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
      5.10.1 Shorthand Notation ................................ 9
   5.11 Records .................................................... 9
   5.12 ECMAScript Numeric Types ............................ 10
      5.12.1 Signed Long Integers ................................ 10
         5.12.1.1 Shorthand Notation ............................ 10
      5.12.2 Unsigned Long Integers ............................ 10
         5.12.2.1 Shorthand Notation ............................ 10
      5.12.3 Single-Precision Floating-Point Numbers ........... 11
         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 ....................................... 14
   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
      7.4 Comments ............................................... 29
      7.5 Keywords and Identifiers ............................. 29
      7.6 Punctuators ............................................ 32
      7.7 Numeric literals ....................................... 32
      7.8 String literals ......................................... 35
      7.9 Regular expression literals .......................... 36
8 Program Structure ........................................... 37
   8.1 Packages ................................................ 37
   8.2 Scopes .................................................... 37
9 Data Model .................................................. 37
   9.1 Objects ................................................ 37
      9.1.1 Undefined ........................................... 38
      9.1.2 Null ............................................... 38
      9.1.3 Booleans ............................................ 38
      9.1.4 Numbers ............................................. 38
      9.1.5 Strings ............................................... 38
      9.1.6 Namespaces .......................................... 38
      9.1.6.1 Qualified Names .................................. 39
      9.1.7 Compound attributes ................................ 39
      9.1.8 Classes .............................................. 39
      9.1.9 Simple Instances .................................... 41
         9.1.9.1 Slots ........................................... 41
      9.1.10 Uninstantiated Functions ........................ 41
      9.1.11 Method Closures ................................... 42
      9.1.12 Dates ............................................... 42
      9.1.13 Regular Expressions ............................... 42
      9.1.14 Packages and Global Objects ...................... 43
   9.2 Objects with Limits .................................... 43
   9.3 References ............................................... 43
   9.4 Phases of evaluation .................................... 44
   9.5 Contexts .................................................. 44
   9.6 Labels .................................................... 44
   9.7 Semantic Exceptions ..................................... 45
   9.8 Function Support ........................................ 45
   9.9 Environment Frames ..................................... 46
      9.9.1 System Frame ......................................... 46
      9.9.2 Function Parameter Frames ........................ 46
         9.9.2.1 Parameters ....................................... 47
      9.9.3 Local Frames ......................................... 47
      9.9.4 With Frames .......................................... 47
   9.10 Environment Bindings .................................... 47
      9.10.1 Static Bindings ..................................... 47
      9.10.2 Instance Bindings .................................. 49
10 Data Operations ............................................. 50
   10.1 Numeric Utilities ....................................... 50
   10.2 Object Utilities ......................................... 52
      10.2.1 objectType ........................................... 52
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 +∞.

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,

\[
\{x \mid x \in \text{INTEGER such that } x^2 < 10\} = \{-3, -2, -1, 0, 1, 2, 3\}
\]

\[
\{x^2 \mid x \in \{-5, -1, 1, 2, 4\}\} = \{1, 4, 16, 25\}
\]

\[
x^{10} + y \mid x \in \{1, 2, 4\}, y \in \{3, 5\} = \{13, 15, 23, 25, 43, 45\}
\]

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:

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

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

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,

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

evaluates to \text{true} and leaves \(x\) set to either 16 or 26. Other examples include:
The unique sequence corresponding to an integer nonnegative integers and every
More
5.6.1 be cascaded, so
Real
Let
example of a real number slightly larger than 3.14.
R
1/3, 7.5, –12/7, and 2
R
domain
Numbers
5.6
For example,
the
quantifier
every \(x \in A\) satisfies \(\text{predicate}(x)\)
returns \(\text{true}\) if there exists no element \(x\) in set \(A\) such that \(\text{predicate}(x)\) computes to \(\text{false}\). If there is at least one such element \(x\), then the \(\text{every}\) quantifier’s result is \(\text{false}\). As a degenerate case, the \(\text{every}\) quantifier is always \(\text{true}\) if the set \(A\) is empty. For example,
\[
(\text{every } x \in \{3, 16, 19, 26\} \text{ satisfies } x \mod 10 = 6) = \text{false};
(\text{every } x \in \{6, 26, 96, 106\} \text{ satisfies } x \mod 10 = 6) = \text{true};
(\text{every } x \in \{\} \text{ satisfies } x \mod 10 = 6) = \text{true}.
\]

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, \(10^{100}\), and \(\square\). Hexadecimal numbers are written by preceding them with “0x”, so 4294967296, 0x10000000, and \(2^{32}\) are all the same integer.

\(\text{INTEGER}\) is the semantic domain of all integers \(\{-3, -2, -1, 0, 1, 2, 3\ldots\}\). 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: \(\text{INTEGER} \square \text{RATIONAL}\). 3, 1/3, 7.5, -12/7, and \(2^{-3}\) are examples of rational numbers.

