExpression] public] do return publicNamespace;
[PrimaryExpression] Number] do return Value[Number];
[PrimaryExpression] String] do return Value[String];
[PrimaryExpression] this] do
  this: OBJECTOPT ] findThis(env, true);
  Note that Validate ensured that this cannot be none at this point.
  if this = inaccessible then throw compileExpressionError end if;
  return this;
[PrimaryExpression] RegularExpression] do ????;
[PrimaryExpression] ParenListExpression] do
  return Eval[ParenListExpression](env, phase);
[PrimaryExpression] ArrayLiteral] do return Eval[ArrayLiteral](env, phase);
[PrimaryExpression] ObjectLiteral] do return Eval[ObjectLiteral](env, phase);
[PrimaryExpression] FunctionExpression] do
  return Eval[FunctionExpression](env, phase)
end proc;

return Eval[AssignmentExpression](env, phase)
end proc;

[ParenListExpression] ParenExpression] do return Eval[ParenExpression](env, phase);
[ParenListExpression] ( ListExpression , AssignmentExpression ] do
  ra: OBJORREF ] Eval[ListExpression](env, phase);
  readReference(ra, phase);
  rb: OBJORREF ] Eval[AssignmentExpression](env, phase);
  return readReference(rb, phase)
end proc;

proc EvalAsList[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
[ParenListExpression] ParenExpression] do
  r: OBJORREF ] Eval[ParenExpression](env, phase);
  elt: OBJECT ] readReference(r, phase);
  return [elt];
[ParenListExpression] ( ListExpression , AssignmentExpression ] do
  elts: OBJECT[] ] EvalAsList[ListExpression](env, phase);
  r: OBJORREF ] Eval[AssignmentExpression](env, phase);
  elt: OBJECT ] readReference(r, phase);
  return elts ] elt]
end proc;
12.4 Function Expressions

Syntax

\[
\text{FunctionExpression} \equiv \\
\text{function FunctionCommon} \mid \text{function Identifier FunctionCommon}
\]

Validation

\[F[\text{FunctionExpression}]: \text{OPENINSTANCE};\]

\[\text{proc Validate}\left[\text{FunctionExpression}\right]\left(\text{cxt: CONTEXT, env: ENVIRONMENT}\right)\]
\[\left[\text{FunctionExpression} \equiv \text{function FunctionCommon}\right] \text{do}
\left[\text{Unchecked: BOOLEAN} \equiv \text{not}\text{.strict and Untyped[FunctionCommon]};\right.
\left[\text{Unchecked[FunctionCommon]} \equiv \text{unchecked};\right.
\left[\text{this: (none, inaccessible)} \equiv \text{unchecked} ? \text{inaccessible : none};\right.
\left[\text{F[FunctionExpression]} \equiv \text{ValidateStaticFunction}[\text{FunctionCommon}]\left(\text{cxt, env, this, unchecked}\right);\right.
\left[\text{FunctionExpression} \equiv \text{function Identifier FunctionCommon}\right] \text{do} ????
\text{end proc;}\]

Setup

\[\text{proc Setup}\left[\text{FunctionExpression}\right]\left(\right)\]
\[\left[\text{FunctionExpression} \equiv \text{function FunctionCommon}\right] \text{do Setup[FunctionCommon]();}\
\left[\text{FunctionExpression} \equiv \text{function Identifier FunctionCommon}\right] \text{do Setup[FunctionCommon]()}
\text{end proc;}\]

Evaluation

\[\text{proc Eval}\left[\text{FunctionExpression}\right]\left(\text{env: ENVIRONMENT, phase: PHASE}\right): \text{OBJORREF}\]
\[\left[\text{FunctionExpression} \equiv \text{function FunctionCommon}\right] \text{do}
\left[\text{if phase = compile then throw compileExpressionError end if};\right.
\left[\text{return instantiateOpenInstance(F[FunctionExpression], env)};\right.
\left[\text{FunctionExpression} \equiv \text{function Identifier FunctionCommon}\right] \text{do}
\left[\text{if phase = compile then throw compileExpressionError end if};\right.
\left[\text{return instantiateOpenInstance(F[FunctionExpression], env)}
\text{end proc;}\]

12.5 Object Literals

Syntax

\[
\text{ObjectLiteral} \equiv \\
\left[ \right] \mid \left\{ \text{FieldList} \right\}
\]

\[
\text{FieldList} \equiv \\
\text{LiteralField} \mid \text{FieldList, LiteralField}
\]

\[
\text{LiteralField} \equiv \text{FieldName : AssignmentExpression}\text{allowIn}
\]
FieldName []
    Identifier
    | String
    | Number

Validation

proc Validate[ObjectLiteral] (ext: CONTEXT, env: ENVIRONMENT)
    [ObjectLiteral [] { }] do nothing;
    [ObjectLiteral [] { FieldList } ] do Validate[FieldList](ext, env)
end proc;

proc Validate[FieldList] (ext: CONTEXT, env: ENVIRONMENT): STRING {}
    [FieldList [] LiteralField] do return Validate[LiteralField](ext, env);
    [FieldList0 [] FieldList1 , LiteralField] do
        names1: STRING {} Validate[FieldList1](ext, env);
        names2: STRING {} Validate[LiteralField](ext, env);
        if names1 ≠ names2 then throw syntaxError end if;
        return names1 end if
end proc;

proc Validate[LiteralField [] FieldName : AssignmentExpression] (ext: CONTEXT, env: ENVIRONMENT): STRING {}
    names: STRING {} Validate[FieldExpression](ext, env);
    Validate[AssignmentExpression](ext, env);
    return names end proc;

proc Validate[FieldName] (ext: CONTEXT, env: ENVIRONMENT): STRING {}
    [FieldName [] Identifier] do return {Name[Identifier]};
    [FieldName [] String] do return {Value[String]};
    [FieldName [] Number] do return {toString(Value[Number], compile)}
end proc;

Setup

Setup[ObjectLiteral] () propagates the call to Setup to every nonterminal in the expansion of ObjectLiteral.

Setup[FieldList] () propagates the call to Setup to every nonterminal in the expansion of FieldList.

Setup[LiteralField] () propagates the call to Setup to every nonterminal in the expansion of LiteralField.

proc Setup[FieldName] ()
    [FieldName [] Identifier] do nothing;
    [FieldName [] String] do nothing;
    [FieldName [] Number] do nothing
end proc;

Evaluation

    [ObjectLiteral [] { }] do
        if phase = compile then throw compileExpressionError end if;
        return new PROTOTYPE[[]]parent: objectPrototype, dynamicProperties: {}[]
end proc;
[ObjectLiteral { FieldList } ] do
  if phase = compile then throw compileExpressionError end if;
  properties: DynamicProperty {} [] Eval[FieldList](env, phase);
  return new Prototype { parent: objectPrototype, dynamicProperties: properties }
end proc;

  na: NamedArgument [] Eval[LiteralField](env, phase);
  return new DynamicProperty { name: na.name, value: na.value }
end proc;

proc Eval[LiteralField] FieldName : AssignmentExpression allowIn (env: Environment, phase: Phase): NamedArgument
  name: STRING [] Eval[FieldName](env, phase);
  r: ObjectRef [] Eval[AssignmentExpression allowIn](env, phase);
  value: Object [] readReference(r, phase);
  return NamedArgument { name: name, value: value }
end proc;

proc Eval[FieldName] (env: Environment, phase: Phase): String
  [FieldName [] Identifier] do return Name[Identifier];
  [FieldName [] String] do return Value[String];
  [FieldName [] Number] do return toString(Value[Number], compile)
end proc;

12.6 Array Literals

Syntax

ArrayLiteral [ ElementList ]

ElementList [] LiteralElement
  | ElementList , LiteralElement

LiteralElement [] «empty»
  | AssignmentExpression allowIn

Validation

proc Validate[ArrayLiteral [ ElementList ]] (cxt: Context, env: Environment)
  ???
end proc;
Setup

proc Setup[ArrayLiteral [] [ ElementList 1 ]] ()
????
end proc;

Evaluation

proc Eval[ArrayLiteral [] [ ElementList 1 ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
????
end proc;

12.7 Super Expressions

Syntax

SuperExpression []
  super
| super ParenExpression

Validation

proc Validate[SuperExpression] (ext: CONTEXT, env: ENVIRONMENT)
  [SuperExpression [] super] do
    if getEnclosingClass(env) = none or findThis(env, false) = none then
      throw syntaxError
    end if;
  end proc;

Setup

Setup[SuperExpression] () propagates the call to Setup to every nonterminal in the expansion of SuperExpression.

Evaluation

proc Eval[SuperExpression] (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT
  [SuperExpression [] super] do
    this: OBJECTI0PT [] findThis(env, false);
    Note that Validate ensured that this cannot be none at this point.
    if this = inaccessible then throw compileExpressionError end if;
    limit: CLASSOPT [] getEnclosingClass(env);
    Note that Validate ensured that limit cannot be none at this point.
    return readLimitedReference(this, limit, phase);
  [SuperExpression [] super ParenExpression] do
    r: OBJORREF [] Eval[ParenExpression](env, phase);
    limit: CLASSOPT [] getEnclosingClass(env);
    Note that Validate ensured that limit cannot be none at this point.
    return readLimitedReference(r, limit, phase)
  end proc;

readLimitedReference(r, phase) reads the reference, if any, inside r and returns the result, retaining limit. The object read from the reference is checked to make sure that it is an instance of limit or one of its descendants. If phase is compile, only compile-time expressions can be evaluated in the process of reading r.
proc readLimitedReference(r: OBJRef, limit: CLASS, phase: PHASE): OPTIONALLIMIT
  o: OBJECT [] readReference(r, phase);
  if o = null then return null end if;
  if o [] INSTANCE or not hasType(o, limit) then throw badValueError end if;
  return LIMITEDINSTANCE [instance: o, limit: limit]
end proc;

12.8 Postfix Expressions

Syntax

PostfixExpression []
  AttributeExpression |
  FullPostfixExpression |
  ShortNewExpression

AttributeExpression []
  SimpleQualifiedIdentifier |
  AttributeExpression MemberOperator |
  AttributeExpression Arguments

FullPostfixExpression []
  PrimaryExpression |
  ExpressionQualifiedIdentifier |
  FullNewExpression |
  FullPostfixExpression MemberOperator |
  SuperExpression MemberOperator |
  FullPostfixExpression Arguments |
  PostfixExpression [no line break] ++ |
  PostfixExpression [no line break] --

FullNewExpression [] new FullNewSubexpression Arguments

FullNewSubexpression []
  PrimaryExpression |
  QualifiedIdentifier |
  FullNewExpression |
  FullNewSubexpression MemberOperator |
  SuperExpression MemberOperator

ShortNewExpression [] new ShortNewSubexpression

ShortNewSubexpression []
  FullNewSubexpression |
  ShortNewExpression

Validation

Validate[PostfixExpression] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of PostfixExpression.

Strict[AttributeExpression]: BOOLEAN;

proc Validate[AttributeExpression] (ext: CONTEXT, env: ENVIRONMENT)
  [AttributeExpression [] SimpleQualifiedIdentifier] do
    Validate[SimpleQualifiedIdentifier](ext, env);
    Strict[AttributeExpression] [] ext.strict;
[AttributeExpression, MemberOperator] do
  Validate[AttributeExpression](ctx, env);
  Validate[MemberOperator](ctx, env);
[AttributeExpression, MemberOperator, Arguments] do
  Validate[AttributeExpression](ctx, env);
  Validate[Arguments](ctx, env)
end proc;

Strict[FullPostfixExpression]: BOOLEAN;

proc Validate[FullPostfixExpression] (ctx: CONTEXT, env: ENVIRONMENT)
[FullPostfixExpression PrimaryExpression] do
  Validate[PrimaryExpression](ctx, env);
[FullPostfixExpression ExpressionQualifiedIdentifier] do
  Validate[ExpressionQualifiedIdentifier](ctx, env);
  Strict[FullPostfixExpression] = ctx.strict;
[FullPostfixExpression FullNewExpression] do
  Validate[FullNewExpression](ctx, env);
[FullPostfixExpression, MemberOperator] do
  Validate[FullPostfixExpression](ctx, env);
  Validate[MemberOperator](ctx, env);
[FullPostfixExpression SuperExpression MemberOperator] do
  Validate[SuperExpression](ctx, env);
  Validate[MemberOperator](ctx, env);
[FullPostfixExpression, Arguments] do
  Validate[FullPostfixExpression](ctx, env);
  Validate[Arguments](ctx, env);
[PostfixExpression ++] do
  Validate[PostfixExpression](ctx, env);
[PostfixExpression --] do
  Validate[PostfixExpression](ctx, env)
end proc;

Validate[FullNewExpression] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of FullNewExpression.

Strict[FullNewSubexpression]: BOOLEAN;

proc Validate[FullNewSubexpression] (ctx: CONTEXT, env: ENVIRONMENT)
[FullNewSubexpression PrimaryExpression] do Validate[PrimaryExpression](ctx, env);
[FullNewSubexpression QualifiedIdentifier] do
  Validate[QualifiedIdentifier](ctx, env);
  Strict[FullNewSubexpression] = ctx.strict;
[FullNewSubexpression FullNewExpression] do Validate[FullNewExpression](ctx, env);
[FullNewSubexpression, MemberOperator] do
  Validate[FullNewSubexpression](ctx, env);
  Validate[MemberOperator](ctx, env);
[FullNewSubexpression SuperExpression MemberOperator] do
  Validate[SuperExpression](ctx, env);
  Validate[MemberOperator](ctx, env)
end proc;
Validate[ShortNewExpression] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ShortNewExpression.

Validate[ShortNewSubexpression] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ShortNewSubexpression.

Setup

Setup[PostfixExpression] () propagates the call to Setup to every nonterminal in the expansion of PostfixExpression.

Setup[AttributeExpression] () propagates the call to Setup to every nonterminal in the expansion of AttributeExpression.

Setup[FullPostfixExpression] () propagates the call to Setup to every nonterminal in the expansion of FullPostfixExpression.

Setup[FullNewExpression] () propagates the call to Setup to every nonterminal in the expansion of FullNewExpression.

Setup[FullNewSubexpression] () propagates the call to Setup to every nonterminal in the expansion of FullNewSubexpression.

Setup[ShortNewExpression] () propagates the call to Setup to every nonterminal in the expansion of ShortNewExpression.

Setup[ShortNewSubexpression] () propagates the call to Setup to every nonterminal in the expansion of ShortNewSubexpression.

Evaluation

[PostfixExpression [] AttributeExpression] do
  return Eval[AttributeExpression](env, phase);
[PostfixExpression [] FullPostfixExpression] do
  return Eval[FullPostfixExpression](env, phase);
[PostfixExpression [] ShortNewExpression] do
  return Eval[ShortNewExpression](env, phase)
end proc;

[AttributeExpression [] SimpleQualifiedIdentifier] do
  return LEXICAL.REFERENCE[env: env, variableMultiname: Multiname[SimpleQualifiedIdentifier],
  strict: Strict[AttributeExpression]]
[AttributeExpression [] AttributeExpression, MemberOperator] do
  r: OBJORREF [] Eval[AttributeExpression](env, phase);
  a: OBJECT [] readReference(r, phase);
  return Eval[MemberOperator](env, a, phase);
[AttributeExpression [] AttributeExpression, Arguments] do
  r: OBJORREF [] Eval[AttributeExpression](env, phase);
  f: OBJECT [] readReference(r, phase);
  base: OBJECT [] referenceBase(v);
  args: ARGUMENTLIST [] Eval[Arguments](env, phase);
  return call(base, f, args, phase)
end proc;

proc Eval[FullPostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[FullPostfixExpression [] PrimaryExpression] do
  return Eval[PrimaryExpression](env, phase);
[FullPostfixExpression [] ExpressionQualifiedIdentifier] do
    return LEXICALREFERENCE[env: env, variableMultiname: Multiname[ExpressionQualifiedIdentifier],
        strict: Strict[FullPostfixExpression]]
end proc;

[FullPostfixExpression [] FullNewExpression] do
    return Eval[FullNewExpression](env, phase);
end proc;

[FullPostfixExpression [] FullPostfixExpression, MemberOperator] do
    r: OBJORREF [] Eval[FullPostfixExpression](env, phase);
    a: OBJECT [] readReference(r, phase);
    return Eval[MemberOperator](env, a, phase);
end proc;

[FullPostfixExpression [] SuperExpression MemberOperator] do
    a: OBJECT[] readReference(r, phase);
    return Eval[MemberOperator](env, a, phase);
end proc;

[FullPostfixExpression [] FullPostfixExpression, Arguments] do
    r: OBJORREF [] Eval[FullPostfixExpression](env, phase);
    f: OBJECT [] readReference(r, phase);
    base: OBJECT [] referenceBase(r);
    args: ARGUMENTLIST [] Eval[Arguments](env, phase);
    return call(base, f, args, phase);
end proc;

[FullPostfixExpression [] PostfixExpression] do
    if phase = compile then throw compileExpressionError end if;
    r: OBJORREF [] Eval[PostfixExpression](env, phase);
    a: OBJECT [] readReference(r, phase);
    b: OBJECT [] plus(a, phase);
    writeReference(r, b, 1.0);
    return b;
end proc;

[FullPostfixExpression [] PostfixExpression] do
    if phase = compile then throw compileExpressionError end if;
    r: OBJORREF [] Eval[PostfixExpression](env, phase);
    a: OBJECT [] readReference(r, phase);
    b: OBJECT [] plus(a, phase);
    c: OBJECT [] subtract(b, 1.0);
    writeReference(r, c, phase);
    return b;
end proc;

proc Eval[FullNewExpression [] new FullNewSubexpression Arguments](
    env: ENVIRONMENT, phase: PHASE): OBJORREF
    r: OBJORREF [] Eval[FullNewSubexpression](env, phase);
    f: OBJECT [] readReference(r, phase);
    args: ARGUMENTLIST [] Eval[Arguments](env, phase);
    return construct(f, args, phase)
end proc;

    [FullNewSubexpression [] PrimaryExpression] do
        return Eval[PrimaryExpression](env, phase);
    end proc;
    [FullNewSubexpression [] QualifiedIdentifier] do
        return LEXICALREFERENCE[env: env, variableMultiname: Multiname[QualifiedIdentifier],
            strict: Strict[FullNewSubexpression]]
    end proc;
    [FullNewSubexpression [] FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    end proc;
[\text{SuperExpression} \text{ MemberOperator}] \text{ do }
\begin{align*}
r: \text{OBJORRef} & \quad \text{ Eval}([\text{FullNewSubexpression}], [\text{env}, \text{phase}];\\
a: \text{OBJECT} & \quad \text{readReference}(r, \text{phase});\\
\text{return} & \quad \text{Eval}([\text{MemberOperator}],[\text{env}, a, \text{phase}];\\
\end{align*}
[\text{FullNewSubexpression} \text{ SuperExpression} \text{ MemberOperator}] \text{ do }
\begin{align*}
a: \text{OBJOPTIONALLIMIT} & \quad \text{Eval}([\text{SuperExpression}],[\text{env}, \text{phase}]);\\
\text{return} & \quad \text{Eval}([\text{MemberOperator}],[\text{env}, a, \text{phase}])
\end{align*}
\text{end proc};

\text{proc} \text{Eval}([\text{ShortNewExpression} \text{ new} \text{ ShortNewSubexpression}]) \text{ (env: \text{ENVIRONMENT}, phase: \text{PHASE}): OBJORRef }
\begin{align*}
r: \text{OBJORRef} & \quad \text{Eval}([\text{ShortNewSubexpression}],[\text{env}, \text{phase}]);\\
f: \text{OBJECT} & \quad \text{readReference}(r, \text{phase});\\
\text{return} & \quad \text{construct}(f, \text{ARGUMENTLIST}[\text{positional: []}, \text{named: {}]} \text{ phase})
\end{align*}
\text{end proc};

\text{proc} \text{Eval}([\text{ShortNewSubexpression}]) \text{ (env: \text{ENVIRONMENT}, phase: \text{PHASE}): OBJORRef }
\begin{align*}
[\text{ShortNewSubexpression} & \text{ FullNewSubexpression}] \text{ do }\\
\text{return} & \quad \text{Eval}([\text{FullNewSubexpression}],[\text{env}, \text{phase}]);\\
[\text{ShortNewSubexpression} & \text{ ShortNewExpression}] \text{ do }\\
\text{return} & \quad \text{Eval}([\text{ShortNewExpression}],[\text{env}, \text{phase}])
\end{align*}
\text{end proc};

\text{referenceBase}(r) \text{ returns REFERENCE } r^\prime \text{'s base or null if there is none. The base's limit, if any, is ignored.}
\text{proc} \text{referenceBase}(r: \text{OBJORRef}): \text{OBJECT }
\begin{align*}
\text{case} \ r \ \text{of }\\
\text{OBJECT} & \quad \text{LEXICALREFERENCE } \text{return null;}\\
\text{DOTREFERENCE} & \quad \text{BRACKETREFERENCE } \text{ do }\\
a: \text{OBJOPTIONALLIMIT} & \quad r.\text{base};\\
\text{case} \ a \ \text{of }\\
\text{OBJECT} & \quad \text{return} \ a;\\
\text{LIMITEDINSTANCE} & \quad \text{return} \ a.\text{instance}
\end{align*}
\text{end case}
\text{end proc};

\text{proc} \text{call}(\text{this: OBJECT}, a: \text{OBJECT}, \text{args: ARGUMENTLIST}, \text{phase: PHASE}): \text{OBJECT }
\begin{align*}
\text{case} \ a \ \text{of }\\
\text{UNDEFINED} & \quad \text{NULL} \quad \text{BOOLEAN} \quad \text{GENERALNUMBER} \quad \text{CHARACTER} \quad \text{STRING} \quad \text{NAMESPACE} \quad \text{COMPOUNDATE tribute} \quad \text{Prototype} \quad \text{Package} \quad \text{Global} \quad \text{do }\\
\text{throw} & \quad \text{badValueError};\\
\text{CLASS} & \quad \text{return} \ a.\text{call}(\text{this, args, phase});\\
\text{INSTANCE} & \quad \text{do }\\
b: \text{SIMPLEINSTANCE} & \quad \text{CALLABLEINSTANCE } \text{ resolveAlias}(a);\\
\text{case} \ b \ \text{of }\\
\text{SIMPLEINSTANCE} & \quad \text{throw} \ \text{badValueError};\\
\text{CALLABLEINSTANCE} & \quad \text{do }\\
\text{Note that resolveAlias is not called when getting the env field.}\\
\text{return} & \quad b.\text{call}(\text{this, args, a.env, phase})
\end{align*}
\text{end case;}
\text{METHODCLOSURE} \quad \text{do }\\
code: \text{INSTANCE} & \quad a.\text{method.code};\\
\text{return} & \quad call(a.\text{this, code, args, phase})
\text{end case}
\text{end proc};
proc construct(a: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
  case a of
    UNDEFINED » NULL » BOOLEAN » GENERALNUMBER » CHARACTER » STRING » NAMESPACE »
    COMPOUNDATTRIBUTE » METHODCLOSURE » Prototype » PACKAGE » GLOBAL do
      throw badValueError;
    CLASS do return a.construct(args, phase);
    INSTANCE do
      b: SIMPLEINSTANCE » CALLABLEINSTANCE » resolveAlias(a);
      case b of
        SIMPLEINSTANCE do throw badValueError;
        CALLABLEINSTANCE do
          Note that resolveAlias is not called when getting the env field.
          return b.construct(args, a.env, phase)
      end case
    end case
  end case
end proc;

12.9 Member Operators

Syntax

MemberOperator []
  . QualifiedIdentifier
  | Brackets

Brackets []
  [ ]
  [ ListExpression allowIn ]
  [ NamedArgumentList ]

Arguments []
  ParenExpressions
  | ( NamedArgumentList )

ParenExpressions []
  ( )
  | ParenListExpression

NamedArgumentList []
  LiteralField
  | ListExpression allowIn , LiteralField
  | NamedArgumentList , LiteralField

Validation

Validate[MemberOperator] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of MemberOperator.

proc Validate[Brackets] (cxt: CONTEXT, env: ENVIRONMENT)
  Brackets [] [ ] do nothing;
  [Brackets [] [ ListExpression allowIn ] do Validate[ListExpression allowIn](cxt, env);
  [Brackets [] [ NamedArgumentList ] do Validate[NamedArgumentList](cxt, env)
end proc;
proc Validate[Arguments] (ctx: CONTEXT, env: ENVIRONMENT)
    [Arguments [] ParenExpressions] do Validate[ParenExpressions](ctx, env);
    [Arguments () ( NamedArgumentList )] do Validate[NamedArgumentList](ctx, env)
end proc;

Validate[ParenExpressions] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ParenExpressions.

    [NamedArgumentList [] LiteralField] do return Validate[LiteralField](ctx, env);
    [NamedArgumentList [] ListExpression allowin , LiteralField] do
        Validate[ListExpression allowin](ctx, env);
        return Validate[LiteralField](ctx, env);
    [NamedArgumentList0 [] NamedArgumentList1 , LiteralField] do
        names1: STRING {} [] Validate[NamedArgumentList1](ctx, env);
        names2: STRING {} [] Validate[LiteralField](ctx, env);
        if names1 [] names2 != {} then throw syntaxError end if;
        return names1 [] names2
end proc;

Setup

Setup[MemberOperator] () propagates the call to Setup to every nonterminal in the expansion of MemberOperator.

Setup[Brackets] () propagates the call to Setup to every nonterminal in the expansion of Brackets.

Setup[Arguments] () propagates the call to Setup to every nonterminal in the expansion of Arguments.

Setup[ParenExpressions] () propagates the call to Setup to every nonterminal in the expansion of ParenExpressions.

Setup[NamedArgumentList] () propagates the call to Setup to every nonterminal in the expansion of NamedArgumentList.

Evaluation

proc Eval[MemberOperator] (env: ENVIRONMENT, base: OPTIONAL LIMIT, phase: PHASE): OBJ OR REF
    [MemberOperator [] . QualifiedIdentifier] do
        return DOTREFERENCE [base: base, propertyMultiname: Multiname[QualifiedIdentifier]]
    [MemberOperator [] Brackets] do
        args: ARGUMENTLIST [] Eval[Brackets](env, phase);
        return BRACKETREFERENCE [base: base, args: args]
end proc;

proc Eval[Brackets] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
    [Brackets [] [ ]] do return ARGUMENTLIST [positional: [], named: {}]
    [Brackets [] [ ListExpression allowin []]] do
        positional: OBJECT [] EvalAsList[ListExpression allowin](env, phase);
        return ARGUMENTLIST [positional: positional, named: {}]
    [Brackets [] [ NamedArgumentList []]] do return Eval[NamedArgumentList](env, phase)
end proc;

proc Eval[Arguments] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
    [Arguments [] ParenExpressions] do return Eval[ParenExpressions](env, phase);
    [Arguments [] ( NamedArgumentList [])] do return Eval[NamedArgumentList](env, phase)
end proc;
proc Eval[ParenExpressions] (env: ENVIRONMENT, phase: PHASE): ARGUMENTLIST
[ParenExpressions [] ( ) ] do return ARGUMENTLIST[positional: [], named: {}
[ParenExpressions [] ParenListExpression] do
position: OBJECT[] EvalAsList(ParenListExpression)(env, phase);
return ARGUMENTLIST[positional: positional, named: {}]
end proc;

[NamedArgumentList [] LiteralField] do
name: NAMEDARGUMENT [] Eval[LiteralField](env, phase);
return ARGUMENTLIST[positional: [], named: {na}]
[NamedArgumentList [] ListExpression[allowIn], LiteralField] do
position: OBJECT[] EvalAsList[ListExpression[allowIn]](env, phase);
name: NAMEDARGUMENT [] Eval[LiteralField](env, phase);
return ARGUMENTLIST[positional: positional, named: {na}]
[NamedArgumentList0 [] NamedArgumentList1, LiteralField] do
args: ARGUMENTLIST [] Eval[NamedArgumentList1](env, phase);
name: NAMEDARGUMENT [] Eval[LiteralField](env, phase);
if some na2 args.named satisfies na2.name = na.name then
throw argumentMismatchError
end if;
return ARGUMENTLIST[positional: args.positional, named: args.named [] {na}]
end proc;

12.10 Unary Operators

Syntax

UnaryExpression []
PostfixExpression
| delete PostfixExpression
| void UnaryExpression
| typeof UnaryExpression
| ++ PostfixExpression
| -- PostfixExpression
| + UnaryExpression
| - UnaryExpression
| - NegatedMinLong
| ~ UnaryExpression
| ! UnaryExpression

Validation

Strict[UnaryExpression]: BOOLEAN;

proc Validate[UnaryExpression] (ctx: CONTEXT, env: ENVIRONMENT)
[UnaryExpression [] PostfixExpression] do Validate[PostfixExpression](ctx, env);
[UnaryExpression [] delete PostfixExpression] do
Validate[PostfixExpression](ctx, env);
Strict[UnaryExpression] [] ctx.strict;
[UnaryExpression0 [] void UnaryExpression1] do Validate[UnaryExpression1](ctx, env);
[UnaryExpression0 [] typeof UnaryExpression1] do
Validate[UnaryExpression1](ctx, env);
[UnaryExpression] ++ PostfixExpression] do Validate[PostfixExpression](ext, env);  
[UnaryExpression] -- PostfixExpression] do Validate[PostfixExpression](ext, env);  
[UnaryExpression] + PostfixExpression] do Validate[UnaryExpression](ext, env);  
[UnaryExpression] - PostfixExpression] do Validate[UnaryExpression](ext, env);  
[UnaryExpression] - NegatedMinLong] do nothing;  
[UnaryExpression] ~ UnaryExpression] do Validate[UnaryExpression](ext, env);  
[UnaryExpression] ! UnaryExpression] do Validate[UnaryExpression](ext, env)
end proc;

Setup

Setup[UnaryExpression] () propagates the call to Setup to every nonterminal in the expansion of UnaryExpression.

Evaluation

proc Eval[UnaryExpression](env: ENVIRONMENT, phase: PHASE): OBJOrREF
[UnaryExpression PostfixExpression] do return Eval[PostfixExpression](env, phase);  
[UnaryExpression delete PostfixExpression] do  
if phase = compile then throw compileExpressionError end if;
  r: OBJOrREF Eval[PostfixExpression](env, phase);  
return deleteReference(r, Strict[UnaryExpression], phase);
[UnaryExpression void UnaryExpression] do  
  r: OBJOrREF Eval[UnaryExpression](env, phase);
  readReference(r, phase);
return undefined;
[UnaryExpression] typeof UnaryExpression] do  
  r: OBJOrREF Eval[UnaryExpression](env, phase);
  a: OBJECT readReference(r, phase);  
case a of
        UNDEFINED do return “undefined”;
        NULL [] PROTOTYPE [] PACKAGE [] GLOBAL do return “object”;
        BOOLEAN do return “boolean”;
        LONG do return “long”;
        ULONG do return “ulong”;
        FLOAT32 do return “float”;
        FLOAT64 do return “number”;
        CHARACTER do return “character”;
        STRING do return “string”;
        NAMESPACE do return “namespace”;
        COMPOUNDATTRIBUTE do return “attribute”;
        CLASS [] METHODCLOSURE do return “function”;
        INSTANCE do return resolveAlias(a).typeofString
end case;
[UnaryExpression] ++ PostfixExpression] do  
if phase = compile then throw compileExpressionError end if;
  r: OBJOrREF Eval[PostfixExpression](env, phase);
  a: OBJECT readReference(r, phase);  
  b: OBJECT plus(a, phase);
  c: OBJECT add(b, 1.0|64|, phase);
  writeReference(r, c, phase);  
return c;
[UnaryExpression ~ PostfixExpression] do
  if phase = compile then throw compileExpressionError end if;
  r: OBJORREF Eval[PostfixExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  b: OBJECT  plus(a, phase);
  c: OBJECT  subtract(b, 1.0f64, phase);
  writeReference(r, c, phase);
  return c;
[UnaryExpression + UnaryExpression] do
  r: OBJORREF Eval[UnaryExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  return plus(a, phase);
[UnaryExpression - UnaryExpression] do
  r: OBJORREF Eval[UnaryExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  return minus(a, phase);
[UnaryExpression - NegatedMinLong] do return LONG[value: -2^63]
[UnaryExpression ~ UnaryExpression] do
  r: OBJORREF Eval[UnaryExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  return bitNot(a, phase);
[UnaryExpression ! UnaryExpression] do
  r: OBJORREF Eval[UnaryExpression](env, phase);
  a: OBJECT  readReference(r, phase);
  return logicalNot(a, phase)
end proc;

plus(a, phase) returns the value of the unary expression +a. If phase is compile, only compile-time operations are permitted.

proc plus(a: OBJECT, phase: PHASE): OBJECT
  return toGeneralNumber(a, phase)
end proc;

proc minus(a: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER  toGeneralNumber(a, phase);
  return generalNumberNegate(x)
end proc;

proc generalNumberNegate(x: GENERALNUMBER): GENERALNUMBER
  case x of
    LONG do return integerToLong(–(x.value));
    ULONG do return integerToULong(–(x.value));
    FLOAT32 do return float32Negate(x);
    FLOAT64 do return float64Negate(x)
  end case
end proc;
proc bitNot(a: OBJECT, phase: PHASE): OBJECT

x: GENERALNUMBER \( \rightarrow \) toGeneralNumber(a, phase);

  case x of
    LONG do i: \(-2^{63} \ldots 2^{63} - 1\) \( \rightarrow \) x.value; return LONG\[\text{value: bitwiseXor}(i, -1)\]\n    ULONG do i: \(0 \ldots 2^{64} - 1\) \( \rightarrow \) x.value;
      return ULONG\[\text{value: bitwiseXor}(i, 0xFFFFFFFFFFFFFFFF)\]\n    FLOAT32 \(\rightarrow\) FLOAT64 do
      i: \(-2^{31} \ldots 2^{31} - 1\) \( \rightarrow \) signedWrap32(truncateToIntger(x));
      return realToFloat64(bitwiseXor(i, -1))
  end case
end proc;

logicalNot(a, phase) returns the value of the unary expression \( \neg a \). If phase is compile, only compile-time operations are permitted.

proc logicalNot(a: OBJECT, phase: PHASE): OBJECT
  return not toBoolean(a, phase)
end proc;

12.11 Multiplicative Operators

Syntax

\[
\text{MultiplicativeExpression} \rightarrow \text{UnaryExpression} \\
| \text{MultiplicativeExpression} \times \text{UnaryExpression} \\
| \text{MultiplicativeExpression} / \text{UnaryExpression} \\
| \text{MultiplicativeExpression} \% \text{UnaryExpression}
\]

Validation

Validate[\text{MultiplicativeExpression}] (\text{cxt: CONTEXT, env: ENVIRONMENT}) propagates the call to Validate to every nonterminal in the expansion of \text{MultiplicativeExpression}.

Setup

Setup[\text{MultiplicativeExpression}] () propagates the call to Setup to every nonterminal in the expansion of \text{MultiplicativeExpression}.

Evaluation

proc Eval[\text{MultiplicativeExpression}] (\text{env: ENVIRONMENT, phase: PHASE}): OBJORREF

\[
\text{Eval[\text{MultiplicativeExpression}]} \rightarrow \text{Eval[\text{UnaryExpression}]}(\text{env, phase});
\]

\[\text{MultiplicativeExpression}_0 \rightarrow \text{MultiplicativeExpression} \times \text{UnaryExpression}\] do
  ra: OBJORREF \( \rightarrow \) Eval[\text{MultiplicativeExpression}](\text{env, phase});
  a: OBJECT \( \rightarrow \) readReference(ra, phase);
  rb: OBJORREF \( \rightarrow \) Eval[\text{UnaryExpression}](\text{env, phase});
  b: OBJECT \( \rightarrow \) readReference(rb, phase);
  return multiply(a, b, phase);
end do

\[\text{MultiplicativeExpression}_0 \rightarrow \text{MultiplicativeExpression} / \text{UnaryExpression}\] do
  ra: OBJORREF \( \rightarrow \) Eval[\text{MultiplicativeExpression}](\text{env, phase});
  a: OBJECT \( \rightarrow \) readReference(ra, phase);
  rb: OBJORREF \( \rightarrow \) Eval[\text{UnaryExpression}](\text{env, phase});
  b: OBJECT \( \rightarrow \) readReference(rb, phase);
  return divide(a, b, phase);
end do
proc multiply(a: Object, b: Object, phase: Phase): Object
x: GeneralNumber toGeneralNumber(a, phase);
y: GeneralNumber toGeneralNumber(b, phase);
if x Long ULong or y Long ULong then
  i: IntegerOpt checkInteger(x);
  j: IntegerOpt checkInteger(y);
  if i ≠ none and j ≠ none then
    k: Integer if y;
    if x ULong or y ULong then return integerToULong(k)
    else return integerToLong(k)
  end if
end if
return float64Multiply(toFloat64(x), toFloat64(y))
end proc;

proc divide(a: Object, b: Object, phase: Phase): Object
x: GeneralNumber toGeneralNumber(a, phase);
y: GeneralNumber toGeneralNumber(b, phase);
if x Long ULong or y Long ULong then
  i: IntegerOpt checkInteger(x);
  j: IntegerOpt checkInteger(y);
  if i ≠ none and j ≠ none and j ≠ 0 then
    q: Rational if y;
    if x ULong or y ULong then return rationalToULong(q)
    else return rationalToLong(q)
  end if
end if
return float64Divide(toFloat64(x), toFloat64(y))
end proc;

proc remainder(a: Object, b: Object, phase: Phase): Object
x: GeneralNumber toGeneralNumber(a, phase);
y: GeneralNumber toGeneralNumber(b, phase);
if x Long ULong or y Long ULong then
  i: IntegerOpt checkInteger(x);
  j: IntegerOpt checkInteger(y);
  if i ≠ none and j ≠ none and j ≠ 0 then
    q: Rational if y;
    k: Integer q ≥ 0 ? k: k
    r: Integer i - j\(k\);
    if x ULong or y ULong then return integerToULong(r)
    else return integerToLong(r)
  end if
end if
return float64Remainder(toFloat64(x), toFloat64(y))
end proc;
12.12 Additive Operators

Syntax

AdditiveExpression []
  MultiplicativeExpression
  | AdditiveExpression + MultiplicativeExpression
  | AdditiveExpression - MultiplicativeExpression

Validation

Validate[AdditiveExpression] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of AdditiveExpression.