\(\text{REAL}\) is the semantic domain of all real numbers. Every rational number is also a real number: \(\text{RATIONAL} \square \text{REAL}\). \(\square\) 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:

\[-x\]Negation
\[x + y\]Sum
\[x - y\]Difference
\[x \cdot y\]Product
\[x / y\]Quotient (\(y\) must not be zero)
\[x^y\]x raised to the \(y\)th power (used only when either \(x\neq0\) and \(y\) is an integer or \(x\) is any number and \(y>0\))
\[\lvert x \rvert\]The absolute value of \(x\), which is \(x\) if \(x\geq0\) and \(-x\) otherwise
\[\lfloor x \rfloor\]Floor of \(x\), which is the unique integer \(i\) such that \(i \leq x < i+1\). \(\lfloor 3 \rfloor = 3\), \(\lfloor -3.5 \rfloor = -4\), and \(\lceil -7 \rceil = 7\).
\[\lceil x \rceil\]Ceiling of \(x\), which is the unique integer \(i\) such that \(i-1 < x \leq i\). \(\lceil 3.5 \rceil = 4\), \(\lceil -3.5 \rceil = -3\), and \(\lfloor 7 \rfloor = 7\).
\[x \mod y\]x modulo \(y\), which is defined as \(x - y\lfloor x/y \rfloor\) \(y\) must not be zero. \(10 \mod 7 = 3\), and \(-1 \mod 7 = 6\).

Real numbers can be compared using =, ≠, <, ≤, >, and ≥. The result is either \(\text{true}\) or \(\text{false}\). Multiple relational operators can be cascaded, so \(x < y < z\) is \(\text{true}\) 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 \(\text{true}\) and 0 bit representing \(\text{false}\).

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 \(a_i \{0, 1\}\). 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
\[ a_i = \lfloor k / 2 \rfloor \mod 2 \]

If \( x \) is zero or positive, then its sequence will have infinitely many consecutive leading 0’s, while a negative integer \( x \) will generate a sequence with infinitely many consecutive leading 1’s. For example, \( 6 \) generates the sequence \( \ldots 0 \ldots 0000110 \), while \( -6 \) generates \( \ldots 1 \ldots 1101110 \).

The logical AND, OR, and XOR operations below operate on corresponding elements of the sequences \( a_i \) and \( b_i \) generated by the two parameters \( x \) and \( y \). 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, \( \text{AND} \)ing corresponding elements of the sequences generated by \( 6 \) and \( -6 \) yields the sequence \( \ldots 0 \ldots 0000010 \), which is the sequence generated by the integer 2. Thus, \( \text{bitwiseAnd}(6, -6) = 2 \).

\[
\begin{align*}
\text{bitwiseAnd}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER} & \quad \text{Return the bitwise AND of } x \text{ and } y \\
\text{bitwiseOr}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER} & \quad \text{Return the bitwise OR of } x \text{ and } y \\
\text{bitwiseXor}(x: \text{INTEGER}, y: \text{INTEGER}): \text{INTEGER} & \quad \text{Return the bitwise XOR of } x \text{ and } y \\
\text{bitwiseShift}(x: \text{INTEGER}, count: \text{INTEGER}): \text{INTEGER} & \quad \text{Return } x \text{ shifted to the left by } \text{count} \text{ bits. If } \text{count} \text{ is negative, return } x \text{ shifted to the right by } -\text{count} \text{ bits. Bits shifted out of the right end are lost; bit shifted in at the right end are zero. } \text{bitwiseShift}(x, \text{count}) \text{ is exactly equivalent to } \lfloor x \cdot 2^{\text{count}} \rfloor
\end{align*}
\]

### 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.

\text{CHARACTER} \text{ is the semantic domain of all 65536 characters ‘«u0000» ... ‘«uFFFF»’}.

Characters can be compared using equal, not equal, less than, greater than, and greater than or equal. These operators compare code point values, so ‘A’ = ‘A’, ‘A’ < ‘B’, and ‘A’ < ‘a’ are all \text{true}.

The procedures \text{characterToCode} and \text{codeToCharacter} convert between characters and their integer Unicode values.

\[
\begin{align*}
\text{characterToCode}(c: \text{CHARACTER}) & : \{0 \ldots 65535 \} \quad \text{Return character } c \text{’s Unicode code point as an integer} \\
\text{codeToCharacter}(i: \{0 \ldots 65535\}): \text{CHARACTER} & \quad \text{Return the character whose Unicode code point is } i
\end{align*}
\]

### 5.8 Lists

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

\[
[\text{element}_0, \text{element}_1, \ldots, \text{element}_n]
\]

For example, the following list contains four strings:

\[
[\text{“parsley”}, \text{“sage”}, \text{“rosemary”}, \text{“thyme”}]
\]

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

\[
[\{f(x) | \lfloor x \cdot u \rfloor \} \text{ such that } \text{predicate}(x)]
\]

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,
\[ \begin{align*} \{ x^2 \mid \square \} & \subseteq \{ -1, 1, 2, 3, 4, 2, 5 \} = \{ 1, 1, 4, 9, 16, 4, 25 \} \\ \{ x + 1 \mid \square \} & \subseteq \{ -1, 1, 2, 3, 4, 5, 3, 10 \} \text{ such that } x \mod 2 = 1 = \{ 0, 2, 4, 6, 4 \} \end{align*} \]