Setup

Setup[AdditiveExpression] () propagates the call to Setup to every nonterminal in the expansion of AdditiveExpression.

Evaluation

  [AdditiveExpression [] MultiplicativeExpression] do
    return Eval[MultiplicativeExpression](env, phase);
  [AdditiveExpression0 [] AdditiveExpression1 + MultiplicativeExpression] do
    ra: OBJORREF [] Eval[AdditiveExpression1](env, phase);
    a: OBJECT [] readReference(ra, phase);
    rb: OBJORREF [] Eval[MultiplicativeExpression](env, phase);
    b: OBJECT [] readReference(rb, phase);
    return add(a, b, phase);
  [AdditiveExpression0 [] AdditiveExpression1 - MultiplicativeExpression] do
    ra: OBJORREF [] Eval[AdditiveExpression1](env, phase);
    a: OBJECT [] readReference(ra, phase);
    rb: OBJORREF [] Eval[MultiplicativeExpression](env, phase);
    b: OBJECT [] readReference(rb, phase);
    return subtract(a, b, phase)
end proc;
proc add(a: object, b: object, phase: phase): object
    ap: primitiveobject [] toprimitive(a, null, phase);
    bp: primitiveobject [] toprimitive(b, null, phase);
    if ap [] character [] string or bp [] character [] string then
        return toString(ap, phase) ⊕ toString(bp, phase)
    end if;
    x: generalnumber [] togeneralnumber(ap, phase);
    y: generalnumber [] togeneralnumber(bp, phase);
    if x [] long [] ulong or y [] long [] ulong then
        i: integeropt [] checkinteger(x);
        j: integeropt [] checkinteger(y);
        if i ≠ none and j ≠ none then
            k: integer [] i + j;
            if x [] ulong or y [] ulong then return integertoulong(k)
            else return integertolong(k)
        end if
    end if;
end proc;

proc subtract(a: object, b: object, phase: phase): object
    x: generalnumber [] togeneralnumber(a, phase);
    y: generalnumber [] togeneralnumber(b, phase);
    if x [] long [] ulong or y [] long [] ulong then
        i: integeropt [] checkinteger(x);
        j: integeropt [] checkinteger(y);
        if i ≠ none and j ≠ none then
            k: integer [] i - j;
            if x [] ulong or y [] ulong then return integertoulong(k)
            else return integertolong(k)
        end if
    end if;
end proc;

12.13 Bitwise Shift Operators

Syntax

ShiftExpression []
    AdditiveExpression
| ShiftExpression << AdditiveExpression
| ShiftExpression >> AdditiveExpression
| ShiftExpression >>> AdditiveExpression

Validation

Validate[ShiftExpression] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ShiftExpression.

Setup

Setup[ShiftExpression] () propagates the call to Setup to every nonterminal in the expansion of ShiftExpression.
Evaluation

proc Eval(ShiftExpression) (env: ENVIRONMENT, phase: PHASE): ObjOrRef
[ShiftExpression [] AdditiveExpression] do
  return Eval[AdditiveExpression](env, phase);
[ShiftExpression0 [] ShiftExpression1 << AdditiveExpression] do
  ra: ObjOrRef[] Eval[ShiftExpression1](env, phase);
  a: OBJECT[] readReference(ra, phase);
  rb: ObjOrRef[] Eval[AdditiveExpression](env, phase);
  b: OBJECT[] readReference(rb, phase);
  return shiftLeft(a, b, phase);
[ShiftExpression0 [] ShiftExpression1 >>> AdditiveExpression] do
  ra: ObjOrRef[] Eval[ShiftExpression1](env, phase);
  a: OBJECT[] readReference(ra, phase);
  rb: ObjOrRef[] Eval[AdditiveExpression](env, phase);
  b: OBJECT[] readReference(rb, phase);
  return shiftRightUnsigned(a, b, phase);
end proc;

proc shiftLeft(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER[] toGeneralNumber(a, phase);
  count: INTEGER[] truncateToInteger(toGeneralNumber(b, phase));
  case x of
    FLOAT32 [] FLOAT64 do
      i: {-2^31 ... 2^31 - 1} [] signedWrap32(truncateToInteger(x));
      count [] bitwiseAnd(count, 0x1F);
      i [] signedWrap32(bitwiseShift(i, count));
      return realToFloat64(i);
    LONG do
      count [] bitwiseAnd(count, 0x3F);
      i: {-2^63 ... 2^63 - 1} [] signedWrap64(bitwiseShift(x.value, count));
      return LONG[value: i]
    ULONG do
      count [] bitwiseAnd(count, 0x3F);
      i: {0 ... 2^64 - 1} [] unsignedWrap64(bitwiseShift(x.value, count));
      return ULONG[value: i]
  end case
end proc;
proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL.NUMBER  toGeneralNumber(a, phase);
count: INTEGER  truncateToInteger(toGeneralNumber(b, phase));
case x of
  FLOAT32  FLOAT64 do
    i: {–2\(^{31}\) ... 2\(^{31}\) – 1} [] signedWrap32(truncateToInteger(x));
    count [] bitwiseAnd(count, 0x1F);
    i [] bitwiseShift(i, –count);
    return realToFloat64(i);
  LONG do
    count [] bitwiseAnd(count, 0x3F);
    i: {–2\(^{63}\) ... 2\(^{63}\) – 1} [] bitwiseShift(x.value, –count);
    return LONG[]value: i[]
  ULONG do
    count [] bitwiseAnd(count, 0x3F);
    i: {–2\(^{63}\) ... 2\(^{63}\) – 1} [] bitwiseShift(unsignedWrap64(x.value), –count);
    return ULONG[]value: unsignedWrap64(i)[]
end case
end proc;

proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
x: GENERAL.NUMBER  toGeneralNumber(a, phase);
count: INTEGER  truncateToInteger(toGeneralNumber(b, phase));
case x of
  FLOAT32  FLOAT64 do
    i: {0 ... 2\(^{32}\) – 1} [] unsignedWrap32(truncateToInteger(x));
    count [] bitwiseAnd(count, 0x1F);
    i [] bitwiseShift(i, –count);
    return realToFloat64(i);
  LONG do
    count [] bitwiseAnd(count, 0x3F);
    i: {0 ... 2\(^{64}\) – 1} [] bitwiseShift(unsignedWrap64(x.value), –count);
    return LONG[]value: signedWrap64(i)[]
  ULONG do
    count [] bitwiseAnd(count, 0x3F);
    i: {0 ... 2\(^{64}\) – 1} [] bitwiseShift(x.value, –count);
    return ULONG[]value: i[]
end case
end proc;

12.14 Relational Operators

Syntax

\[
\text{RelationalExpression} \allowin \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} < \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} > \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} <= \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} >= \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} \text{is} \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} \text{as} \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} \text{in} \text{ShiftExpression} \\
| \text{RelationalExpression} \allowin \text{ShiftExpression} \text{instanceof} \text{ShiftExpression}
\]
RelationalExpression\(^{\text{min}}\) □
  ShiftExpression
  | RelationalExpression\(^{\text{min}}\) < ShiftExpression
  | RelationalExpression\(^{\text{min}}\) > ShiftExpression
  | RelationalExpression\(^{\text{min}}\) <= ShiftExpression
  | RelationalExpression\(^{\text{min}}\) >= ShiftExpression
  | RelationalExpression\(^{\text{min}}\) is ShiftExpression
  | RelationalExpression\(^{\text{min}}\) as ShiftExpression
  | RelationalExpression\(^{\text{min}}\) instanceof ShiftExpression

Validation

\[\text{Validate}[\text{RelationalExpression}\(^{\text{min}}\)] (\text{ctx: CONTEXT, env: ENVIRONMENT})\] propagates the call to \text{Validate} to every nonterminal in the expansion of \text{RelationalExpression}\(^{\text{min}}\).

Setup

\[\text{Setup}[\text{RelationalExpression}\(^{\text{min}}\)] ()\] propagates the call to \text{Setup} to every nonterminal in the expansion of \text{RelationalExpression}\(^{\text{min}}\).

Evaluation

\[
\text{proc Eval}[\text{RelationalExpression}\(^{\text{min}}\)] (\text{env: ENVIRONMENT, phase: PHASE}): \text{OBJORREF}
\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ ShiftExpression] do \text{return Eval[ShiftExpression]}(\text{env, phase});\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\), < ShiftExpression] do \text{ra: OBJORREF □ Eval[RelationalExpression]\(^{\text{min}}\)](\text{env, phase}); a: OBJECT □ readReference(ra, phase); rb: OBJORREF □ Eval[ShiftExpression](\text{env, phase}); b: OBJECT □ readReference(rb, phase); \text{return isLess}(a, b, phase);\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\), > ShiftExpression] do \text{ra: OBJORREF □ Eval[RelationalExpression]\(^{\text{min}}\)](\text{env, phase}); a: OBJECT □ readReference(ra, phase); rb: OBJORREF □ Eval[ShiftExpression](\text{env, phase}); b: OBJECT □ readReference(rb, phase); \text{return isLess}(b, a, phase);\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\), <= ShiftExpression] do \text{ra: OBJORREF □ Eval[RelationalExpression]\(^{\text{min}}\)](\text{env, phase}); a: OBJECT □ readReference(ra, phase); rb: OBJORREF □ Eval[ShiftExpression](\text{env, phase}); b: OBJECT □ readReference(rb, phase); \text{return isLessOrEqual}(a, b, phase);\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\), >= ShiftExpression] do \text{ra: OBJORREF □ Eval[RelationalExpression]\(^{\text{min}}\)](\text{env, phase}); a: OBJECT □ readReference(ra, phase); rb: OBJORREF □ Eval[ShiftExpression](\text{env, phase}); b: OBJECT □ readReference(rb, phase); \text{return isLessOrEqual}(b, a, phase);\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\) is ShiftExpression] do \text{??};\]

\[\text{[RelationalExpression}\(^{\text{min}}\) □ RelationalExpression}\(^{\text{min}}\) as ShiftExpression] do \text{??};\]

\[\text{[RelationalExpression}^\text{allowIn} □ RelationalExpression}^\text{allowIn} \text{ in ShiftExpression}] do \text{??};\]
[RelationalExpression] [] RelationalExpression instanceof ShiftExpression] do ????
end proc;

proc isLess(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  ap: PRIMITIVEOBJECT [] toPrimitive(a, null, phase);
  bp: PRIMITIVEOBJECT [] toPrimitive(b, null, phase);
  if ap [] CHARACTER [] STRING and bp [] CHARACTER [] STRING then
    return toString(ap, phase) < toString(bp, phase)
  end if;
  return generalNumberCompare(toGeneralNumber(ap, phase), toGeneralNumber(bp, phase)) = less
end proc;

proc isLessOrEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  ap: PRIMITIVEOBJECT [] toPrimitive(a, null, phase);
  bp: PRIMITIVEOBJECT [] toPrimitive(b, null, phase);
  if ap [] CHARACTER [] STRING and bp [] CHARACTER [] STRING then
    return toString(ap, phase) ≤ toString(bp, phase)
  end if;
  return generalNumberCompare(toGeneralNumber(ap, phase), toGeneralNumber(bp, phase)) [] {less, equal}
end proc;

12.15 Equality Operators

Syntax

EqualityExpression [] RelationalExpression
  | EqualityExpression == RelationalExpression
  | EqualityExpression != RelationalExpression
  | EqualityExpression === RelationalExpression
  | EqualityExpression !== RelationalExpression

Validation

ValidateEqualityExpression (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of EqualityExpression.

Setup

SetupEqualityExpression () propagates the call to Setup to every nonterminal in the expansion of EqualityExpression.

Evaluation

proc EvalEqualityExpression (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [EqualityExpression] [] RelationalExpression do
    return Eval[RelationalExpression](env, phase);
  [EqualityExpression] [] == RelationalExpression do
    ra: OBJORREF [] EvalEqualityExpression](env, phase);
    a: OBJECT [] readReference(ra, phase);
    rb: OBJORREF [] Eval[RelationalExpression](env, phase);
    b: OBJECT [] readReference(rb, phase);
    return isEqual(a, b, phase);
[EqualityExpression۱۰][EqualityExpression۱۱] != RelationalExpression۱۲] do
  ra: OBJORREFEval[EqualityExpression۱۲](env, phase);
  a: OBJECTreadReference(ra, phase);
  rb: OBJORREFEval[RelationalExpression۱۲](env, phase);
  b: OBJECTreadReference(rb, phase);
  c: BOOLEANisEqual(a, b, phase);
  return not c;
[EqualityExpression۱۰][EqualityExpression۱۱] === RelationalExpression۱۲] do
  ra: OBJORREFEval[EqualityExpression۱۲](env, phase);
  a: OBJECTreadReference(ra, phase);
  rb: OBJORREFEval[RelationalExpression۱۲](env, phase);
  b: OBJECTreadReference(rb, phase);
  return isStrictlyEqual(a, b, phase);
[EqualityExpression۱۰][EqualityExpression۱۱] !== RelationalExpression۱۲] do
  ra: OBJORREFEval[EqualityExpression۱۲](env, phase);
  a: OBJECTreadReference(ra, phase);
  rb: OBJORREFEval[RelationalExpression۱۲](env, phase);
  b: OBJECTreadReference(rb, phase);
  c: BOOLEANisEqual(a, b, phase);
  return not c
end proc;
proc isEqual(a: Object, b: Object, phase: Phase): Boolean
 tbl

1. if a is Object then return isEqual(a, b, phase)
2. elseif a is Object and b is Object then return isEqual(a, b, phase)
3. elseif a is String then return isEqual(a, b, phase)
4. elseif b is String then return isEqual(a, b, phase)
5. elseif a is Number then return isEqual(a, b, phase)
6. elseif b is Number then return isEqual(a, b, phase)
7. else return isEqual(a, b, phase)

end proc;

proc isStrictlyEqual(a: Object, b: Object, phase: Phase): Boolean
 tbl

1. if a is Object then return isStrictlyEqual(a, b, phase)
2. elseif a is Object and b is Object then return isStrictlyEqual(a, b, phase)
3. elseif a is String then return isStrictlyEqual(a, b, phase)
4. elseif b is String then return isStrictlyEqual(a, b, phase)
5. elseif a is Number then return isStrictlyEqual(a, b, phase)
6. elseif b is Number then return isStrictlyEqual(a, b, phase)
7. else return isEqual(a, b, phase)

end proc;

12.16 Binary Bitwise Operators

Syntax

BitwiseAndExpression
   EqualityExpression
   | BitwiseAndExpression & EqualityExpression

BitwiseXorExpression
   BitwiseAndExpression
   | BitwiseXorExpression ^ BitwiseAndExpression
\[ \text{BitwiseOrExpression} \mid \text{BitwiseXorExpression} \mid \text{BitwiseAndExpression} \]

Validation

\[ \text{Validate}[\text{BitwiseAndExpression}] (\text{ext: CONTEXT, env: ENVIRONMENT}) \] propagates the call to \text{Validate} to every nonterminal in the expansion of \text{BitwiseAndExpression}.

\[ \text{Validate}[\text{BitwiseXorExpression}] (\text{ext: CONTEXT, env: ENVIRONMENT}) \] propagates the call to \text{Validate} to every nonterminal in the expansion of \text{BitwiseXorExpression}.

\[ \text{Validate}[\text{BitwiseOrExpression}] (\text{ext: CONTEXT, env: ENVIRONMENT}) \] propagates the call to \text{Validate} to every nonterminal in the expansion of \text{BitwiseOrExpression}.

Setup

\[ \text{Setup}[\text{BitwiseAndExpression}] () \] propagates the call to \text{Setup} to every nonterminal in the expansion of \text{BitwiseAndExpression}.

\[ \text{Setup}[\text{BitwiseXorExpression}] () \] propagates the call to \text{Setup} to every nonterminal in the expansion of \text{BitwiseXorExpression}.

\[ \text{Setup}[\text{BitwiseOrExpression}] () \] propagates the call to \text{Setup} to every nonterminal in the expansion of \text{BitwiseOrExpression}.

Evaluation

\[ \text{proc Eval}[\text{BitwiseAndExpression}] (\text{env: ENVIRONMENT, phase: PHASE}): \text{OBJORREF} \]

\[ \text{Eval}[\text{BitwiseAndExpression}] (\text{expr: EXPRESSION, env: ENVIRONMENT}) \]

\[ \text{for } (\text{expr} = \text{BitwiseAndExpression}) \text{ do} \]

\[ \text{return Eval}[\text{BitwiseAndExpression}](\text{expr, env, phase}) \]

\[ \text{end proc} ; \]

\[ \text{proc Eval}[\text{BitwiseXorExpression}] (\text{expr: EXPRESSION, env: ENVIRONMENT}) \]

\[ \text{for } (\text{expr} = \text{BitwiseXorExpression}) \text{ do} \]

\[ \text{return Eval}[\text{BitwiseXorExpression}](\text{expr, env, phase}) \]

\[ \text{end proc} ; \]

\[ \text{proc Eval}[\text{BitwiseOrExpression}] (\text{expr: EXPRESSION, env: ENVIRONMENT}) \]

\[ \text{for } (\text{expr} = \text{BitwiseOrExpression}) \text{ do} \]

\[ \text{return Eval}[\text{BitwiseOrExpression}](\text{expr, env, phase}) \]

\[ \text{end proc} ; \]
[BitwiseOrExpression\^0[ ] [ BitwiseOrExpression\^1 | BitwiseXorExpression\^2] do
ra: OBJORREF[] Eval[BitwiseOrExpression\^1](env, phase);
a: OBJECT[] readReference(ra, phase);
rh: OBJORREF[] Eval[BitwiseXorExpression\^2](env, phase);
b: OBJECT[] readReference(rh, phase);
return bitOr(a, b, phase)
end proc;

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER[] toGeneralNumber(a, phase);
y: GENERALNUMBER[] toGeneralNumber(b, phase);
if x LONG ULONG or y LONG ULONG then
i: [-2\^63 ... 2\^63 - 1] [] signedWrap64(truncateToInteger(x));
j: [-2\^63 ... 2\^63 - 1] [] signedWrap64(truncateToInteger(y));
k: [-2\^63 ... 2\^63 - 1] [] bitwiseAnd(i, j);
if x ULONG or y ULONG then return ULONG[\value: unsignedWrap64(k)]
else return LONG[\value: k]
end if
else
i: [-2\^31 ... 2\^31 - 1] [] signedWrap32(truncateToInteger(x));
j: [-2\^31 ... 2\^31 - 1] [] signedWrap32(truncateToInteger(y));
return realToFloat64(bitwiseAnd(i, j))
end if
end proc;

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER[] toGeneralNumber(a, phase);
y: GENERALNUMBER[] toGeneralNumber(b, phase);
if x LONG ULONG or y LONG ULONG then
i: [-2\^63 ... 2\^63 - 1] [] signedWrap64(truncateToInteger(x));
j: [-2\^63 ... 2\^63 - 1] [] signedWrap64(truncateToInteger(y));
k: [-2\^63 ... 2\^63 - 1] [] bitwiseXor(i, j);
if x ULONG or y ULONG then return ULONG[\value: unsignedWrap64(k)]
else return LONG[\value: k]
end if
else
i: [-2\^31 ... 2\^31 - 1] [] signedWrap32(truncateToInteger(x));
j: [-2\^31 ... 2\^31 - 1] [] signedWrap32(truncateToInteger(y));
return realToFloat64(bitwiseXor(i, j))
end if
end proc;
proc bitOr(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER  
  toGeneralNumber(a, phase);
y: GENERALNUMBER  
  toGeneralNumber(b, phase);
if x \ LONG \ ULONG or y \ LONG \ ULONG then
  i: \{–2^{63} ... 2^{63} – 1\} \ signedWrap64(truncateToInteger(x));
  j: \{–2^{63} ... 2^{63} – 1\} \ signedWrap64(truncateToInteger(y));
  k: \{–2^{63} ... 2^{63} – 1\} \ bitwiseOr(i, j);
else return ULONG::value: unsignedWrap64(k)[]
end if
else
  i: \{–2^{31} ... 2^{31} – 1\} \ signedWrap32(truncateToInteger(x));
  j: \{–2^{31} ... 2^{31} – 1\} \ signedWrap32(truncateToInteger(y));
  return realToFloat64(bitwiseOr(i, j))
end if
end proc;

12.17 Binary Logical Operators

Syntax

 LogicalAndExpression[]  
  BitwiseOrExpression[[]
  | LogicalAndExpression[] & & BitwiseOrExpression[][]

 LogicalXorExpression[]  
  LogicalAndExpression[]
  | LogicalXorExpression[] ^ ^ LogicalAndExpression[][]

 LogicalOrExpression[]  
  LogicalXorExpression[]
  | LogicalOrExpression[] | | LogicalXorExpression[][]

Validation

 Validate[LogicalAndExpression[][]] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalAndExpression[][].

 Validate[LogicalXorExpression[][]] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalXorExpression[][].

 Validate[LogicalOrExpression[][]] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalOrExpression[][].

Setup

 Setup[LogicalAndExpression[][]] () propagates the call to Setup to every nonterminal in the expansion of LogicalAndExpression[][].

 Setup[LogicalXorExpression[][]] () propagates the call to Setup to every nonterminal in the expansion of LogicalXorExpression[][].

 Setup[LogicalOrExpression[][]] () propagates the call to Setup to every nonterminal in the expansion of LogicalOrExpression[][].
Evaluation

```
proc Eval[LogicalAndExpression](env: ENVIRONMENT, phase: PHASE): OBJORREF
  LogicalAndExpression[ LogicalOrExpression[ BitwiseOrExpression ]]
do
  return Eval[BitwiseOrExpression](env, phase);
  LogicalAndExpression[ LogicalAndExpression, LogicalOrExpression ]
do
  ra: OBJORREF Eval[LogicalAndExpression](env, phase);
  a: OBJECT readReference(ra, phase);
  if toBoolean(a, phase) then
    rb: OBJORREF Eval[BitwiseOrExpression](env, phase);
    return readReference(rb, phase)
  else return a
end if
end proc;

proc Eval[LogicalXorExpression](env: ENVIRONMENT, phase: PHASE): OBJORREF
  LogicalXorExpression[ LogicalOrExpression[ LogicalXorExpression ]]
do
  return Eval[LogicalAndExpression](env, phase);
  LogicalXorExpression[ LogicalXorExpression, LogicalAndExpression ]
do
  ra: OBJORREF Eval[LogicalXorExpression](env, phase);
  a: OBJECT readReference(ra, phase);
  rb: OBJORREF Eval[LogicalAndExpression](env, phase);
  b: OBJECT readReference(rb, phase);
  ba: BOOLEAN toBoolean(a, phase);
  bb: BOOLEAN toBoolean(b, phase);
  return ba xor bb
end proc;

proc Eval[LogicalOrExpression](env: ENVIRONMENT, phase: PHASE): OBJORREF
  LogicalOrExpression[ LogicalOrExpression[ LogicalXorExpression ]]
do
  return Eval[LogicalXorExpression](env, phase);
  LogicalOrExpression[ LogicalOrExpression, LogicalXorExpression ]
do
  ra: OBJORREF Eval[LogicalOrExpression](env, phase);
  a: OBJECT readReference(ra, phase);
  if toBoolean(a, phase) then return a
  else
    rb: OBJORREF Eval[LogicalXorExpression](env, phase);
    return readReference(rb, phase)
  end if
end proc;
```

12.18 Conditional Operator

Syntax

```
ConditionalExpression[ LogicalExpression ] | LogicalExpression ? AssignmentExpression : AssignmentExpression
NonAssignmentExpression[ LogicalExpression ] | LogicalExpression ? NonAssignmentExpression : NonAssignmentExpression
```


Validation

Validate[ConditionalExpression][EXT: CONTEXT, ENV: ENVIRONMENT] propagates the call to Validate to every nonterminal in the expansion of ConditionalExpression.

Validate[NonAssignmentExpression][EXT: CONTEXT, ENV: ENVIRONMENT] propagates the call to Validate to every nonterminal in the expansion of NonAssignmentExpression.

Setup

Setup[ConditionalExpression][EXT: CONTEXT] () propagates the call to Setup to every nonterminal in the expansion of ConditionalExpression.

Setup[NonAssignmentExpression][EXT: CONTEXT] () propagates the call to Setup to every nonterminal in the expansion of NonAssignmentExpression.

Evaluation

proc Eval[ConditionalExpression][EXT: ENVIRONMENT, PHASE: PHASE]: OBJORREF
  [ConditionalExpression][EXT: ] LogicalOrExpression[EXT: ] do
    return Eval[LogicalOrExpression][EXT: ](ENV, PHASE);
  endproc;

proc Eval[NonAssignmentExpression][EXT: ENVIRONMENT, PHASE: PHASE]: OBJORREF
  [NonAssignmentExpression][EXT: ] LogicalOrExpression[EXT: ] do
    return Eval[LogicalOrExpression][EXT: ](ENV, PHASE);
  endproc;
12.19 Assignment Operators

Syntax

AssignmentExpressionᵢ =
  ConditionalExpressionᵢ
  | PostfixExpression = AssignmentExpressionᵢ
  | PostfixExpression CompoundAssignment AssignmentExpressionᵢ
  | PostfixExpression LogicalAssignment AssignmentExpressionᵢ

CompoundAssignment ⋄

  *=
  /=
  %=
  +=
  -=
  <<=
  >>=
  >>>=
  &=
  ^=
  |=

LogicalAssignment ⋄

  &&=
  ||=
  ^^=
  ^=

Semantics

  tag andEq;
  tag xorEq;
  tag orEq;

Validation

proc Validate[AssignmentExpressionᵢ] (ctx: CONTEXT, env: ENVIRONMENT)
  [AssignmentExpressionᵢ ⋄ ConditionalExpressionᵢ] do
    Validate[ConditionalExpressionᵢ](ctx, env);
    [AssignmentExpressionᵢ ⋄ PostfixExpression = AssignmentExpressionᵢ] do
      Validate[PostfixExpression](ctx, env);
      Validate[AssignmentExpressionᵢ](ctx, env);
    [AssignmentExpressionᵢ ⋄ PostfixExpression CompoundAssignment AssignmentExpressionᵢ] do
      Validate[PostfixExpression](ctx, env);
      Validate[AssignmentExpressionᵢ](ctx, env);
    [AssignmentExpressionᵢ ⋄ PostfixExpression LogicalAssignment AssignmentExpressionᵢ] do
      Validate[PostfixExpression](ctx, env);
      Validate[AssignmentExpressionᵢ](ctx, env)
  end proc;
Setup

proc Setup[AssignmentExpression^l]( )
  [AssignmentExpression^l] do Setup[ConditionalExpression^l]( );
  [AssignmentExpression^l] do
    Setup[PostfixExpression]( );
    Setup[AssignmentExpression^l]( );
  [AssignmentExpression^l] do
    Setup[PostfixExpression CompoundAssignment AssignmentExpression^l]( );
    Setup[PostfixExpression]( );
    Setup[AssignmentExpression^l]( );
  end proc;

Evaluation

proc Eval[AssignmentExpression^l]( env: ENVIRONMENT, phase: PHASE): OBJORREF
  [AssignmentExpression^l] do Eval[ConditionalExpression^l](env, phase);
  [AssignmentExpression^l] do
    if phase = compile then throw compileExpressionError end if;
    ra: OBJORREF do Eval[PostfixExpression](env, phase);
    rb: OBJORREF do Eval[AssignmentExpression^l](env, phase);
    b: OBJECT do readReference(rb, phase);
    writeReference(ra, b, phase);
    return b;
  [AssignmentExpression^l] do
    if phase = compile then throw compileExpressionError end if;
    oLeft: OBJECT do readReference(oLeft, phase);
    oRight: OBJECT do readReference(oRight, phase);
    result: OBJECT do Op[CompoundAssignment](oLeft, oRight, phase);
    writeReference(oLeft, result, phase);
    return result;
[AssignmentExpression\[^0\]]_0\ PostfixExpression LogicalAssignment AssignmentExpression\[^1\] do
if phase = compile then throw compileExpressionError end if;
  rLeft: OBJORRef[] Eval[PostfixExpression](env, phase);
  oLeft: OBJECT[] readReference(rLeft, phase);
  bLeft: BOOLEAN[] toBoolean(oLeft, phase);
  result: OBJECT[] oLeft;
case Operator[LogicalAssignment] of
  {andEq} do
    if bLeft then
      result[] readReference(Eval[AssignmentExpression\[^1\]](env, phase), phase)
    end if;
  {xorEq} do
    bRight: BOOLEAN[] toBoolean(readReference(Eval[AssignmentExpression\[^1\]](env, phase), phase), phase);
    result[] bLeft xor bRight;
  {orEq} do
    if not bLeft then
      result[] readReference(Eval[AssignmentExpression\[^1\]](env, phase), phase)
    end if;
  end case;
writeReference(rLeft, result, phase);
return result
end proc;

Op[CompoundAssignment]: OBJECT[] OBJECT[] PHASE[] OBJECT;
Op[CompoundAssignment] fi *= multiply;
Op[CompoundAssignment] fi /= divide;
Op[CompoundAssignment] fi %= remainder;
Op[CompoundAssignment] fi += add;
Op[CompoundAssignment] fi -= subtract;
Op[CompoundAssignment] fi <<= shiftLeft;
Op[CompoundAssignment] fi >>= shiftRight;
Op[CompoundAssignment] fi >>>= shiftRightUnsigned;
Op[CompoundAssignment] fi &= bitAnd;
Op[CompoundAssignment] fi ^= bitXor;
Op[CompoundAssignment] fi |= bitOr;

Operator[LogicalAssignment]: {andEq, xorEq, orEq};
Operator[LogicalAssignment] fi &= andEq;
Operator[LogicalAssignment] fi ^= xorEq;
Operator[LogicalAssignment] fi |= orEq;

12.20 Comma Expressions

Syntax

ListExpression\[^0\] []
  AssignmentExpression\[^0\]
| ListExpression\[^1\] , AssignmentExpression\[^0\]

OptionalExpression []
  ListExpression\[^0\]
| «empty»
Validation

Validation[ListExpression
] (cxt: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of ListExpression
.

Setup

Setup[ListExpression
] () propagates the call to Setup to every nonterminal in the expansion of ListExpression
.

Evaluation

proc Eval[ListExpression
] (env: ENVIRONMENT, phase: PHASE): OBJORREF

  [ListExpression
  , AssignmentExpression
] do
    return Eval[AssignmentExpression
](env, phase);
  [ListExpression
  , AssignmentExpression
] do
    ra: OBJORREF  Eval[ListExpression
](env, phase);
    readReference(ra, phase);
    rb: OBJORREF  Eval[AssignmentExpression
](env, phase);
    return readReference(rb, phase)
  end proc;

proc EvalAsList[ListExpression
] (env: ENVIRONMENT, phase: PHASE): OBJECT[]

  [ListExpression
  , AssignmentExpression
] do
    r: OBJORREF  Eval[AssignmentExpression
](env, phase);
    elt: OBJECT  readReference(r, phase);
    return [elt];
  [ListExpression
  , AssignmentExpression
] do
    els: OBJECT[]  EvalAsList[ListExpression
](env, phase);
    r: OBJORREF  Eval[AssignmentExpression
](env, phase);
    elt: OBJECT  readReference(r, phase);
    return els ⊕ [elt]
  end proc;

12.21 Type Expressions

Syntax

TypeExpression
 NonAssignmentExpression

Validation

proc Validate[TypeExpression
 NonAssignmentExpression
] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[NonAssignmentExpression
](cxt, env)
end proc;

Setup and Evaluation

proc SetupAndEval[TypeExpression
 NonAssignmentExpression
] (env: ENVIRONMENT): CLASS
  Setup[NonAssignmentExpression
](cxt, env)
  r: OBJORREF  Eval[NonAssignmentExpression
](env, compile);
  o: OBJECT  readReference(r, compile);
  if o [CLASS then throw badValueError end if;
  return o
end proc;
13 Statements

Syntax

```plaintext
{abbrev, noShortIf, full}

Statement[1] [ ]
  ExpressionStatement Semicolon[2]
  SuperStatement Semicolon[2]
  Block
  LabeledStatement[3]
  IfStatement[4]
  SwitchStatement
  DoStatement Semicolon[2]
  WhileStatement[4]
  ForStatement[3]
  WithStatement
  ContinueStatement Semicolon[2]
  BreakStatement Semicolon[2]
  ReturnStatement Semicolon[2]
  ThrowStatement Semicolon[2]
  TryStatement

Substatement[5] [ ]
  EmptyStatement
  Statement[6]
  SimpleVariableDefinition Semicolon[2]
  Attributes [no line break] { Substatements }

Substatements [ ]
  «empty»
  SubstatementsPrefix Substatement[abbrev]

SubstatementsPrefix [ ]
  «empty»
  SubstatementsPrefix Substatement[full]

Semicolon[abbrev] [ ]
  ;
  VirtualSemicolon
  «empty»

Semicolon[noShortIf] [ ]
  ;
  VirtualSemicolon
  «empty»

Semicolon[full] [ ]
  ;
  VirtualSemicolon
```

Validation

```plaintext
do
  Validate[ExpressionStatement](cxt, env);
```
end proc;

Enabled[Substatement] : BOOLEAN;

proc Validate[Substatement](ext : CONTEXT, env : ENVIRONMENT, sl : LABEL {}, jt : Jumptargets) {
  [Substatement] [] EmptyStatement do nothing;
  [Substatement] [] Statement do Validate[Statement](ext, env, sl, jt, plural);
  [Substatement] [] SimpleVariableDefinition Semicolon do Validate[SimpleVariableDefinition](ext, env);
  [Substatement] [] Attributes [no line break] { Substatements } do
    Validate[Attributes](ext, env);
    Setup[Attributes]();
    attr ATTRIBUTE Eval[Attributes](env, compile);
    if attr BOOLEAN then throw badValueError end if;
    Enabled[Substatement] attr;
    if attr then Validate[Substatements](ext, env, jt) end if
end proc;

proc Validate[Substatements](ext : CONTEXT, env : ENVIRONMENT, jt : Jumptargets) {
  [Substatements] [] empty do nothing;
  [Substatements] [] SubstatementsPrefix Substatement do
    Validate[SubstatementsPrefix](ext, env, jt);
    Validate[Substatement](ext, env, {}, jt)
end proc;

proc Validate[SubstatementsPrefix](ext : CONTEXT, env : ENVIRONMENT, jt : Jumptargets) {
  [SubstatementsPrefix] [] empty do nothing;
  [SubstatementsPrefix] [] SubstatementsPrefix Substatement do
    Validate[SubstatementsPrefix](ext, env, jt);
    Validate[Substatement](ext, env, {}, jt)
end proc;

Setup

Setup[Statement]() propagates the call to Setup to every nonterminal in the expansion of Statement.
proc Setup[Substatement\^2] ()
    [Substatement\^0 □ EmptyStatement] do nothing;
    [Substatement\^0 □ Statement\^3] do Setup[Statement\^3]();
    [Substatement\^0 □ SimpleVariableDefinition Semicolon\^3] do
        Setup[SimpleVariableDefinition]();
    [Substatement\^0 □ Attributes [no line break] { Substatements } ] do
        if Enabled[Substatement\^0] then Setup[Substatements]() end if
end proc;

Setup[Substatements] () propagates the call to Setup to every nonterminal in the expansion of Substatements.