Let \( u = [e_0, e_1, \ldots, e_{n-1}] \) and \( v = [f_0, f_1, \ldots, f_{m-1}] \) be lists, \( e \) be an element, \( i \) and \( j \) be integers, and \( x \) be a value. The operations below can be done on lists. The operations are meaningful only when their preconditions are met; the semantics never use the operations below without meeting their preconditions.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Precondition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>u</td>
<td>)</td>
</tr>
<tr>
<td>( u[i] )</td>
<td>( 0 \leq i &lt;</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \ldots j] )</td>
<td>( 0 \leq i \leq j \leq</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \ldots] )</td>
<td>( 0 \leq i \leq</td>
<td>u</td>
</tr>
<tr>
<td>( u[i \setminus x] )</td>
<td>( 0 \leq i &lt;</td>
<td>u</td>
</tr>
<tr>
<td>( u \oplus v )</td>
<td></td>
<td>The concatenated list ([e_0, e_1, \ldots, e_{n-1}, f_0, f_1, \ldots, f_{m-1}])</td>
</tr>
<tr>
<td>( \text{repeat}(e, i) )</td>
<td>( i \geq 0 )</td>
<td>The list ([e, e, \ldots, e]) of length ( i ) containing ( i ) identical elements ( e )</td>
</tr>
<tr>
<td>( u = v )</td>
<td></td>
<td>\textbf{true} if the lists ( u ) and ( v ) are equal and \textbf{false} otherwise. Lists ( u ) and ( v ) are equal if they have the same length and all of their corresponding elements are equal.</td>
</tr>
<tr>
<td>( u \neq v )</td>
<td></td>
<td>\textbf{false} if the lists ( u ) and ( v ) are equal and \textbf{true} otherwise.</td>
</tr>
</tbody>
</table>

If \( T \) is a semantic domain, then \( T[] \) is the semantic domain of all lists whose elements are members of \( T \). The empty list \([\] \) is a member of \( T[] \) for any semantic domain \( T \).

In addition to the above, the \textbf{some} and \textbf{every} quantifiers can be used on lists just as on sets:

\textbf{some} \( x \subseteq u \) \textbf{satisfies} \textit{predicate}(x) \\
\textbf{every} \( x \subseteq u \) \textbf{satisfies} \textit{predicate}(x)

These quantifiers’ behaviour on lists is analogous to that on sets, except that, if the \textbf{some} quantifier returns \textbf{true} then it leaves variable \( x \) set to the \textit{first} element of list \( u \) that satisfies condition \textit{predicate}(x). For example,

\textbf{some} \( x \subseteq [3, 36, 19, 26] \textbf{satisfies} x \mod 10 = 6 \)

evaluates to \textbf{true} and leaves \( x \) set to 36.

### 5.9 Strings

A list of characters is called a \textit{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\LF\”

is equivalent to:

\[ [‘W’, ‘o’, ‘n’, ‘,’ ‘e’, ‘r’, ‘\LF’] \]

The empty string is usually written as \“\“.

In addition to the other list operations, \(<\), \(\leq\), \(>\), and \(\geq\) 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 \( y \).

\textbf{STRING} is the semantic domain of all strings, \textbf{STRING} = \textbf{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>

label₁ through labelₙ are the names of the fields. T₁ through Tₙ are informative semantic domains of possible values that the corresponding fields may hold.

The notation

\[ \text{NAME} \cdot \text{label}_1: v_1, \ldots, \text{label}_n: v_n \]

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

\[ \text{NAME} \cdot \text{label}_i: v_i, \ldots, \text{label}_d: v_d, \text{other fields from } a \]

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

If a is the tuple \[ \text{NAME} \cdot \text{label}_1: v_1, \ldots, \text{label}_n: v_n \] then

\[ a.\text{label}_i \]

returns the \( i^{th} \) 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.

5.10.1 Shorthand Notation

The semantic notation ns::id is a shorthand for QUALIFIEDNAME[namespace: ns, id: id] See section 9.1.6.1.

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>

label₁ through labelₙ are the names of the fields. T₁ through Tₙ are informative semantic domains of possible values that the corresponding fields may hold.

The expression

\[ \text{new NAME} \cdot \text{label}_1: v_1, \ldots, \text{label}_n: v_n \]
creates a record with name \texttt{NAME} and a new address \texttt{[]}. The fields labelled \texttt{label}_1 through \texttt{label}_k at address \texttt{[]} are initialised with values \texttt{v}_1 through \texttt{v}_k respectively. Each value \texttt{v}_i is a member of the corresponding semantic domain \( T_i \). A \texttt{label}_i: \texttt{v}_i pair may be omitted from a \texttt{new} expression, which indicates that the initial value of field \texttt{label}_i 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 \texttt{a}, the \texttt{new} expression can be abbreviated as

\texttt{new \texttt{NAME}}\texttt{[label}_1\texttt{: v}_1, \ldots, \texttt{label}_k\texttt{: v}_k\texttt{, \ldots]}\texttt{, other fields from \texttt{a}[]}

which represents a record \texttt{b} with name \texttt{NAME} and a new address \texttt{[]}. The fields labelled \texttt{label}_j through \texttt{label}_k at address \texttt{[]} are initialised with values \texttt{v}_j through \texttt{v}_k respectively; the other fields at address \texttt{[]} are initialised with the values of correspondingly labeled fields from \texttt{a}’s address.

If \texttt{a} is a record with name \texttt{NAME} and address \texttt{[]} then

\texttt{a.label}_i

returns the current value \texttt{v} of the \texttt{i}th field at address \texttt{[]}. That field may be set to a new value \texttt{w}, which must be a member of the semantic domain \( T_i \), using the assignment

\texttt{a.label}_i \texttt{[]= w}

after which \texttt{a.label}_i will evaluate to \texttt{w}. Any record with a different address \texttt{[]} is unaffected by the assignment.

The equality operators = and \# 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}.