Setup[SubstatementsPrefix] () propagates the call to Setup to every nonterminal in the expansion of SubstatementsPrefix.

proc Setup[Semicolor\^0] ()
    [Semicolor\^0 □ .] do nothing;
    [Semicolor\^0 □ VirtualSemicolon] do nothing;
    [Semicolor\^1 abbrev □ «empty»] do nothing;
    [Semicolor\^1 noShort □ «empty»] do nothing
end proc;

Evaluation

proc Eval[Statement\^1] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [Statement\^1 □ ExpressionStatement Semicolor\^2] do
        return Eval[ExpressionStatement](env);
    [Statement\^1 □ SuperStatement Semicolor\^2] do return Eval[SuperStatement](env);
    [Statement\^1 □ Block] do return Eval[Block](env, d);
    [Statement\^1 □ LabeledStatement\^3] do return Eval[LabeledStatement\^3](env, d);
    [Statement\^1 □ IfStatement\^3] do return Eval[IfStatement\^3](env, d);
    [Statement\^1 □ SwitchStatement] do return Eval[SwitchStatement](env, d);
    [Statement\^1 □ DoStatement Semicolor\^1] do return Eval[DoStatement](env, d);
    [Statement\^1 □ WhileStatement\^3] do return Eval[WhileStatement\^3](env, d);
    [Statement\^1 □ ForStatement\^3] do return Eval[ForStatement\^3](env, d);
    [Statement\^1 □ WithStatement\^3] do return Eval[WithStatement\^3](env, d);
    [Statement\^1 □ ContinueStatement Semicolor\^2] do
        return Eval[ContinueStatement](env, d);
    [Statement\^1 □ BreakStatement Semicolor\^1] do return Eval[BreakStatement](env, d);
    [Statement\^1 □ ReturnStatement Semicolor\^1] do return Eval[ReturnStatement](env);
    [Statement\^1 □ ThrowStatement Semicolor\^1] do return Eval[ThrowStatement](env);
    [Statement\^1 □ TryStatement] do return Eval[TryStatement](env, d)
end proc;

proc Eval[Substatement\^3] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [Substatement\^3 □ EmptyStatement] do return d;
    [Substatement\^3 □ Statement\^3] do return Eval[Statement\^3](env, d);
    [Substatement\^3 □ SimpleVariableDefinition Semicolor\^3] do
        return Eval[SimpleVariableDefinition](env, d);
13.1 Empty Statement

Syntax

EmptyStatement

13.2 Expression Statement

Syntax

ExpressionStatement

Validation

Evaluation

  [ExpressionStatement {{function,}} ListExpression] (env, d)
  end proc;

proc Setup[ExpressionStatement] (env: ENVIRONMENT): OBJECT
  [ExpressionStatement {{function,}} ListExpression] ()
  end proc;

proc Eval[ExpressionStatement] (env: ENVIRONMENT): OBJECT
  [ExpressionStatement {{function,}} ListExpression] (env, run);
  return readReference(r, run)
  end proc;
13.3 Super Statement

Syntax

SuperStatement [] super Arguments

Validation

proc Validate[SuperStatement [] super Arguments] (ext: CONTEXT, env: ENVIRONMENT)
????
end proc;

Setup

proc Setup[SuperStatement [] super Arguments] ()
    Setup[Arguments]()
end proc;

Evaluation

proc Eval[SuperStatement [] super Arguments] (env: ENVIRONMENT): OBJECT
????
end proc;

13.4 Block Statement

Syntax

Block [] { Directives }

Validation

proc Validate[Block [] { Directives }] (ext: CONTEXT, env: ENVIRONMENT, jt: Jumptargets, pl: Plurality)
    compileFrame: BLOCKFRAME []
    new BLOCKFRAME [staticReadBindings: {}, staticWriteBindings: {}, plurality: pl]
    CompileFrame[Block] [] compileFrame;
    Validate[Directives](ext, [compileFrame] ⊕ env, jt, pl, none)
end proc;

proc ValidateUsingFrame[Block [] { Directives }]
    (ext: CONTEXT, env: ENVIRONMENT, jt: Jumptargets, pl: Plurality, frame: FRAME)
    Validate[Directives](ext, [frame] ⊕ env, jt, pl, none)
end proc;

Setup

proc Setup[Block [] { Directives }] ()
    Setup[Directives]()
end proc;
Evaluation

```plaintext
proc Eval[Block $\{\text{Directives}\}] (env: ENVIRONMENT, d: OBJECT): OBJECT
    compileFrame: BLOCKFRAME $\{\}$ CompileFrame[Block];
    runtimeFrame: BLOCKFRAME;
    case compileFrame.plurality of
        {singular} do runtimeFrame $\{\}$ compileFrame;
        {plural} do
            runtimeFrame $\{\}$ new BLOCKFRAME staticReadBindings: $\{\}$, staticWriteBindings: $\{\}$, plurality: singular
            instantiateFrame(compileFrame, runtimeFrame, $\{\}$ runtimeFrame $\oplus$ env)
        end case;
    return Eval[Directives]($\{\}$ runtimeFrame $\oplus$ env, d)
end proc;
```

```
proc EvalUsingFrame[Block $\{\}$ (Directives)] (env: ENVIRONMENT, frame: FRAME, d: OBJECT): OBJECT
    return Eval[Directives]($\{\}$ frame $\oplus$ env, d)
end proc;
```

```
CompileFrame[Block]: BLOCKFRAME;
```

13.5 Labeled Statements

Syntax

$LabeledStatement^0 \shortmid Identifier : Substatement^0$

Validation

```plaintext
proc Validate[LabeledStatement^0 $\{ Identifier : Substatement^0\}] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL $\{\}$, jt: JUMPTARGETS)
    name: STRING $\{\}$ Name[Identifier];
    if name $\notin$ jt.breakTargets then throw syntaxError end if;
    jt2: JUMPTARGETS $\{\}$ JUMPTARGETS breakTargets: jt.breakTargets $\{\}$ name, continueTargets: jt.continueTargets
    Validate[Substatement^0]($\{\}$ cxt, env, sl $\{\}$ name, jt2)
end proc;
```

Setup

```plaintext
proc Setup[LabeledStatement^0 $\{ Identifier : Substatement^0\}] ()
    Setup[Substatement^0]()
end proc;
```

Evaluation

```plaintext
proc Eval[LabeledStatement^0 $\{ Identifier : Substatement^0\}] (env: ENVIRONMENT, d: OBJECT): OBJECT
    try return Eval[Substatement^0](env, d)
    catch x: SEMANTICEXCEPTION do
        if x $\in\notin$ BREAK and x.label = Name[Identifier] then return x.value
    else throw x
    end if
    end try
end proc;
```
13.6 If Statement

Syntax

\[
\text{IfStatement}^{\text{abbrev}} \begin{cases} 
\text{if} & \text{ParenListExpression Substatement}^{\text{abbrev}} \\
\text{else} & \text{Substatement}^{\text{abbrev}} 
\end{cases} \\
\text{IfStatement}^{\text{full}} \begin{cases} 
\text{if} & \text{ParenListExpression Substatement}^{\text{full}} \\
\text{else} & \text{Substatement}^{\text{full}} 
\end{cases} \\
\text{IfStatement}^{\text{noShortIf}} \begin{cases} 
\text{if} & \text{ParenListExpression Substatement}^{\text{noShortIf}} \\
\text{else} & \text{Substatement}^{\text{noShortIf}} 
\end{cases}
\]

Validation

\[
\text{proc Validate}[\text{IfStatement}]^{\text{abbrev}} (cxt: \text{CONTEXT}, env: \text{ENVIRONMENT}, jt: \text{JUMP TARGETS}) \\
\begin{align*}
\text{Validate}[\text{ParenListExpression}]^{\text{abbrev}}(cxt, env) & ; \\
\text{Validate}[\text{Substatement}]^{\text{abbrev}}(cxt, env, \{\}, jt) & ;
\end{align*}
\]

\[
\text{proc Validate}[\text{IfStatement}]^{\text{full}} (cxt: \text{CONTEXT}, env: \text{ENVIRONMENT}, jt: \text{JUMP TARGETS}) \\
\begin{align*}
\text{Validate}[\text{ParenListExpression}]^{\text{full}}(cxt, env) & ; \\
\text{Validate}[\text{Substatement}]^{\text{full}}(cxt, env, \{\}, jt) & ;
\end{align*}
\]

\[
\text{proc Validate}[\text{IfStatement}]^{\text{noShortIf}} (cxt: \text{CONTEXT}, env: \text{ENVIRONMENT}, jt: \text{JUMP TARGETS}) \\
\begin{align*}
\text{Validate}[\text{ParenListExpression}]^{\text{noShortIf}}(cxt, env) & ; \\
\text{Validate}[\text{Substatement}]^{\text{noShortIf}}(cxt, env, \{\}, jt) & ;
\end{align*}
\]

end proc;

Setup

Setup[IfStatement]^{\text{abbrev}} () propagates the call to Setup to every nonterminal in the expansion of IfStatement^{\text{abbrev}}.

Evaluation

\[
\text{proc Eval}[\text{IfStatement}]^{\text{abbrev}} (env: \text{ENVIRONMENT}, d: \text{OBJECT}) : \text{OBJECT} \\
\begin{align*}
\text{if} & \text{ParenListExpression Substatement}^{\text{abbrev}}(cxt, env, run) ; \\
\text{readReference}(r, run) & ; \\
\text{if} & \text{toBoolean}(o, run) \text{ then return } \text{Eval}[\text{Substatement}]^{\text{abbrev}}(env, d) \text{ else return } d ;
\end{align*}
\]

\[
\text{if} & \text{ParenListExpression Substatement}^{\text{full}}(cxt, env, run) ; \\
\text{readReference}(r, run) & ; \\
\text{if} & \text{toBoolean}(o, run) \text{ then return } \text{Eval}[\text{Substatement}]^{\text{full}}(env, d) \text{ else return } d ;
\end{align*}
\]

\[
\text{if} & \text{ParenListExpression Substatement}^{\text{noShortIf}}(cxt, env, run) ; \\
\text{readReference}(r, run) & ; \\
\text{if} & \text{toBoolean}(o, run) \text{ then return } \text{Eval}[\text{Substatement}]^{\text{noShortIf}}(env, d) \text{ else return } \text{Eval}[\text{Substatement}]^{\text{full}}(env, d) ;
\end{align*}
\]

end proc;
13.7 Switch Statement

Syntax

\[
\text{SwitchStatement} \equiv \text{switch ParenListExpression} \; \{ \text{CaseStatements} \}
\]

\[
\text{CaseStatements} \equiv
\begin{align*}
\text{«empty»} & \\
| \quad \text{CaseLabel} & \\
| \quad \text{CaseLabel CaseStatementsPrefix CaseStatement}^\text{abbrev} &
\end{align*}
\]

\[
\text{CaseStatementsPrefix} \equiv
\begin{align*}
\text{«empty»} & \\
| \quad \text{CaseStatementsPrefix CaseStatement}^\text{full} &
\end{align*}
\]

\[
\text{CaseStatement}^\text{full} \equiv
\begin{align*}
\text{Substatement} & \\
| \quad \text{default} &
\end{align*}
\]

Validation

\[
\text{proc Validate}[\text{SwitchStatement} \equiv \text{switch ParenListExpression} \; \{ \text{CaseStatements} \}]
\]

\[
(\text{ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS})
\]

\[
\text{Validate}[\text{ParenListExpression}](\text{ext, env});
\]

13.8 Do-While Statement

Syntax

\[
\text{DoStatement} \equiv \text{do Substatement}^\text{abbrev while ParenListExpression}
\]

Validation

\[
\text{Labels[DoStatement]: LABEL{}};
\]
proc Validate[DoStatement [] do Substatement ^abbrev while ParenListExpression]  
    (ctx: CONTEXT, env: ENVIRONMENT, sl: LABEL {}; jt: JUMPTARGETS)  
    continueLabels: LABEL {} [] sl [] {default};  
    Labels[DoStatement] [] continueLabels;  
    jt2: JUMPTARGETS [] JUMPTARGETS[breakTargets: jt.breakTargets [] {default},  
        continueTargets: jt.continueTargets [] continueLabels];  
    Validate[Substatement ^abbrev](ctx, env, {}, jt2);  
    Validate[ParenListExpression](ctx, env)  
end proc;

Setup

Setup[DoStatement] () propagates the call to Setup to every nonterminal in the expansion of DoStatement.

Evaluation

proc Eval[DoStatement [] do Substatement ^abbrev while ParenListExpression]  
    (env: ENVIRONMENT, d: OBJECT): OBJECT  
    try  
        d1: OBJECT [] d;  
        while true do  
            try d1 [] Eval[Substatement ^abbrev](env, d1)  
                catch x: SEMANTICEXCEPTION do  
                    if x [] CONTINUE and x.label [] Labels[DoStatement] then d1 [] x.value  
                        else throw x  
                    end if  
                end try;  
            r: OBJORREF [] Eval[ParenListExpression](env, run);  
            o: OBJECT [] readReference(r, run);  
            if not toBoolean(o, run) then return d1 end if  
        end while  
        catch x: SEMANTICEXCEPTION do  
            if x [] BREAK and x.label = default then return x.value else throw x end if  
        end try  
    end proc;

13.9 While Statement

Syntax

WhileStatement[^] [] while ParenListExpression Substatement[^]

Validation

Labels[WhileStatement[^]]: LABEL {};

proc Validate[WhileStatement[^] [] while ParenListExpression Substatement[^]]  
    (ctx: CONTEXT, env: ENVIRONMENT, sl: LABEL {}; jt: JUMPTARGETS)  
    Validate[ParenListExpression](ctx, env);  
    continueLabels: LABEL {} [] sl [] {default};  
    Labels[WhileStatement[^]] [] continueLabels;  
    jt2: JUMPTARGETS [] JUMPTARGETS[breakTargets: jt.breakTargets [] {default},  
        continueTargets: jt.continueTargets [] continueLabels];  
    Validate[Substatement[^]](ctx, env, {}, jt2)  
end proc;
Setup

Setup[WhileStatement[^0]]() propagates the call to Setup to every nonterminal in the expansion of \texttt{WhileStatement[^0]}.

Evaluation

\textbf{proc} Eval[WhileStatement[^0] [while \\ ParenListExpression \\ Substatement[^3]] (env: \texttt{ENVIRONMENT}, d: \texttt{OBJECT}): \texttt{OBJECT} \textbf{try}
\begin{align*}
d & : \texttt{OBJECT} \rightarrow d; \\
\text{while toBoolean} & (\text{readReference(Eval}}[\text{ParenListExpression}] (env, \text{run}, \text{run}), \text{run}) \text{ do} \\
\text{try } d & \rightarrow \text{Eval}[\text{Substatement[^3]}](\text{env}, d) \\
\text{catch } x & : \texttt{SEMANTICEXCEPTION} \text{ do} \\
\text{if } x & \rightarrow \texttt{CONTINUE} \text{ and } x.label \rightarrow \texttt{Labels}[\text{WhileStatement[^3]}] \text{ then} \\
\text{d} & \rightarrow x.value \\
\text{else } & \text{throw } x \\
\text{end } & \text{if} \\
\text{end } & \text{try} \\
\text{end while}; \\
\text{return } & d \\
\text{catch } x & : \texttt{SEMANTICEXCEPTION} \text{ do} \\
\text{if } x & \rightarrow \texttt{BREAK} \text{ and } x.label = \texttt{default} \text{ then return } x.value \text{ else throw } x \\
\text{end } & \text{if} \\
\text{end } & \text{try} \\
\text{end } & \text{proc};
\end{align*}

13.10 For Statements

Syntax

ForStatement[^0] [empty]
\begin{align*}
\text{for} & (\text{ForInitialiser} ; \text{OptionalExpression} ; \text{OptionalExpression}) \text{ Substatement[^3]} \\
| & \text{for} (\text{ForInBinding in ListExpression}[^{allowIn}]) \text{ Substatement[^3]}
\end{align*}

ForInitialiser [empty]
\begin{align*}
\text{ListExpression}[^{non}] \\
| & \text{VariableDefinitionKind VariableBindingList}[^{non}] \\
| & \text{Attributes [no line break] VariableDefinitionKind VariableBindingList}[^{non}]
\end{align*}

ForInBinding [empty]
\begin{align*}
\text{PostfixExpression} \\
| & \text{VariableDefinitionKind VariableBinding}[^{non}] \\
| & \text{Attributes [no line break] VariableDefinitionKind VariableBinding}[^{non}]
\end{align*}

Validation

\textbf{proc} Validate[ForStatement[^3]] (\textit{ctx}: \texttt{CONTEXT}, \textit{env}: \texttt{ENVIRONMENT}, \textit{sl}: \texttt{LABEL}[, \textit{jt}: \texttt{JUMPTARGETS})
\begin{align*}
\text{ForStatement[^3]} [\text{for} (\text{ForInitialiser} ; \text{OptionalExpression} ; \text{OptionalExpression}) \text{ Substatement[^3]}] & \text{ do} \\
????; \\
\text{ForStatement[^3]} [\text{for} (\text{ForInBinding in ListExpression}[^{allowIn}]) \text{ Substatement[^3]}] & \text{ do} \\
????; \\
\text{end proc};
\end{align*}

Setup

\textbf{proc} Setup[ForStatement[^3]] ()
\begin{align*}
\text{ForStatement[^3]} [\text{for} (\text{ForInitialiser} ; \text{OptionalExpression} ; \text{OptionalExpression}) \text{ Substatement[^3]}] & \text{ do} \\
????; \\
\end{align*}
[ForStatement\^3 [ for ( ForInBinding in ListExpression\_allowed ) Substatement\^3 ] do 
  
end proc;

Evaluation

proc Eval[ForStatement\^3] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [ForStatement\^3 [ for ( ForInitialiser ; OptionalExpression ; OptionalExpression ) Substatement\^3 ] do 
  
  end proc;

13.11 With Statement

Syntax

WithStatement\^3 [ with ParenListExpression Substatement\^3 ]

Validation

proc Validate[WithStatement\^3 [ with ParenListExpression Substatement\^3 ]
  (ctx: CONTEXT, env: ENVIRONMENT, jt: JUMP_TARGETS)
  Validate[ParenListExpression](ctx, env);
  Validate[Substatement\^3](ctx, env, {}, jt)
end proc;

Setup

Setup[WithStatement\^3] () propagates the call to Setup to every nonterminal in the expansion of WithStatement\^3.

Evaluation

proc Eval[WithStatement\^3 [ with ParenListExpression Substatement\^3 ] (env: ENVIRONMENT, d: OBJECT): OBJECT
  
end proc;

13.12 Continue and Break Statements

Syntax

ContinueStatement
  continue
  | continue [no line break] Identifier

BreakStatement
  break
  | break [no line break] Identifier

Validation

proc Validate[ContinueStatement] (jt: JUMP_TARGETS)
  [ContinueStatement [ continue] do 
  if default [] jt.continueTargets then throw syntaxError end if;
end proc;
[ContinueStatement] continue [no line break] Identifier do
  if Name[Identifier] jt.continueTargets then throw syntaxError end if
end proc;

proc Validate[BreakStatement] (jt: JUMPTARGETS)
  [BreakStatement] break do
    if default jt.breakTargets then throw syntaxError end if;
  [BreakStatement] break [no line break] Identifier do
    if Name[Identifier] jt.breakTargets then throw syntaxError end if
  end proc;

Setup

proc Setup[ContinueStatement] ()
  [ContinueStatement] continue do nothing;
  [ContinueStatement] continue [no line break] Identifier do nothing
end proc;

proc Setup[BreakStatement] ()
  [BreakStatement] break do nothing;
  [BreakStatement] break [no line break] Identifier do nothing
end proc;

Evaluation

proc Eval[ContinueStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [ContinueStatement] continue do throw CONTINUE{value: d, label: default}]
  [ContinueStatement] continue [no line break] Identifier do
    throw CONTINUE{value: d, label: Name[Identifier]}
end proc;

  [BreakStatement] break do throw BREAK{value: d, label: default}]
  [BreakStatement] break [no line break] Identifier do
    throw BREAK{value: d, label: Name[Identifier]}
end proc;

13.13 Return Statement

Syntax

  ReturnStatement]
    return  
    | return [no line break] ListExpression

Validation

proc Validate[ReturnStatement] (ext: CONTEXT, env: ENVIRONMENT)
  [ReturnStatement] return do
    if getRegionalFrame(env) PARAMETERFRAME then throw syntaxError end if;
  [ReturnStatement] return [no line break] ListExpression do
    if getRegionalFrame(env) PARAMETERFRAME then throw syntaxError end if;
    Validate[ListExpression]((ext, env)
end proc;
Setup

Setup[ReturnStatement] () propagates the call to Setup to every nonterminal in the expansion of ReturnStatement.

Evaluation

proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
    [ReturnStatement [] return] do throw RETURNEDVALUE[value: undefined]
    [ReturnStatement [] return [no line break] ListExpression[allowIn] do
        r: ObjOrRef [] Eval[ListExpression[allowIn]](env, run);
        a: OBJECT [] readReference(r, run);
        throw RETURNEDVALUE[value: a]
    end proc;

13.14 Throw Statement

Syntax

ThrowStatement [] throw [no line break] ListExpression[allowIn]

Validation

proc Validate[ThrowStatement [] throw [no line break] ListExpression[allowIn]] (ctx: CONTEXT, env: ENVIRONMENT)
    Validate[ListExpression[allowIn]](ctx, env)
end proc;

Setup

proc Setup[ThrowStatement [] throw [no line break] ListExpression[allowIn]] ()
    Setup[ListExpression[allowIn]]()
end proc;

Evaluation

proc Eval[ThrowStatement [] throw [no line break] ListExpression[allowIn]] (env: ENVIRONMENT): OBJECT
    r: ObjOrRef [] Eval[ListExpression[allowIn]](env, run);
    a: OBJECT [] readReference(r, run);
    throw THROWNVALUE[value: a]
end proc;

13.15 Try Statement

Syntax

TryStatement []
    try Block CatchClauses
    | try Block CatchClausesOpt finally Block

CatchClausesOpt []
    «empty»
    | CatchClauses

CatchClauses []
    CatchClause
    | CatchClauses CatchClause

CatchClause [] catch ( Parameter ) Block
Validation

proc Validate[TryStatement] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [TryStatement try Block CatchClauses] do
        Validate[Block](ext, env, jt, plural);
        Validate[CatchClauses](ext, env, jt);
    [TryStatement try Block1 CatchClausesOpt finally Block2] do
        Validate[Block1](ext, env, jt, plural);
        Validate[CatchClausesOpt](ext, env, jt);
        Validate[Block2](ext, env, jt, plural)
end proc;

Validate[CatchClausesOpt] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS) propagates the call to Validate to every nonterminal in the expansion of CatchClausesOpt.

Validate[CatchClauses] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS) propagates the call to Validate to every nonterminal in the expansion of CatchClauses.

proc Validate[CatchClause catch ( Parameter ) Block] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
end proc;

Setup

Setup[TryStatement] () propagates the call to Setup to every nonterminal in the expansion of TryStatement.

Setup[CatchClausesOpt] () propagates the call to Setup to every nonterminal in the expansion of CatchClausesOpt.

Setup[CatchClauses] () propagates the call to Setup to every nonterminal in the expansion of CatchClauses.

proc Setup[CatchClause catch ( Parameter ) Block] ()
end proc;

Evaluation

  [TryStatement try Block CatchClauses] do
    try return Eval[Block](env, d)
    catch x: SEMANTICEXCEPTION do
      if x THROWNVALUE then throw x end if;
      exception: OBJECT x.value;
      r: OBJECT [reject] Eval[CatchClauses](env, exception);
      if r ≠ reject then return r else throw x end if
    end try;

[TryStatement]  try Block1, CatchClausesOpt finally Block2] do
  result: OBJECT | semanticEXCEPTION;
  try result [] Eval[Block3](env, d)
  catch x: semanticEXCEPTION do result [] x
  end try;
  if result [] thrownVALUE then
    exception: OBJECT | result.value;
    try
      r: OBJECT | {reject} [] Eval[CatchClausesOpt](env, exception);
      if r ≠ reject then result [] r end if
    catch y: semanticEXCEPTION do result [] y
    end try
  end if;
  Eval[Block2](env, undefined);
  case result of
    OBJECT do return result;
    semanticEXCEPTION do throw result
  end case
end proc;

proc Eval[CatchClausesOpt] (env: environment, exception: OBJECT): OBJECT | {reject}
  [CatchClausesOpt] «empty» do return reject;
  [CatchClausesOpt] CatchClauses do return Eval[CatchClauses](env, exception)
end proc;

proc Eval[CatchClauses] (env: environment, exception: OBJECT): OBJECT | {reject}
  [CatchClauses] CatchClause do return Eval[CatchClause](env, exception);
  [CatchClauses0] CatchClauses, CatchClause do
    r: OBJECT | {reject} [] Eval[CatchClauses](env, exception);
    if r ≠ reject then return r else return Eval[CatchClause](env, exception) end if
  end proc;

proc Eval[CatchClause] catch (Parameter) Block (env: environment, exception: OBJECT): OBJECT | {reject}
end proc;

14 Directives

Syntax

  Directive
  EmptyStatement
  | Statement
  | AnnotatableDirective
  | Attributes [no line break] AnnotatableDirective
  | Attributes [no line break] { Directives }
  | PackageDefinition
  | Pragma Semicolon
AnnotatableDirective
  ExportDefinition Semicolon
  VariableDefinition Semicolon
  FunctionDefinition
  ClassDefinition
  NamespaceDefinition Semicolon
  ImportDirective Semicolon
  UseDirective Semicolon

Directives
  «empty»
  DirectivesPrefix Directiveabbrev

DirectivesPrefix
  «empty»
  DirectivesPrefix Directivefull

Validation

[Directive] EmptyStatement do return cxt;
[Directive] Statement do
  if attr œ { none, true } then throw syntaxError end if;
  Validate[Statement](cxt, env, {}, jt, pl);
  return cxt;
  return Validate[AnnotatableDirective](cxt, env, pl, attr);
  Validate[Attributes](cxt, env);
  Setup[Attributes]();
  attr2: ATTRIBUTE = Eval[Attributes](env, compile);
  attr3: ATTRIBUTE = combineAttributes(attr, attr2);
  Enabled[Directive] attr3 ≠ false;
  if attr3 ≠ false then return Validate[AnnotatableDirective](cxt, env, pl, attr) else return cxt end if;
  Validate[Attributes](cxt, env);
  Setup[Attributes]();
  attr2: ATTRIBUTE = Eval[Attributes](env, compile);
  attr3: ATTRIBUTE = combineAttributes(attr, attr2);
  Enabled[Directive] attr3 ≠ false;
  if attr3 = false then return cxt end if;
  return Validate[Directives](cxt, env, jt, pl, attr3);
[Directive] PackageDefinition do
  if attr œ { none, true } then ???? else throw syntaxError end if;
[Directive] Pragma Semicolon do
  if attr œ { none, true } then return Validate[Pragma](cxt)
  else throw syntaxError end if
end proc;
proc Validate[AnnotatableDirective][]
  [AnnotatableDirective] ExportDefinition Semicolon do ???:
  [AnnotatableDirective] VariableDefinition Semicolon do Validate[VariableDefinition](cxt, env, attr);
  return cxt;
  [AnnotatableDirective] FunctionDefinition do Validate[FunctionDefinition](cxt, env, pl, attr);
  return cxt;
  [AnnotatableDirective] ClassDefinition do Validate[ClassDefinition](cxt, env, pl, attr);
  return cxt;
  [AnnotatableDirective] NamespaceDefinition Semicolon do Validate[NamespaceDefinition](cxt, env, pl, attr);
  return cxt;
  [AnnotatableDirective] UseDirective Semicolon do if attr {none, true} then return Validate[UseDirective](cxt, env)
  else throw syntaxError
end if
end proc;

  [Directives] «empty» do return cxt;
  [Directives] DirectivesPrefix Directive^abbrev do
    cxt2: CONTEXT Validate[DirectivesPrefix](cxt, env, jt, pl, attr);
  return Validate[Directive^abbrev](cxt2, env, jt, pl, attr)
end proc;

  [DirectivesPrefix] «empty» do return cxt;
  [DirectivesPrefix] DirectivesPrefix, Directive^alt do
    cxt2: CONTEXT Validate[DirectivesPrefix](cxt, env, jt, pl, attr);
  return Validate[Directive^alt](cxt2, env, jt, pl, attr)
end proc;

Setup

proc Setup[Directive^1] ()
  [Directive^1] EmptyStatement do nothing;
  [Directive^1] Statement^1 do Setup[Statement^1]();
    if Enabled[Directive^1] then Setup[AnnotatableDirective^1]() end if;
  [Directive^1] Attributes [no line break] (Directives) do
    if Enabled[Directive^1] then Setup[Directives]() end if;
  [Directive^1] PackageDefinition do ???:
  [Directive^1] Pragma Semicolon do nothing
end proc;
proc Setup[AnnotatableDirective] ()
  [AnnotatableDirective [] ExportDefinition Semicolon] do ???
  [AnnotatableDirective [] VariableDefinition Semicolon] do
    Setup[VariableDefinition]();
  [AnnotatableDirective [] FunctionDefinition] do Setup[FunctionDefinition]();
  [AnnotatableDirective [] ClassDefinition] do Setup[ClassDefinition]();
  [AnnotatableDirective [] NamespaceDefinition Semicolon] do nothing;
end proc;

Setup[Directives] () propagates the call to Setup to every nonterminal in the expansion of Directives.

Setup[DirectivesPrefix] () propagates the call to Setup to every nonterminal in the expansion of DirectivesPrefix.

Evaluation

  [Directive [] EmptyStatement] do return d;
  [Directive [] Statement] do return Eval[Statement](env, d);
  [Directive [] AnnotatableDirective] do return Eval[AnnotatableDirective](env, d);
    if Enabled[Directive] then return Eval[AnnotatableDirective](env, d)
    else return d
  end if;
  [Directive [] Attributes [no line break] { Directives }] do
    if Enabled[Directive] then return Eval[Directives](env, d) else return d end if;
  [Directive [] PackageDefinition] do ???
  [Directive [] Pragma Semicolon] do return d
end proc;

  [AnnotatableDirective [] ExportDefinition Semicolon] do ???
  [AnnotatableDirective [] VariableDefinition Semicolon] do
    return Eval[VariableDefinition](env, d);
  [AnnotatableDirective [] FunctionDefinition] do return d;
  [AnnotatableDirective [] ClassDefinition] do return Eval[ClassDefinition](env, d);
  [AnnotatableDirective [] NamespaceDefinition Semicolon] do return d;
end proc;

proc Eval[Directives] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Directives «empty»] do return d;
  [Directives DirectivesPrefix Directive abbrev] do
    o: OBJECT = Eval[DirectivesPrefix](env, d);
    return Eval[Directive abbrev](env, o)
end proc;


```plaintext
  [DirectivesPrefix «empty»] do return d;
  [DirectivesPrefix0 «DirectivesPrefix, Directive] do
    o: OBJECT = Eval[DirectivesPrefix](env, d);
    return Eval[Directive](env, o)
  end proc;

Enabled[Directive]: BOOLEAN;

14.1 Attributes

Syntax

  Attributes []
  | Attribute
  | AttributeCombination

  AttributeCombination [] Attribute [no line break] Attributes

  Attribute []
  | AttributeExpression
  | true
  | false
  | public
  | NonexpressionAttribute

  NonexpressionAttribute []
  | final
  | private
  | static

Validation

  Validate[Attributes] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of Attributes.

  Validate[AttributeCombination] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of AttributeCombination.

  Validate[Attribute] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of Attribute.

  proc Validate[NonexpressionAttribute] (ext: CONTEXT, env: ENVIRONMENT)
    [NonexpressionAttribute «final»] do nothing.
    [NonexpressionAttribute «private»] do
      if getEnclosingClass(env) = none then throw syntaxError end if,
    [NonexpressionAttribute «static»] do nothing
  end proc;

Setup

  Setup[Attributes] () propagates the call to Setup to every nonterminal in the expansion of Attributes.

  Setup[AttributeCombination] () propagates the call to Setup to every nonterminal in the expansion of AttributeCombination.

  Setup[Attribute] () propagates the call to Setup to every nonterminal in the expansion of Attribute.
```
proc Setup[NonexpressionAttribute] ()
[NonexpressionAttribute [] final] do nothing;
[NonexpressionAttribute [] private] do nothing;
[NonexpressionAttribute [] static] do nothing
end proc;

Evaluation

proc Eval[Attributes] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
[Attributes [] Attribute] do return Eval[Attribute](env, phase);
[Attributes [] AttributeCombination] do return Eval[AttributeCombination](env, phase)
end proc;

proc Eval[AttributeCombination] [] Attribute [no line break] Attributes
(env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
a: ATTRIBUTE [] Eval[Attribute](env, phase);
if a = false then return false end if;
b: ATTRIBUTE [] Eval[Attributes](env, phase);
return combineAttributes(a, b)
end proc;

proc Eval[Attribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
[Attribute [] AttributeExpression] do
r: OBJECTREF [] Eval[AttributeExpression](env, phase);
a: OBJECT [] readReference(r, phase);
if a [] ATTRIBUTE then throw badValueError end if;
return a;
[Attribute [] true] do return true;
[Attribute [] false] do return false;
[Attribute [] public] do return publicNamespace;
[Attribute [] NonexpressionAttribute] do
return Eval[NonexpressionAttribute](env, phase)
end proc;

proc Eval[NonexpressionAttribute] (env: ENVIRONMENT, phase: PHASE): ATTRIBUTE
[NonexpressionAttribute [] final] do
return COMPOUNDATTRIBUTE[namespaces: {}, explicit: false, dynamic: false, memberMod: final, overrideMod: none, prototype: false, unused: false]
[NonexpressionAttribute [] private] do
c: CLASSOPT [] getEnclosingClass(env);
Note that Validate ensured that c cannot be none at this point.
return c.privateNamespace;
[NonexpressionAttribute [] static] do
return COMPOUNDATTRIBUTE[namespaces: {}, explicit: false, dynamic: false, memberMod: static, overrideMod: none, prototype: false, unused: false]
end proc;

14.2 Use Directive

Syntax

UseDirective [] use namespace ParenListExpression
Validation

\[
\text{proc Validate[UseDirective use namespace ParenListExpression]}(\text{ext: CONTEXT, env: ENVIRONMENT}): \text{CONTEXT} \\
\text{Validate[ParenListExpression]}(\text{ext, env}); \\
\text{Setup[ParenListExpression]}(); \\
\text{values: OBJECT}[] \text{EvalAsList[ParenListExpression]}(\text{env, compile}); \\
\text{namespaces: NAMESPACE}[] \text{ included }; \\
\text{for each } v \text{ in values do} \\
\text{ if } v \text{ is NAMESPACE or v is namespaces then throw badValueError end if; } \\
\text{namespaces [] namespaces } v \\
\text{end for each; } \\
\text{return CONTEXT} \text{openNamespaces: ext.openNamespaces } \text{ namespaces, other fields from ext} \\
\text{end proc; }
\]

14.3 Import Directive

Syntax

\[
\text{ImportDirective}[] \\
\quad \text{import ImportBinding IncludesExcludes} \\
\quad | \text{import ImportBinding, namespace ParenListExpression IncludesExcludes} \\
\]

\[
\text{ImportBinding}[] \\
\quad \text{ ImportSource} \\
\quad | \text{ Identifier } = \text{ ImportSource} \\
\]

\[
\text{ImportSource}[] \\
\quad \text{String} \\
\quad | \text{ PackageName} \\
\]

\[
\text{IncludesExcludes}[] \\
\quad \text{ «empty»} \\
\quad | , \text{ exclude ( NamePatterns )} \\
\quad | , \text{ include ( NamePatterns )} \\
\]

\[
\text{NamePatterns}[] \\
\quad \text{ «empty»} \\
\quad | \text{ NamePatternList} \\
\]

\[
\text{NamePatternList}[] \\
\quad \text{ QualifiedIdentifier} \\
\quad | \text{ NamePatternList, QualifiedIdentifier} \\
\]

14.4 Pragma

Syntax

\[
\text{Pragma}[] \text{ use PragmalItems} \\
\]

\[
\text{PragmalItems}[] \\
\quad \text{ PragmalItem} \\
\quad | \text{ PragmalItems, PragmalItem} \\
\]

\[
\text{PragmalItem}[] \\
\quad \text{PragmaExpr} \\
\quad | \text{PragmaExpr ?} \\
\]
PragmaExpr []
  Identifier
  | Identifier ( PragmaArgument )

PragmaArgument []
  true
  | false
  | Number
  | - Number
  | - NegatedMinLong
  | String

Validation

proc Validate[Pragma [] use PragmaItems] (cxt: CONTEXT): CONTEXT
  return Validate[PragmaItems](cxt)
end proc;

proc Validate[PragmaItems] (cxt: CONTEXT): CONTEXT
  [PragmaItems [] PragmaItem] do return Validate[PragmaItem](cxt);
  [PragmaItems0 [] PragmaItems1, PragmaItem] do
    cxt2: CONTEXT [] Validate[PragmaItems0](cxt);
    return Validate[PragmaItem](cxt2)
  end proc;

proc Validate[PragmaItem] (cxt: CONTEXT): CONTEXT
  [PragmaItem [] PragmaExpr] do return Validate[PragmaExpr](cxt, false);
  [PragmaItem [] PragmaExpr?] do return Validate[PragmaExpr](cxt, true)
end proc;

  [PragmaExpr [] Identifier] do
    return processPragma(cxt, Name[Identifier], undefined, optional);
  [PragmaExpr [] Identifier ( PragmaArgument )] do
    arg: OBJECT [] Value[PragmaArgument];
    return processPragma(cxt, Name[Identifier], arg, optional)
  end proc;

Value[PragmaArgument]: OBJECT;
  Value[PragmaArgument [] true] = true;
  Value[PragmaArgument [] false] = false;
  Value[PragmaArgument [] Number] = Value[Number];
  Value[PragmaArgument [] - Number] = generalNumberNegate(Value[Number]);
  Value[PragmaArgument [] - NegatedMinLong] = LONG[value: -2^63]
  Value[PragmaArgument [] String] = Value[String];
proc processPragma(cxt: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN): CONTEXT
    if name = "strict" then
        if value [] \{true, undefined\} then
            return CONTEXT[strict: true], other fields from cxt\end if;
        if value = false then return CONTEXT[strict: false], other fields from cxt\end if;
        if name = "ecmascript" then
            if value [] \{undefined, 4.0_64\} then return cxt end if;
            if value [] \{1.0_64, 2.0_64, 3.0_64\} then
                An implementation may optionally modify cxt to disable features not available in ECMAScript Edition value other than subsequent pragmas.
                return cxt
            end if;
        end if;
        if optional then return cxt else throw badValueError end if
    end proc;

15 Definitions

15.1 Export Definition

Syntax

ExportDefinition [] export ExportBindingList

ExportBindingList [] ExportBinding
    ExportBinding | ExportBindingList, ExportBinding

ExportBinding [] FunctionName
    FunctionName = FunctionName

15.2 Variable Definition

Syntax

VariableDefinition [] VariableDefinitionKind VariableBindingList\allowbreak

VariableDefinitionKind []
    var
    | const

VariableBindingList[] [] VariableBinding
    VariableBinding | VariableBindingList, VariableBinding

VariableBinding[] [] TypedIdentifier VariableInitialisation

VariableInitialisation[] [] «empty»
    | = VariableInitialiser
VariableInitialiser
  AssignmentExpression | NonexpressionAttribute | AttributeCombination

TypedIdentifier
  Identifier | Identifier : TypeExpression

Validation

proc Validate[VariableDefinition [] VariableDefinitionKind VariableBindingList\allowIn]
  (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE)
  immutable: BOOLEAN [] Immutable[VariableDefinitionKind];
  Validate[VariableBindingList\allowIn](cxt, env, attr, immutable)
end proc;

Immutable[VariableDefinitionKind]: BOOLEAN;
Immutable[VariableDefinitionKind [] var] = false;
Immutable[VariableDefinitionKind [] const] = true;

Validate[VariableBindingList\allowIn]
  (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN) propagates the call to
  Validate to every nonterminal in the expansion of VariableBindingList\allowIn.