\section{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}:

\begin{verbatim}
GENERALNUMBER = LONG [] ULONG [] FLOAT32 [] FLOAT64
\end{verbatim}

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}:

\begin{verbatim}
FINITEGENERALNUMBER = LONG [] ULONG [] FINITEFLOAT32 [] FINITEFLOAT64
\end{verbatim}

\subsection{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:

\begin{itemize}
  \item \texttt{value} \{-2^{63} \ldots 2^{63} - 1\} \quad \text{The signed 64-bit integer}
\end{itemize}

\subsection{Shorthand Notation}

In this specification, when \( i \) is an integer between \(-2^{63} \) and \( 2^{63} - 1 \), the notation \( i_{\text{long}} \) indicates the result of \texttt{LONG[\texttt{value}: \texttt{[]}]} which is the integer \( i \) wrapped in a \texttt{LONG} tuple.

\subsection{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:
Field | Contents | Note
---|---|---
value | \{0 \ldots 2^{64} – 1\} | The unsigned 64-bit integer

5.12.2.1 Shorthand Notation

In this specification, when \(i\) is an integer between 0 and \(2^{64} – 1\), the notation \(i_{\text{ulong}}\) indicates the result of \(\text{ULONG}[\text{value}]\) which is the integer \(i\) wrapped in a \(\text{ULONG}\) tuple.

5.12.3 Single-Precision Floating-Point Numbers

\(\text{FLOAT32}\) 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. \(\text{FLOAT32}\) is the union of the following semantic domains:

\[
\begin{align*}
\text{FLOAT32} &= \text{FINITEFLOAT32} \sqcup \{+\infty_{\text{f32}}, -\infty_{\text{f32}}, \text{NaN}_{\text{f32}}\}; \\
\text{FINITEFLOAT32} &= \text{NONZEROFINITEFLOAT32} \sqcup \{+\text{zero}_{\text{f32}}, -\text{zero}_{\text{f32}}\}
\end{align*}
\]

The non-zero finite values are wrapped in a tuple (see section 5.10) to keep them disjoint from members of the semantic domains \(\text{LONG, ULONG,}\) and \(\text{FLOAT64}.\) A \(\text{NONZEROFINITEFLOAT32}\) tuple has the field below:

Field | Contents | Note
---|---|---
value | \(\text{NORMALISEDFLOAT32VALUES} \sqcup \text{DENORMALISEDFLOAT32VALUES}\) | The value, represented as an exact rational number

There are 4261412864 (that is, \(2^{32} – 2^{25}\)) \text{normalised} values:

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

\(m\) is called the significand.

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

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

\(m\) is called the significand.

The remaining \(\text{FLOAT32}\) 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 \(\text{NaN}_{\text{f32}}\) (not a number).

Members of the semantic domain \(\text{NONZEROFINITEFLOAT32}\) with value greater than zero are called \text{positive finite}. The remaining members of \(\text{NONZEROFINITEFLOAT32}\) are called \text{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 \text{false} for different tags, so \(+\text{zero}_{\text{f32}} \neq -\text{zero}_{\text{f32}}\) but \(\text{NaN}_{\text{f32}} = \text{NaN}_{\text{f32}}\). The ECMAScript \(x == y\) and \(x === y\) operators have different behavior for \(\text{FLOAT32}\) values, defined by \text{isEqual} and \text{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 \(\text{realToFloat32}(x)\), which is the “closest” \(\text{FLOAT32}\) value as defined below. Thus, 3.4 is a \text{REAL} number, while 3.4_{\text{f32}} is a \(\text{FLOAT32}\) value (whose exact value is actually 3.400000095367431640625). The positive finite \(\text{FLOAT32}\) values range from \(10^{-45}_{\text{f32}}\) to \((3.4028235 \cdot 10^{38})_{\text{f32}}\).

5.12.3.2 Conversion

The procedure \(\text{realToFloat32}\) converts a real number \(x\) into the applicable element of \(\text{FLOAT32}\) as follows:
There are 18428729675200069632 (that is, 2<sup>64</sup>) values considered indistinguishable from each other. If two elements of \( s \) are equally close, let \( a \) be the one with an even significand; for this purpose \(-2^{128}, 0, 2^{128}\) are considered to have even significands.

- If \( a = 2^{128} \) then return \(+\infty_{32}\)
- Elif \( a = -2^{128} \) then return \(-\infty_{32}\)
- Elif \( a \neq 0 \) then return \( \text{NonzeroFiniteFloat32}\)\( \text{Value} : a \)
- Elif \( x < 0 \) then return \(-0_{32}\)
- Else return \(+0_{32}\)

end proc

**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:

\[
\text{proc truncateFiniteFloat32}(x: \text{FiniteFloat32}): \text{Integer}
\]

- If \( x \in \{+0_{32}, -0_{32}\} \) then return 0
- Else return \( r: \text{Rational} \)\( x.\text{Value}: r \)
- If \( r > 0 \) then return \( \lfloor r \rfloor \) else return \( \lceil r \rceil \)

end proc

### 5.12.3.3 Arithmetic

The following table defines negation of `Float32` values using IEEE 754 rules. Note that \( (\text{expr})_{32} \) is a shorthand for `realToFloat32(expr)`.

\[
\begin{array}{|c|c|}
\hline
x & \text{Result} \\
\hline
-\infty_{32} & +\infty_{32} \\
\text{negative finite} & (-x.\text{Value})_{32} \\
-0_{32} & +0_{32} \\
+0_{32} & -0_{32} \\
\text{positive finite} & (-x.\text{Value})_{32} \\
+\infty_{32} & -\infty_{32} \\
\text{NaN}_{32} & \text{NaN}_{32} \\
\hline
\end{array}
\]