CompileEnv[VariableBinding\allowIn]: ENVIRONMENT;
CompileVar[VariableBinding\allowIn]: HOISTEDVAR VARIABLE INSTANCEVARIABLE;
OverriddenRead[VariableBinding\allowIn]: OVERRIDENMEMBER;
OverriddenWrite[VariableBinding\allowIn]: OVERRIDENMEMBER;
Multiname[VariableBinding\allowIn]: MULTINAME;
  (ctx: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOptNotFALSE, immutable: BOOLEAN)
Validate[TypedIdentifier[^3]](ctx, env);
Validate[VariableInitialisation[^5]](ctx, env);
CompileEnv[VariableBinding[^3]] □ env;
name: STRING □ Name[TypedIdentifier[^3]];
if not ctx.strict and getRegionalFrame(env) □ GLOBAL □ PARAMETERFRAME and not immutable and
  attr = none and not TypePresent[TypedIdentifier[^3]] then
    qname: QUALIFIEDNAME □ QUALIFIEDNAME[namespace: publicNamespace, id: name[]]
Multiname[VariableBinding[^3]] □ {qname};
CompileVar[VariableBinding[^3]] □ defineHoistedVar(env, name, undefined)
else
  a: COMPOUNDATTRIBUTE □ toCompoundAttribute(attr);
  if a.dynamic or a.prototype then throw definitionError end if;
memberMod: MEMBERMODIFIER □ a.memberMod;
if env[0] □ CLASS then if memberMod = none then memberMod □ final end if
else if memberMod ≠ none then throw definitionError end if
end if;
case memberMod of
  {none, static} do
    proc evalType(): CLASS
      type: CLASSOPT □ SetupAndEval[TypedIdentifier[^3]](env);
      if type = none then return objectClass end if;
      return type
    end proc;
    proc evalInitialiser(): OBJECT
      Setup[VariableInitialisation[^5]]();
      value: OBJECTOPT □ Eval[VariableInitialisation[^5]](env, compile);
      if value = none then throw compileExpressionError end if;
      return value
    end proc;
    initialValue: VARIABLEVALUE □ inaccessible;
    if immutable then initialValue □ evalInitialiser end if;
    v: VARIABLE □ new VARIABLE[type: evalType, value: initialValue, immutable: immutable[]]
    Multiname: MULTINAME □ defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit,
        readWrite, v);
    Multiname[VariableBinding[^3]] □ multiname;
    CompileVar[VariableBinding[^3]] □ v;
  {virtual, final} do
    c: CLASS □ env[0];
    proc evalInitialValue(): OBJECTOPT
      return Eval[VariableInitialisation[^5]](env, run)
    end proc;
    v: INSTANCEVARIABLE □ new INSTANCEVARIABLE[evalInitialValue: evalInitialValue,
        immutable: immutable, final: memberMod = final[]]
    os: OVERRIDESTATUSPAIR □ defineInstanceMember(c, ctx, name, a.namespaces, a.overrideMod,
        a.explicit, readWrite, v);
    CompileVar[VariableBinding[^3]] □ v;
    OverriddenRead[VariableBinding[^3]] □ os.readStatus.overriddenMember;
    OverriddenWrite[VariableBinding[^3]] □ os.writeStatus.overriddenMember;
  {constructor} do throw definitionError
  end case
end if
Validate[VariableInitialisation\textsuperscript{2}] (\textit{ctx}: CONTEXT, \textit{env}: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of VariableInitialisation\textsuperscript{1}.

Validate[VariableInitialiser\textsuperscript{2}] (\textit{ctx}: CONTEXT, \textit{env}: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of VariableInitialiser\textsuperscript{1}.

Name[TypedIdentifier\textsuperscript{1}]: STRING;

Name[TypedIdentifier\textsuperscript{1} \textit{Identifier}] = Name[Identifier];

Name[TypedIdentifier\textsuperscript{1} \textit{Identifier} : TypeExpression\textsuperscript{1}] = Name[Identifier];

TypePresent[TypedIdentifier\textsuperscript{2}]: BOOLEAN;

TypePresent[TypedIdentifier\textsuperscript{2} \textit{Identifier}] = false;

TypePresent[TypedIdentifier\textsuperscript{2} \textit{Identifier} : TypeExpression\textsuperscript{2}] = true;

\begin{verbatim}
proc Validate[TypedIdentifier\textsuperscript{2}] (\textit{ctx}: CONTEXT, \textit{env}: ENVIRONMENT) [TypedIdentifier\textsuperscript{2} \textit{Identifier}] do nothing;
  [TypedIdentifier\textsuperscript{2} \textit{Identifier} : TypeExpression\textsuperscript{2}] do
    Validate[TypeExpression\textsuperscript{2}](\textit{ctx}, \textit{env})
  end
end proc;
\end{verbatim}

Setup

\begin{verbatim}
proc Setup[VariableDefinition \textit{VariableDefinitionKind} VariableBindingList\textsuperscript{allowIn}] ()
  Setup[VariableBindingList\textsuperscript{allowIn}()]
end proc;
\end{verbatim}

Setup[VariableBindingList\textsuperscript{3}] () propagates the call to Setup to every nonterminal in the expansion of VariableBindingList\textsuperscript{3}.
proc Setup[VariableBinding\[\] TypedIdentifier\[\] VariableInitialisation\[\]]() 

env: ENVIRONMENT \[ CompileEnv[VariableBinding\[\]]; 
v: HOISTEDVAR \[ VARIABLE \[ INSTANCEVARIABLE \[ CompileVar[VariableBinding\[\]]; 

case v of 

    HOISTEDVAR do Setup[VariableInitialisation\[\]](); 
    VARIABLE do 
      type: CLASS \[ getVariableType(v, compile); 
      case v.value of 
        \{ inaccessible \} do Setup[VariableInitialisation\[\]](); 
        (\[] \[ Object do 
          v.value \[ inaccessible; 
          Setup[VariableInitialisation\[\]](); 
          try 
            value: OBJECTOPT \[ Eval[VariableInitialisation\[\]](env, compile); 
            if value ≠ none then 
              coercedValue: OBJECT \[ type.implicitCoerce(value); 
              v.value \[ coercedValue 
            end if 
            catch x: SEMANTICEXCEPTION do 
              if x ≠ compileExpressionError then throw x end if; 
            end catch 
          end try 
        end object 
      end case; 
    INSTANCEVARIABLE do 
      t: CLASSOPT \[ SetupAndEval[TypedIdentifier\[\]](env); 
      if t = none then 
        overriddenRead: OVERRIDENMEMBER \[ OverriddenRead[VariableBinding\[\]]; 
        overriddenWrite: OVERRIDENMEMBER \[ OverriddenWrite[VariableBinding\[\]]; 
        if overriddenRead \[ \{ none, potentialConflict \} then 
          Note that defineInstanceMember already ensured that overriddenRead \[ INSTANCEMETHOD. 
          t \[ overriddenRead.type 
        elseif overriddenWrite \[ \{ none, potentialConflict \} then 
          Note that defineInstanceMember already ensured that overriddenWrite \[ INSTANCEMETHOD. 
          t \[ overriddenWrite.type 
        else t \[ objectClass 
      end if 
    end case 
end proc; 

Setup[VariableInitialisation\[\]]() propagates the call to Setup to every nonterminal in the expansion of VariableInitialisation\[\]. 

Setup[VariableInitialiser\[\]]() propagates the call to Setup to every nonterminal in the expansion of VariableInitialiser\[\].
Evaluation

```javascript
proc Eval[VariableDefinition] (env: ENVIRONMENT, d: OBJECT)
    immutable: BOOLEAN [] Immutable[VariableDefinitionKind];
    Eval[VariableBindingList] (env, immutable);
    return d
end proc;

proc Eval[VariableBindingList] (env: ENVIRONMENT, immutable: BOOLEAN)
    [VariableBindingList] [ VariableBinding ] do Eval[VariableBinding] (env, immutable);
    Eval[VariableBindingList] (env, immutable);
end proc;

proc Eval[TypedIdentifier] (env: ENVIRONMENT, immutable: BOOLEAN)
    value: OBJECTOPT [] Eval[VariableInitialisation] (env, run);
    if value ≠ none then
        lexicalWrite(env, Multiname[VariableBinding], value, false, run)
    end if;
    VARIABLE do
        localFrame: FRAME [] env[0];
        members: STATICMEMBER [] { b.content | b | localFrame.staticWriteBindings such that
            b.qname Multiname[VariableBinding]};
        Note that the members set consists of exactly one VARIABLE element because localFrame was constructed with that VARIABLE inside Validate.
        v: VARIABLE [] the one element of members;
        if v.value = inaccessible then
            value: OBJECTOPT [] Eval[VariableInitialisation] (env, run);
            type: CLASS [] getVariableType(v, run);
            coercedValue: OBJECTU;
            if value ≠ none then coercedValue [ type.implicitCoerce(value)
                elif immutable then coercedValue [] uninitialised
                else coercedValue [] type.defaultValue
            end if;
            v.value [] coercedValue
        end if;
    INSTANCE_VARIABLE do nothing
end case
end proc;

proc Eval[VariableInitialisation] (env: ENVIRONMENT, phase: PHASE): OBJECTOPT
    [VariableInitialisation] [ empty ] do return none;
    [VariableInitialisation] = VariableInitialiser[] do
        Eval[VariableInitialiser] (env, phase)
end proc;

proc Eval[VariableInitialiser] (env: ENVIRONMENT, phase: PHASE): OBJECT
    [VariableInitialiser] [ AssignmentExpression ] do
        r: OBJORREF [] Eval[AssignmentExpression] (env, phase);
        return readReference(r, phase);
    [VariableInitialiser] [ NonexpressionAttribute ] do
        Eval[NonexpressionAttribute] (env, phase);
end proc;
```
VariableInitialiser \[\] AttributeCombination do
  return Eval[AttributeCombination](env, phase)
end proc;

proc SetupAndEval[TypedIdentifier \[\] Identifier do]
  return none;
end proc;

proc SetupAndEval[TypedIdentifier \[\] Identifier : TypeExpression do]
  return SetupAndEval[TypeExpression](env)
end proc;

**15.3 Simple Variable Definition**

**Syntax**

A SimpleVariableDefinition represents the subset of VariableDefinition expansions that may be used when the variable definition is used as a Substatement instead of a Directive in non-strict mode. In strict mode variable definitions may not be used as substatements.

SimpleVariableDefinition \[\] var UntypedVariableBindingList

UntypedVariableBindingList \[\]
  UntypedVariableBinding
  | UntypedVariableBinding , UntypedVariableBinding

UntypedVariableBinding \[\] Identifier VariableInitialisation\allowbreak

**Validation**

proc Validate[SimpleVariableDefinition \[\] var UntypedVariableBindingList](ext: CONTEXT, env: ENVIRONMENT)
  if ext.strict or getRegionalFrame(env) \[\] GLOBAL \[\] PARAMETERFRAME then
    throw syntaxError
  end if;
  Validate[UntypedVariableBindingList](ext, env)
end proc;

Validate[UntypedVariableBindingList](ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of UntypedVariableBindingList.

proc Validate[UntypedVariableBinding \[\] Identifier VariableInitialisation\allowbreak](ext: CONTEXT, env: ENVIRONMENT)
  Validate[VariableInitialisation\allowbreak](ext, env);
  defineHoistedVar(env, Name[Identifier], undefined)
end proc;

**Setup**

proc Setup[SimpleVariableDefinition \[\] var UntypedVariableBindingList]()
  Setup[UntypedVariableBindingList]()
end proc;

Setup[UntypedVariableBindingList]() propagates the call to Setup to every nonterminal in the expansion of UntypedVariableBindingList.

proc Setup[UntypedVariableBinding \[\] Identifier VariableInitialisation\allowbreak](
  Setup[VariableInitialisation\allowbreak]()
end proc;
Evaluation

**proc Eval**[SimpleVariableDefinition [] var UntypedVariableBindingList] (env: ENVIRONMENT, d: OBJECT): OBJECT
    Eval[UntypedVariableBindingList](env);
    return d
end proc;

**proc Eval**[UntypedVariableBindingList] (env: ENVIRONMENT)
    Eval[UntypedVariableBindingList](env);
end proc;

**proc Eval**[UntypedVariableBinding [] Identifier VariableInitialisation allowIn] (env: ENVIRONMENT)
    value: OBJECTOPT Eval[VariableInitialisation allowIn](env, run);
    if value ≠ none then
        qname: QUALIFIEDNAME QUALIFIEDNAME namespace: publicNamespace, id: Name[Identifier]
        lexicalWrite(env, {qname}, value, false, run)
    end if
end proc;

### 15.4 Function Definition

**Syntax**

```
FunctionDefinition [] function FunctionName FunctionCommon

FunctionName [] function FunctionName FunctionCommon

Identifier
    | get [no line break] Identifier
    | set [no line break] Identifier

FunctionCommon [] ( Parameters ) Result Block
```
Validation

proc Validate[FunctionDefinition {} function FunctionName FunctionCommon]
  (ext: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOptNotFalse)
  name: STRING [] Name[FunctionName];
  kind: FUNCTIONKind [] Kind[FunctionName];
  a: COMPOUNDAttribute [] toCompoundAttribute(attr);
  if a dynamic then throw definitionError end if;
  unchecked: BOOLEAN [] not ext strict and env[0] [] CLASS and kind = normal and Untyped[FunctionCommon];
  Unchecked[FunctionCommon] [] unchecked;
  prototype: BOOLEAN [] unchecked or a.prototype;
  memberMod: MEMBERMODIFIER [] a.memberMod;
  if env[0] [] CLASS then if memberMod = none then memberMod [] virtual end if
  else if memberMod ≠ none then throw definitionError end if
  end if;
  if prototype and (kind ≠ normal or memberMod = constructor) then
    throw definitionError
  end if;
  this: {none, inaccessible} [] none;
  if prototype or memberMod [] {constructor, virtual, final} then this [] inaccessible
  end if;
  case memberMod of
    {none, static} do
      f: INSTANCE [] OPENINSTANCE;
      if kind [] {get, set} then ???
      else /[] ValidateStaticFunction[FunctionCommon](ext, env, this, prototype)
      end if;
      if pl = singular then f [] instantiateOpenInstance(f, env) end if;
      if unchecked and attr = none and
        (env[0] [] GLOBAL or (env[0] [] BLOCKFRAME and env[0] [] PARAMETERFRAME)) then
        defineHoistedVar(env, name, f)
      else
        v: VARIABLE [] new VARIABLE[] type: functionClass, value: f, immutable: true[]
        defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
      end if;
    {virtual, final} do ???
    {constructor} do ???
  end case
end proc;

Kind[FunctionName]: FUNCTIONKind;
  Kind[FunctionName [] Identifier] = normal;
  Kind[FunctionName [] get [no line break] Identifier] = get;
  Kind[FunctionName [] set [no line break] Identifier] = set;

Name[FunctionName]: STRING;
  Name[FunctionName [] Identifier] = Name[Identifier];
  Name[FunctionName [] get [no line break] Identifier] = Name[Identifier];
  Name[FunctionName [] set [no line break] Identifier] = Name[Identifier];

Untyped[FunctionCommon] [] ( Parameters ) Result Block]: BOOLEAN = Untyped[Parameters] and Untyped[Result];

Unchecked[FunctionCommon]: BOOLEAN;

CompileEnv[FunctionCommon]: ENVIRONMENT;
CompileFrame[FunctionCommon]: PARAMETERFRAME;

Signature[FunctionCommon]: SIGNATURE;

proc Validate[FunctionCommon] ( Parameters ) Result Block
  (ext: CONTEXT, env: ENVIRONMENT, this: {none, inaccessible}, prototype: BOOLEAN): INTEGER
  compileFrame: PARAMETERFRAME new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}],
  plurality: plural, this: this, prototype: prototype[],
  compileEnv: ENVIRONMENT compileFrame @ new PARAMETERFRAME compileFrame;
  CompileEnv[FunctionCommon] compileFrame;
  Validate[Result](ext, compileEnv);
  Validate[Block](ext, compileEnv, JUMPTARGETS breakTargets: {}, continueTargets: {} plural);
  return nFixedParameters
end proc;

proc ValidateStaticFunction[FunctionCommon] ( Parameters ) Result Block
  (ext: CONTEXT, env: ENVIRONMENT, this: {none, inaccessible}, prototype: BOOLEAN): OPENINSTANCE
  nFixedParameters: INTEGER Validate[FunctionCommon](ext, env, this, prototype);
  if prototype then ???
  else
    initialSlots: SLOT[] new SLOT[] id: findInstanceMember(functionClass, QUALIFIEDNAME, namespace: publicNamespace, id: "length", read),
    value: RealsToFloat64(nFixedParameters[]);
  end if
end proc;

Setup

proc Setup[FunctionDefinition] function FunctionName FunctionCommon ()
  Setup[FunctionCommon]()
end proc;

proc Setup[FunctionCommon] ( Parameters ) Result Block ()
end proc;

Evaluation

proc EvalNormalCall[FunctionCommon] ( Parameters ) Result Block
  (this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
  if phase = compile then throw compileExpressionError end if;
  runtimeFrame: PARAMETERFRAME new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}],
  plurality: singular, this: none, prototype: false[
  instantiateFrame(CompileFrame[FunctionCommon], runtimeFrame, runtimeFrame @ runtimeEnv);
  assignArguments(runtimeFrame, Signature[FunctionCommon], Unchecked[FunctionCommon], args);
  result: OBJECT;
  try Eval[Block](runtimeFrame @ runtimeEnv, undefined); result undefined
    catch x: SEMANTICEXCEPTION do
      if x[] RETURNEDVALUE then result x.value else throw x end if
    end try;
  return result
end proc;
proc EvalPrototypeCall[FunctionCommon] (Parameters) Result Block
   (this: OBJECT, args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
   if phase = compile then throw compileExpressionError end if;
   runtimeThis: OBJECT ¨ this;
   g: PACKAGE ¨ GLOBAL ¦ getPackageOrGlobalFrame(runtimeEnv);
   if runtimeThis œ {null, undefined} and g œ GLOBAL then runtimeThis œ g end if;
   runtimeFrame: PARAMETERFRAME ¨ new PARAMETERFRAME[staticReadBindings: {}, staticWriteBindings: {}], plurality: singular, this: runtimeThis, prototype: true]\]
   instantiateFrame(CompileFrame[FunctionCommon], runtimeFrame, [runtimeFrame] ⊕ runtimeEnv);
   assignArguments(runtimeFrame, Signature[FunctionCommon], Unchecked[FunctionCommon], args);
   result: OBJECT;
   try Eval[Block](runtimeFrame] ⊕ runtimeEnv, undefined); result œ undefined
   catch x: SEMANTICEXCEPTION do
      if x [RETURNEDVALUE then result œ x.value else throw x end if
   end try;
   return result
end proc;

proc EvalPrototypeConstruct[FunctionCommon] (Parameters) Result Block
   (args: ARGUMENTLIST, runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
   ????
end proc;

proc assignArguments(runtimeFrame: PARAMETERFRAME, sig: SIGNATURE, unchecked: BOOLEAN, 
      args: ARGUMENTLIST)
   ????
end proc;

Syntax

Parameters [«empty» | AllParameters

AllParameters [ Parameter | Parameter , AllParameters | OptionalParameters

OptionalParameters [ OptionalParameter | OptionalParameter , OptionalParameters | RestAndNamedParameters

RestAndNamedParameters [ NamedParameters | RestParameter | RestParameter , NamedParameters | NamedRestParameter

NamedParameters [ NamedParameter | NamedParameter , NamedParameters

Parameter [ TypedIdentifier| allowIn | const TypedIdentifier| allowIn
OptionalParameter \[ Parameter = AssignmentExpression_{allow} \]

TypedInitialiser \[ TypedIdentifier_{allow} = AssignmentExpression_{allow} \]

NamedParameter \[ named TypedInitialiser
| const named TypedInitialiser
| named const TypedInitialiser \]

RestParameter \[...
| ... Parameter \]

NamedRestParameter \[...
| ... const named Identifier
| ... named const Identifier \]

Result \[«empty»
| : TypeExpression_{allow} \]

Validation

Untyped[Parameters]: BOOLEAN;
Untyped[Parameters «empty»] = true;
Untyped[Parameters AllParameters] = ????;

func Validate[Parameters] (ext: CONTEXT, env: ENVIRONMENT): INTEGER
[Parameters «empty»] do return 0;
[Parameters AllParameters] do ????
end func;

Untyped[Result]: BOOLEAN;
Untyped[Result «empty»] = true;
Untyped[Result : TypeExpression_{allow}] = false;

Validate[Result] (ext: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of Result.