### 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:

\[
\begin{align*}
\text{Float64} & = \text{FiniteFloat64} \cup \{-\infty_{64}, +\infty_{64}, \text{NaN}_{64}\}; \\
\text{FiniteFloat64} & = \text{NonzeroFiniteFloat64} \cup \{+0_{64}, -0_{64}\}
\end{align*}
\]

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>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>value</code></td>
<td><code>NormalisedFloat64Values</code></td>
<td>The value, represented as an exact rational number</td>
</tr>
</tbody>
</table>

There are 18428729675200069632 (that is, \( 2^{64} - 2^{54} \)) normalised values:

\[
\text{NormalisedFloat64Values} = \{s \in \text{Tuple}[2] | s \in \{-1, 1\}, \underline{m} \in \{2^{52} ... 2^{51-1}\}, \underline{e} \in \{-1074 ... 971\}\}
\]

\( m \) is called the significand.
There are also 9007199254740990 (that is, $2^{53} - 2$) denormalised non-zero values:

$\text{DENORMALISEDFLOAT64VALUES} = \{s\in \{0,1\} | m \text{ } \in \{-2^{1024}, -2^{1023}, \ldots, -1, 1\}, \text{ } m \text{ } \in \{1 \ldots 2^{52} - 1\}\}$

$m$ 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 $\text{NaN}\_64$ (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 FLOAT64 values, defined by isEqual and 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 realToFloat64($x$), which is the “closest” FLOAT64 value as defined below. Thus, 3.4 is a REAL number, while 3.4\_64 is a FLOAT64 value (whose exact value is actually $3.399999999999999911182158029987476766109466552734375$). The positive finite FLOAT64 values range from $(5 \cdot 10^{-324})\_64$ to $(1.7976931348623157 \cdot 10^{308})\_64$.

5.12.4.2 Conversion

The procedure realToFloat64 converts a real number $x$ into the applicable element of FLOAT64 as follows:

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

$$
s: RATIONAL \notin \text{NORMALISEDFLOAT64VALUES} \notin \text{DENORMALISEDFLOAT64VALUES} \notin \{-2^{1024}, 0, 2^{1024}\}.
$$

Let $a: RATIONAL$ 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, and $2^{1024}$ are considered to have even significands.

$$
\text{if } a = 2^{1024} \text{ then return } +\infty\_64
$$

$$
\text{elsif } a = -2^{1024} \text{ then return } -\infty\_64
$$

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

$$
\text{elsif } x < 0 \text{ then return } -\text{zero}\_64
$$

$$
\text{else return } +\text{zero}\_64
$$

end proc

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

The procedure float32ToFloat64 converts a FLOAT32 number $x$ into the corresponding FLOAT64 number as defined by the following table:

$$
\text{float32ToFloat64(x: FLOAT32): FLOAT64}
$$

<table>
<thead>
<tr>
<th>$x$</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty_32$</td>
<td>$-\infty_64$</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>$+\infty_32$</td>
<td>$+\infty_64$</td>
</tr>
<tr>
<td>$\text{NaN}_32$</td>
<td>$\text{NaN}_64$</td>
</tr>
</tbody>
</table>

Any NONZEROFINITEFLOAT32 value

$$
\text{NONZEROFINITEFLOAT64} \text{value: } x.\text{value}
$$

The procedure truncateFiniteFloat64 truncates a FINITEFLOAT64 value to an integer, rounding towards zero:

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

$$
\text{if } x \notin \{+\text{zero}_64, -\text{zero}_64\} \text{ then return 0 end if;}
$$

$$
\text{r: RATIONAL } \cdot x.\text{value;}
$$

$$
\text{if } r > 0 \text{ then return } \lfloor r \rfloor \text{ else return } \lceil r \rceil \text{ end if}
$$

end proc
5.12.4.3 Arithmetic

The following tables define procedures that perform common arithmetic on \texttt{FLOAT64} values using IEEE 754 rules. Note that \texttt{(expr)64} is a shorthand for \texttt{realToFloat64(expr)}.

\texttt{float64Abs(x: FLOAT64): FLOAT64}

<table>
<thead>
<tr>
<th>(x)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)64</td>
<td>(+\infty)64</td>
</tr>
<tr>
<td>negative finite</td>
<td>((-x)).value)64</td>
</tr>
<tr>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>positive finite</td>
<td>(x)</td>
</tr>
<tr>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
</tr>
<tr>
<td>NaN(64)</td>
<td>NaN(64)</td>
</tr>
</tbody>
</table>

\texttt{float64Negate(x: FLOAT64): FLOAT64}

<table>
<thead>
<tr>
<th>(x)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)64</td>
<td>(+\infty)64</td>
</tr>
<tr>
<td>negative finite</td>
<td>((-x)).value)64</td>
</tr>
<tr>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>positive finite</td>
<td>((-x)).value)64</td>
</tr>
<tr>
<td>(+\infty)64</td>
<td>(-\infty)64</td>
</tr>
<tr>
<td>NaN(64)</td>
<td>NaN(64)</td>
</tr>
</tbody>
</table>

\texttt{float64Add(x: FLOAT64, y: FLOAT64): FLOAT64}

<table>
<thead>
<tr>
<th>(x)</th>
<th>(-\infty)64 \text{negative finite}</th>
<th>(-\text{zero})64</th>
<th>(+\text{zero})64 \text{positive finite}</th>
<th>(+\infty)64</th>
<th>NaN(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)64</td>
<td>(-\infty)64</td>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64 \text{positive finite}</td>
<td>(+\infty)64</td>
<td>NaN(64)</td>
</tr>
<tr>
<td>negative finite &amp; (-\infty)64</td>
<td>((x)).value + ((y)).value)64</td>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64 \text{positive finite}</td>
<td>(+\infty)64</td>
<td>NaN(64)</td>
</tr>
<tr>
<td>(-\text{zero})64</td>
<td>(-\text{zero})64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
</tr>
<tr>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>positive finite &amp; (+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
</tr>
<tr>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
</tr>
</tbody>
</table>

\texttt{float64Subtract(x: FLOAT64, y: FLOAT64): FLOAT64}

<table>
<thead>
<tr>
<th>(x)</th>
<th>(-\infty)64 \text{negative finite}</th>
<th>(-\text{zero})64</th>
<th>(+\text{zero})64 \text{positive finite}</th>
<th>(+\infty)64</th>
<th>NaN(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)64</td>
<td>(-\infty)64</td>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64 \text{positive finite}</td>
<td>(+\infty)64</td>
<td>NaN(64)</td>
</tr>
<tr>
<td>negative finite &amp; (-\infty)64</td>
<td>((x)).value - ((y)).value)64</td>
<td>(-\text{zero})64</td>
<td>(+\text{zero})64 \text{positive finite}</td>
<td>(+\infty)64</td>
<td>NaN(64)</td>
</tr>
<tr>
<td>(-\text{zero})64</td>
<td>(-\text{zero})64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
<td>((-\text{zero}))64</td>
</tr>
<tr>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>positive finite &amp; (+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
<td>(+\text{zero})64</td>
</tr>
<tr>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
<td>(+\infty)64</td>
</tr>
<tr>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
<td>NaN(64)</td>
</tr>
</tbody>
</table>

\textbf{NOTE}  The identity for floating-point addition is \(-\text{zero}\)64; not \(+\text{zero}\)64.
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.

A procedure is denoted as:

```
proc f(param1: T1, ... , paramn: Tn): T
  step1;
  step2;
  ...;
  stepn
end proc;
```
If the procedure does not return a value, the : T on the first line is omitted.

\( f \) is the procedure’s name, \( \text{param}_i \) through \( \text{param}_n \) 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 \( \text{step}_1 \) through \( \text{step}_m \) 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 \( \text{param}_i \) through \( \text{param}_n \); each reference to a parameter \( \text{param} \), 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 \( f(\text{arg}_1, ..., \text{arg}_n) \) 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 =, ≠, 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 \[] T_2 \[] ... \[] T_n \[] T \). If \( n = 0 \), this semantic domain is written as \( () \[] T \). If the procedure does not produce a result, the semantic domain of procedures is written either as \( T_1 \[] T_2 \[] ... \[] T_n \[] () \) or as \( () \[] () \).

### 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 return or propagates an exception.

- **nothing**
  A nothing step performs no operation.

- **note Comment**
  A note step performs no operation. It provides an informative comment about the algorithm. If Comment is an expression, then the note step is an informative comment that asserts that the expression, if evaluated at this point, would be guaranteed to evaluate to true.

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

\[ v: T \[] \text{expression} \]
\[ v \[] \text{expression} \]

An assignment step is indicated using the assignment operator \( \[] \). This step computes the value of expression and assigns the result to the temporary variable or mutable global (see *****) \( v \). 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.

- **a.label \[] \text{expression}**
  This form of assignment sets the value of field \( \text{label} \) of record \( a \) to the value of \( \text{expression} \).
if \textit{expression}_1\text{, then} \text{step}; \text{step}; ...; \text{step}
elsif \textit{expression}_2\text{, then} \text{step}; \text{step}; ...; \text{step}
...  
elsif \textit{expression}_n\text{, then} \text{step}; \text{step}; ...; \text{step}
else \text{step}; \text{step}; ...; \text{step}
end if

An \textit{if} step computes \textit{expression}_1, which will evaluate to either \textbf{true} or \textbf{false}. If it is \textbf{true}, the first list of \textit{steps} is performed. Otherwise, \textit{expression}_2 is computed and tested, and so on. If no \textit{expression} evaluates to \textbf{true}, the list of \textit{steps} following the \textit{else} is performed. The \textit{else} clause may be omitted, in which case no action is taken when no \textit{expression} evaluates to \textbf{true}.

case \textit{expression} of
  \text{T}_1 \text{ do} \text{step}; \text{step}; ...; \text{step};
  \text{T}_2 \text{ do} \text{step}; \text{step}; ...; \text{step};
  ...
  \text{T}_n \text{ do} \text{step}; \text{step}; ...; \text{step}
else \text{step}; \text{step}; ...; \text{step}
end case

A \textit{case} step computes \textit{expression}, which will evaluate to a value \textit{v}. If \textit{v} \in \text{T}_1, then the first list of \textit{steps} is performed. Otherwise, if \textit{v} \in \text{T}_2, then the second list of \textit{steps} is performed, and so on. If \textit{v} is not a member of any \text{T}_n, the list of \textit{steps} following the \textit{else} is performed. The \textit{else} clause may be omitted, in which case \textit{v} will always be a member of some \text{T}_i.

while \textit{expression} do
  \text{step};
  \text{step};
  ...
  \text{step}
end while

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

for each \textit{x} \in \textit{expression} do
  \text{step};
  \text{step};
  ...
  \text{step}
end for each