15.5 Class Definition

Syntax

ClassDefinition \[ class Identifier Inheritance Block \]

Inheritance \[«empty»
| extends TypeExpression_{allow} \]

Validation

Class[ClassDefinition]: CLASS;
proc Validate [ClassDefinition [] class Identifier Inheritance Block] 
  (ctx: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOptNotFALSE) 
  if pl ≠ singular then throw syntaxError end if; 
  superclass: CLASS [] Validate [Inheritance] (ctx, env); 
  a: COMPOUNDATTRIBUTE [] toCompoundAttribute (attr); 
  if not superclass.complete or superclass.final then throw definitionError end if; 
  proc call (this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT 
    ????
  end proc; 
  proc construct (args: ARGUMENTLIST, phase: PHASE): OBJECT 
    ????
  end proc; 
  prototype: OBJECT [] null; 
  if a.prototype then ???? end if; 
  final: BOOLEAN; 
  case a.memberMod of 
    {none} do final [] false; 
    {static} do if env[0] [] CLASS then throw definitionError end if; final [] false; 
    {final} do final [] true; 
    {constructor, virtual} do throw definitionError
  end case; 
  privateNamespace: NAMESPACE [] new NAMESPACE [name: “private”]; 
  dynamic: BOOLEAN [] a.dynamic or superclass.dynamic; 
  c: CLASS [] new CLASS [ staticReadBindings: {}, staticWriteBindings: {}, instanceReadBindings: {}, 
    instanceWriteBindings: {}, instanceInitOrder: []], complete: false, super: superclass, prototype: prototype, 
    privateNamespace: privateNamespace, dynamic: dynamic, allowNull: true, final: final, call: call, 
    construct: construct, defaultValue: null ];
  proc coerce (o: OBJECT): OBJECT 
    if relaxedHasType (o, c) then return o end if; 
    throw badValueError
  end proc; 
  c: implicitCoerce [] coerce; 
  Class [ClassDefinition [] c]; 
  v: VARIABLE [] new VARIABLE [type: classClass, value: c, immutable: true ];
  defineStaticMember (env, Name [Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v); 
  ValidateUsingFrame [Block] (ctx, env, JUMPTARGETS [breakTargets: {}], continueTargets: {} [], pl, c); 
  c: complete [] true
  end proc;

proc Validate [Inheritance] (ctx: CONTEXT, env: ENVIRONMENT): CLASS 
  [Inheritance [] «empty»] do return objectClass; 
  [Inheritance [] extends TypeExpression [allowIn]] do 
    Validate [TypeExpression [allowIn]] (ctx, env); 
    return SetupAndEval [TypeExpression [allowIn]] (env)
  end proc;

Setup 

proc Setup [ClassDefinition [] class Identifier Inheritance Block] () 
  Setup [Block] ()
end proc;
Evaluation

proc Eval[ClassDefinition [] class Identifier Inheritance Block] (env: ENVIRONMENT, d: OBJECT): OBJECT
  c: CLASS [] Class[ClassDefinition];
  return EvalUsingFrame[Block](env, c, d)
end proc;

15.6 Namespace Definition

Syntax

NamespaceDefinition [] namespace Identifier

Validation

proc Validate[NamespaceDefinition [] namespace Identifier]
  (ext: CONTEXT, env: ENVIRONMENT, pl: PLURALITY, attr: ATTRIBUTEOptNotFALSE)
  if pl ≠ singular then throw syntaxError end if;
  a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
  if a.dynamic or a.prototype then throw definitionError end if;
  if not (a.memberMod = none or (a.memberMod = static and env[0] [] CLASS)) then
    throw definitionError
  end if;
  name: STRING [] Name[Identifier];
  ns: NAMESPACE [] new NAMESPACE[]name: name[]
  v: VARIABLE [] new VARIABLE[]type: namespaceClass, value: ns, immutable: true[]
  defineStaticMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
end proc;

15.7 Package Definition

Syntax

PackageDefinition []
  package Block
  | package PackageName Block

PackageName []
  Identifier
  | PackageName . Identifier

16 Programs

Syntax

Program [] Directives
Evaluation

EvalProgram[Program [] Directives]: Object
begin
  Validate[Directives](initialContext, initialEnvironment, JUMPTARGETS)breakTargets: {}, continueTargets: {}[]
  singular, none);
  Setup[Directives]();
  return Eval[Directives](initialEnvironment, undefined)
end;

17 Predefined Identifiers

18 Built-in Classes

proc makeBuiltInClass(superclass: CLASSOPT, dynamic: BOOLEAN, allowNull: BOOLEAN, final: BOOLEAN, defaultValue: OBJECT): CLASS
proc call(this: OBJECT, args: ARGUMENTLIST, phase: PHASE): OBJECT
????
end proc;
proc construct(args: ARGUMENTLIST, phase: PHASE): OBJECT
????
end proc;
proc coerce(o: OBJECT): OBJECT
????
end proc;
privateNamespace: NAMESPACE [] new NAMESPACE[] name: “private”[]
return new CLASS[] staticReadBindings: {}, staticWriteBindings: {}, instanceReadBindings: {},
  instanceWriteBindings: {}, instanceInitOrder: [], complete: true, super: superclass, prototype: null,
  privateNamespace: privateNamespace, dynamic: dynamic, allowNull: allowNull, final: final, call: call,
  construct: construct, implicitCoerce: coerce, defaultValue: defaultValue[]
end proc;

objectClass: CLASS = makeBuiltInClass(None, false, true, false, undefined);
undefinedClass: CLASS = makeBuiltInClass(objectClass, false, false, true, undefined);
nullClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
booleanClass: CLASS = makeBuiltInClass(objectClass, false, false, true, false);
generalNumberClass: CLASS = makeBuiltInClass(objectClass, false, false, false, NaN64);
longClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, LONG[value: 0][]);
uLongClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, ULONG[value: 0][]);
floatClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, NaN32);
numberClass: CLASS = makeBuiltInClass(generalNumberClass, false, false, true, NaN64);
characterClass: CLASS = makeBuiltInClass(objectClass, false, false, true, ‘nUL’);
stringClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
namespaceClass: CLASS = makeBuiltInClass(objectClass, false, true, true, null);
attributeClass: `CLASS = makeBuiltInClass(objectClass, false, true, true, null);`

classClass: `CLASS = makeBuiltInClass(objectClass, false, true, true, null);`

functionClass: `CLASS = makeBuiltInClass(objectClass, false, true, true, null);`

prototypeClass: `CLASS = makeBuiltInClass(objectClass, true, true, true, null);`

packageClass: `CLASS = makeBuiltInClass(objectClass, true, true, true, null);`

`objectPrototype: PROTOTYPE = new PROTOTYPE {parent: none, dynamicProperties: {}}`
18.1 Object
18.2 Never
18.3 Void
18.4 Null
18.5 Boolean
18.6 Integer
18.7 Number
18.7.1 ToNumber Grammar
18.8 Character
18.9 String
18.10 Function
18.11 Array
18.12 Type
18.13 Math
18.14 Date
18.15 RegExp
18.15.1 Regular Expression Grammar
18.16 Error
18.17 Attribute

19 Built-in Functions

20 Built-in Attributes
21 Built-in Namespaces

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

A Index

A.1 Nonterminals

AdditiveExpression 97
AllParameters 145
AnnotatableDirective 128
Arguments 90
ArrayLiteral 83
ASCIIDigit 32
AssignmentExpression 109
Attribute 131
AttributeCombination 131
AttributeExpression 85
Attributes 131
BitwiseAndExpression 104
BitwiseOrExpression 104
BitwiseXorExpression 104
Block 117
BlockCommentCharacters 28
Brackets 90
BreakStatement 123
CaseLabel 120
CaseStatement 120
CaseStatements 119
CaseStatementsPrefix 119
CatchClause 125
CatchClauses 125
CatchClausesOpt 125
ClassDefinition 146
CompoundAssignment 109
ConditionalExpression 108
ContinueStatement 123
ContinuingIdentifierCharacter 30
ContinuingIdentifierCharacterOrEscape 30
ControlEscape 34
DecimalDigits 32
DecimalIntegerLiteral 32
DecimalLiteral 32
Directive 127
Directives 128
DirectivesPrefix 128
DivisionPunctuator 31
DoStatement 120
ElementList 83
EmptyStatement 116
EndOfInput 27
EqualityExpression 102
ExportBinding 135
ExportBindingList 135
ExportDefinition 135
ExpressionQualifiedIdentifier 77
ExpressionStatement 116
FieldList 81
FieldName 81
ForInBinding 122
ForInitialiser 122
ForStatement 122
Fraction 32
FullNewExpression 85
FullNewSubexpression 85
FullPostfixExpression 85
FunctionCommon 142
FunctionDefinition 142
FunctionExpression 81
FunctionName 142
HexDigit 32
HexEscape 34
HexIntegerLiteral 32
Identifier 77
IdentifierName 29
IdentityOrKeyword 29
IdentityEscape 34
IfStatement 118
ImportBinding 133
ImportDirective 133
ImportSource 133
IncludesExcludes 133
Inheritance 146
InitialIdentifierCharacter 30
InitialIdentifierCharacterOrEscape 30
InputElement 27
IntegerLiteral 31
LabeledStatement 118
LetterE 32
LetterF 31
LetterL 31
LetterU 31
LetterX 32
LineBreak 28
LineBreaks 28
LineComment 28
LineCommentCharacters 28
LineTerminator 28
ListExpression 111
LiteralElement 83
LiteralField 81
LiteralStringChar 34
LogicalAndExpression 106
LogicalAssignment 109
LogicalOrExpression 106
LogicalXorExpression 106
Mantissa 32
MemberOperator 90
MultiLineBlockComment 29
A.2 Tags

- 10, 12  
++ 10, 12  
+zero 10, 12  
abstract 38, 48  
andEq 109  
compile 45  
constructor 38  
default 45  
equal 51  
false 4, 37  
final 38  
fixed 41  
forbidden 47  
generic 68  
get 44  
greater 51  
inaccessible 37, 45  
less 51  
NaN 10, 12  
none 37, 38, 39, 40  
normal 44  
null 37  
orEq 109  
plural 45  
potentialConflict 61  
propertyLookup 65  
read 58  
readWrite 58  
run 45  
set 44  
singular 45  
static 38  
true 4, 37  
undefined 37  
uninitialised 37  
unordered 37  
virtual 51  
write 58  
xorEq 109  
–zero 10, 12

A.3 Semantic Domains

ACCESS 58  
ALIASINSTANCE 41  
ARGUMENTLIST 44  
ATTRIBUTE 39  
ATTRIBUTEOPTNOTFALSE 39  
ATTRIBUTEBLOCKFRAME 46  
BOOLEAN 4, 37  
BOOLEANOPT 37  
BRACKETREFERENCE 43  
CALLABLEINSTANCE 41  
CHARACTER 7  
CLASS 39
**A.4 Globals**

<table>
<thead>
<tr>
<th>Function</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>98</td>
</tr>
<tr>
<td><code>addStaticBindings</code></td>
<td>59</td>
</tr>
<tr>
<td><code>assignArguments</code></td>
<td>145</td>
</tr>
<tr>
<td><code>attributeClass</code></td>
<td>150</td>
</tr>
<tr>
<td><code>badConstruct</code></td>
<td>76</td>
</tr>
<tr>
<td><code>bitAnd</code></td>
<td>105</td>
</tr>
<tr>
<td><code>bitNot</code></td>
<td>94</td>
</tr>
<tr>
<td><code>bitOr</code></td>
<td>106</td>
</tr>
<tr>
<td><code>bitwiseAnd</code></td>
<td>7</td>
</tr>
<tr>
<td><code>bitwiseOr</code></td>
<td>7</td>
</tr>
<tr>
<td><code>bitwiseShift</code></td>
<td>7</td>
</tr>
<tr>
<td><code>bitwiseXor</code></td>
<td>7</td>
</tr>
<tr>
<td><code>bitXor</code></td>
<td>106</td>
</tr>
<tr>
<td><code>booleanClass</code></td>
<td>149</td>
</tr>
<tr>
<td><code>bracketDelete</code></td>
<td>57</td>
</tr>
<tr>
<td><code>bracketRead</code></td>
<td>57</td>
</tr>
<tr>
<td><code>bracketWrite</code></td>
<td>57</td>
</tr>
<tr>
<td><code>call</code></td>
<td>89</td>
</tr>
<tr>
<td><code>characterClass</code></td>
<td>149</td>
</tr>
<tr>
<td><code>characterToCode</code></td>
<td>7</td>
</tr>
<tr>
<td><code>checkInteger</code></td>
<td>50</td>
</tr>
<tr>
<td><code>classClass</code></td>
<td>150</td>
</tr>
<tr>
<td><code>codeToCharacter</code></td>
<td>7</td>
</tr>
<tr>
<td><code>combineAttributes</code></td>
<td>56</td>
</tr>
<tr>
<td><code>integerOpt</code></td>
<td>37</td>
</tr>
<tr>
<td><code>jumpTargets</code></td>
<td>45</td>
</tr>
<tr>
<td><code>label</code></td>
<td>45</td>
</tr>
<tr>
<td><code>lexicalLookup</code></td>
<td>66</td>
</tr>
<tr>
<td><code>lexicalReference</code></td>
<td>43</td>
</tr>
<tr>
<td><code>limitedInstance</code></td>
<td>42</td>
</tr>
<tr>
<td><code>long</code></td>
<td>10</td>
</tr>
<tr>
<td><code>lookupKind</code></td>
<td>66</td>
</tr>
<tr>
<td><code>memberInstantiation</code></td>
<td>64</td>
</tr>
<tr>
<td><code>memberModifier</code></td>
<td>38</td>
</tr>
<tr>
<td><code>methodClosure</code></td>
<td>40</td>
</tr>
<tr>
<td><code>multiname</code></td>
<td>38</td>
</tr>
<tr>
<td><code>namedArgument</code></td>
<td>44</td>
</tr>
<tr>
<td><code>namedParameter</code></td>
<td>44</td>
</tr>
<tr>
<td><code>namespace</code></td>
<td>38</td>
</tr>
<tr>
<td><code>null</code></td>
<td>37</td>
</tr>
<tr>
<td><code>object</code></td>
<td>37</td>
</tr>
<tr>
<td><code>objectIOpt</code></td>
<td>37</td>
</tr>
<tr>
<td><code>objectOpt</code></td>
<td>37</td>
</tr>
<tr>
<td><code>objectU</code></td>
<td>37</td>
</tr>
<tr>
<td><code>obUnaryOptionalLimit</code></td>
<td>43</td>
</tr>
<tr>
<td><code>obUnaryRef</code></td>
<td>43</td>
</tr>
<tr>
<td><code>openInstance</code></td>
<td>41</td>
</tr>
<tr>
<td><code>order</code></td>
<td>51</td>
</tr>
<tr>
<td><code>overriddenMember</code></td>
<td>61</td>
</tr>
<tr>
<td><code>overrideModifier</code></td>
<td>38</td>
</tr>
<tr>
<td><code>overrideStatus</code></td>
<td>61</td>
</tr>
<tr>
<td><code>overrideStatusPair</code></td>
<td>61</td>
</tr>
<tr>
<td><code>package</code></td>
<td>42</td>
</tr>
<tr>
<td><code>parameter</code></td>
<td>44</td>
</tr>
<tr>
<td><code>parameterFrame</code></td>
<td>46</td>
</tr>
<tr>
<td><code>phase</code></td>
<td>45</td>
</tr>
<tr>
<td><code>plurality</code></td>
<td>45</td>
</tr>
<tr>
<td><code>primitiveObject</code></td>
<td>37</td>
</tr>
<tr>
<td><code>prototype</code></td>
<td>40</td>
</tr>
<tr>
<td><code>prototypeOpt</code></td>
<td>40</td>
</tr>
<tr>
<td><code>qualifiedName</code></td>
<td>38</td>
</tr>
<tr>
<td><code>qualifiedNameOpt</code></td>
<td>38</td>
</tr>
<tr>
<td><code>rational</code></td>
<td>6</td>
</tr>
<tr>
<td><code>real</code></td>
<td>6</td>
</tr>
<tr>
<td><code>reference</code></td>
<td>43</td>
</tr>
<tr>
<td><code>setter</code></td>
<td>47</td>
</tr>
<tr>
<td><code>signature</code></td>
<td>44</td>
</tr>
<tr>
<td><code>simpleInstance</code></td>
<td>40</td>
</tr>
<tr>
<td><code>slot</code></td>
<td>42</td>
</tr>
<tr>
<td><code>staticInstance</code></td>
<td>40</td>
</tr>
<tr>
<td><code>staticMember</code></td>
<td>47</td>
</tr>
<tr>
<td><code>staticMemberOpt</code></td>
<td>47</td>
</tr>
<tr>
<td><code>string</code></td>
<td>8</td>
</tr>
<tr>
<td><code>stringOpt</code></td>
<td>38</td>
</tr>
<tr>
<td><code>systemFrame</code></td>
<td>46</td>
</tr>
<tr>
<td><code>token</code></td>
<td>26</td>
</tr>
<tr>
<td><code>undefined</code></td>
<td>37</td>
</tr>
<tr>
<td><code>variable</code></td>
<td>47</td>
</tr>
<tr>
<td><code>variableType</code></td>
<td>47</td>
</tr>
<tr>
<td><code>variableValue</code></td>
<td>47</td>
</tr>
</tbody>
</table>
isStrictEqual 104
lexicalDelete 65
lexicalRead 65
lexicalWrite 65
logicalNot 94
longClass 149
makeBuiltInClass 149
minus 94
multiply 96
namespaceClass 149
nullClass 149
numberClass 149
objectClass 149
objectPrototype 150
objectType 52
packageClass 150
plus 94
processPragma 135
prototypeClass 150
rationalToLong 50
rationalToULong 50
readDynamicProperty 70
readInstanceMember 69
readLimitedReference 84
readProperty 69
readReference 56
readStaticMember 70
readVariable 71
realToFloa32 11
realToFloa64 12
referenceBase 89
relaxedHasType 52
remainder 96
resolveAlias 51
resolveInstanceMemberName 68
resolveOverrides 62
searchForOverrides 62
selectPublicName 66
shiftLeft 99
shiftRight 100
shiftRightUnsigned 100
signedWrap32 49
signedWrap64 49
staticBindingsWithAccess 59
stringClass 149
subtract 98
toBoolean 52
toCompoundAttribute 56
toFloat 51
toGeneralNumber 53
toPrimitive 55
toRational 50
toString 53
truncateFiniteFloat32 11
truncateFiniteFloat64 13
truncateToInt32 49
uLongClass 149
undefinedClass 149
unsignedWrap32 49
unsignedWrap64 49
writeDynamicProperty 74
writeInstanceMember 73
writeProperty 72
writeReference 57
writeStaticMember 73
writeVariable 74