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

\textbf{return} \textit{expression}

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

\textbf{invariant} \textit{expression}

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

\textbf{throw} \textit{expression}

A \textbf{throw} step computes \textit{expression} to obtain a value \textit{v} and begins propagating exception \textit{v} outwards, exiting partially performed steps and procedure calls until the exception is caught by a \textbf{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 an exception \( e \), then if \( e \in T \), then exception \( e \) stops propagating, variable \( v \) is bound to the value \( e \), and the second list of steps is performed. If \( e \notin T \), 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 \( \overrightarrow{N} \) that contains a nonterminal \( N \), one may replace an occurrence of \( N \) in \( \overrightarrow{N} \) 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 \( \overrightarrow{N} \) 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

\[
\text{SampleList} \ \ [
\text{\"empty\"} \\
\ldots \ \text{Identifier} \\
\text{SampleListPrefix} \\
\text{SampleListPrefix} \ \ \ldots \ \text{Identifier}
\]

states that the nonterminal \textit{SampleList} can represent one of four kinds of sequences of input tokens:

- It can represent nothing (indicated by the \text{\"empty\"} alternative).
- It can represent the terminal \ldots followed by any expansion of the nonterminal \textit{Identifier}.
- It can represent any expansion of the nonterminal \textit{SampleListPrefix}.
- It can represent any expansion of the nonterminal \textit{SampleListPrefix} followed by the terminals \ldots and \ldots and any expansion of the nonterminal \textit{Identifier}.

\subsection*{5.14.2 Lookahead Constraints}

If the phrase \text{\"[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 \textit{set}. That \textit{set} can be written as a list of terminals enclosed in curly braces. For convenience, \textit{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

\[
\text{DecimalDigit} \ \ [0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9]
\]

\[
\text{DecimalDigits} \ \ [\text{DecimalDigit} \ |
\text{DecimalDigits} \ \text{DecimalDigit}]
\]

the rule

\[
\text{LookaheadExample} \ \ [\text{n} \ \text{\[lookahead [] \{1, 3, 5, 7, 9\}\]} \ \text{DecimalDigits} \\
\text{\text{DecimalDigit} \ [lookahead [] \{\text{DecimalDigit}\}]}
\]

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

\subsection*{5.14.3 Line Break Constraints}

If the phrase \text{\"[no line break]\"} appears in the expansion of a production, it indicates that this 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

\[
\text{ReturnStatement} \ \ [\text{return} \\
\text{return} \ [\text{no line break} \text{ListExpression}{\text{allowIn}}]
\]

indicates that the second production may not be used if a line break occurs in the program between the \text{return} token and the \text{ListExpression}{\text{allowIn}}.

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.

\subsection*{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

\[
\text{\[\text{\{normal, initial\}}\]}
\]

For example, the syntactic definition

\[
\text{SampleList} \ \ [\text{\"empty\"} \\
\ldots \ \text{Identifier} \\
\text{SampleListPrefix} \\
\text{SampleListPrefix} \ \ \ldots \ \text{Identifier}
\]

states that the nonterminal \textit{SampleList} can represent one of four kinds of sequences of input tokens:

- It can represent nothing (indicated by the \text{\"empty\"} alternative).
- It can represent the terminal \ldots followed by any expansion of the nonterminal \textit{Identifier}.
- It can represent any expansion of the nonterminal \textit{SampleListPrefix}.
- It can represent any expansion of the nonterminal \textit{SampleListPrefix} followed by the terminals \ldots and \ldots and any expansion of the nonterminal \textit{Identifier}.
introduce grammar arguments \[allowIn\] and \[noIn\]. 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

\[
\text{AssignmentExpression} \allowIn, \noIn \fi \\
\text{ConditionalExpression} \allowIn, \noIn \\
\mid \text{LeftSideExpression} \allowIn = \text{AssignmentExpression} \allowIn, \noIn \\
\mid \text{LeftSideExpression} \allowIn \text{CompoundAssignment AssignmentExpression} \allowIn, \noIn
\]

expands into the following four rules:

\[
\begin{align*}
\text{AssignmentExpression} & \allowIn, \noIn \fi \\
\text{ConditionalExpression} & \allowIn, \noIn \fi \\
\mid \text{LeftSideExpression} & \allowIn = \text{AssignmentExpression} \allowIn, \noIn \\
\mid \text{LeftSideExpression} & \allowIn \text{CompoundAssignment AssignmentExpression} \allowIn, \noIn
\end{align*}
\]

\[
\begin{align*}
\text{AssignmentExpression} & \noIn, \allowIn \fi \\
\text{ConditionalExpression} & \noIn, \allowIn \fi \\
\mid \text{LeftSideExpression} & \noIn = \text{AssignmentExpression} \noIn, \allowIn \\
\mid \text{LeftSideExpression} & \noIn \text{CompoundAssignment AssignmentExpression} \noIn, \allowIn
\end{align*}
\]

\[
\begin{align*}
\text{AssignmentExpression} & \allowIn, \noIn \fi \\
\text{ConditionalExpression} & \allowIn, \noIn \fi \\
\mid \text{LeftSideExpression} & \allowIn = \text{AssignmentExpression} \allowIn, \noIn \\
\mid \text{LeftSideExpression} & \allowIn \text{CompoundAssignment AssignmentExpression} \allowIn, \noIn
\end{align*}
\]

\[
\begin{align*}
\text{AssignmentExpression} & \noIn, \allowIn \fi \\
\text{ConditionalExpression} & \noIn, \allowIn \fi \\
\mid \text{LeftSideExpression} & \noIn = \text{AssignmentExpression} \noIn, \allowIn \\
\mid \text{LeftSideExpression} & \noIn \text{CompoundAssignment AssignmentExpression} \noIn, \allowIn
\end{align*}
\]

\[
\text{AssignmentExpression} \allowIn, \noIn \fi
\]

is now an unparametrised nonterminal and processed normally by the grammar.

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 \[].

Some lexical rules contain the metaword except. These rules match any expansion that is listed before the except but that does not match any expansion after the except; if multiple expansions are listed after the 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 UnicodeCharacter except the * and / characters:

\[
\text{NonAsteriskOrSlash} \fi \text{UnicodeCharacter} \except * | / \]

### 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] = Value[Digit];
  DecimalValue[Digits1 # Digits] = 10 ¥ DecimalValue[Digits1] + Value[Digit];

proc BaseValue[Digits] (base: INTEGER): INTEGER
  [Digits do value
    d: INTEGER = Value[Digit];
    if d < base then return d else throw syntaxError end if;
  [Digits1 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] = DecimalValue[Digits];
  Value[Numeral] = DecimalValue[Digits1 # Digits]
    begin
      base: 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 the sample inputs to get the following results:

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

\[
\text{Value}[\text{Numeral} \# \text{Digits}] = \text{DecimalValue}[\text{Digits}];
\]

\[
\text{Value}[\text{Numeral} \# \text{Digits}_1 \# \text{Digits}_2]
\begin{align*}
\text{begin} \\
\text{base: INTEGER} & \text{ Integer} \text{Value}[\text{Digits}_2]; \\
\text{if base} & \geq 2 \text{ and base} \leq 10 \text{ then return BaseValue[Digits]}_1(\text{base}) \\
\text{else throw syntaxError} \\
\text{end if} \\
\text{end};
\end{align*}
\]

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

\[
\text{Value[Digit] : INTEGER} = \text{Digit’s decimal value (an integer between 0 and 9)}
\]

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 \[ \mathbf{fi} \]
Subexpression
\[ \mathbf{*} \]
Expression
\[ \mathbf{+} \]
Subexpression
\[ \mathbf{this} \]

the notation

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

is an abbreviation for the following:

\begin{verbatim}
proc \texttt{Validate[Expression]} (\texttt{cxt: CONTEXT, env: ENVIRONMENT})
    \begin{array}{l}
    \texttt{[Expression \mathbf{fi} Subexpression] do Validate[Subexpression](cxt, env);}
    \texttt{[Expression \mathbf{*} Subexpression] do}
    \texttt{Validate[Expression_0](cxt, env); Validate[Subexpression](cxt, env);}
    \texttt{[Expression \mathbf{+} Subexpression_1 + Subexpression_2] do}
    \texttt{Validate[Subexpression_1](cxt, env); Validate[Subexpression_2](cxt, env);}
    \texttt{[Expression \mathbf{fi} this] do nothing}
    \end{array}
end proc;
\end{verbatim}

Note that:

• The expanded calls to \texttt{Validate} get the same arguments \texttt{cxt} and \texttt{env} passed in to the call to \texttt{Validate} on \texttt{Expression}.

• When an expansion of \texttt{Expression} has more than one nonterminal on its right side, \texttt{Validate} is called on all of the nonterminals in left-to-right order.

• When an expansion of \texttt{Expression} has no nonterminals on its right side, \texttt{Validate} does nothing.

### 5.15.3 Action Notation Summary

The following notation is used to define semantic actions:

\texttt{Action[nonterminal]: T;}

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

\texttt{Action[nonterminal \mathbf{fi} expansion] = expression;}

This notation specifies the value that action \texttt{Action} on nonterminal \texttt{nonterminal} computes in the case where nonterminal \texttt{nonterminal} expands to the given \texttt{expansion}. \texttt{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 \texttt{expansion} can be subscripted to allow them to be unambiguously referenced by action references or nonterminal references inside \texttt{expression}.

\texttt{Action[nonterminal \mathbf{fi} expansion]: T = expression;}

This notation combines the above two — it specifies the semantic domain of the action as well as its value.
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 \texttt{step}_1\texttt{ through }\texttt{step}_m. A \texttt{return} step produces the value of the action.

\begin{verbatim}
proc \texttt{Action[nonterminal \ expansion]} (\texttt{param}_1: \texttt{T}_1, ..., \texttt{param}_n: \texttt{T}_n): \texttt{T}
  \texttt{step}_1;
  \texttt{step}_2;
  ... ;
  \texttt{step}_m
end proc;
\end{verbatim}

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

\begin{verbatim}
proc \texttt{Action[nonterminal]} (\texttt{param}_1: \texttt{T}_1, ..., \texttt{param}_n: \texttt{T}_n): \texttt{T}
  [\texttt{nonterminal \ expansion}_1] do
    step;
    ... ;
  [\texttt{nonterminal \ expansion}_2] do
    step;
    ... ;
  ...;
  [\texttt{nonterminal \ expansion}_n] do
    step;
    ... ;
end proc;
\end{verbatim}

This notation is used only when \texttt{Action} returns a procedure when applied to nonterminal \texttt{nonterminal} with several expansions \texttt{expansion}_1\texttt{ through }\texttt{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.

\texttt{Action[nonterminal]} (\texttt{param}_1: \texttt{T}_1, ..., \texttt{param}_n: \texttt{T}_n) propagates the call to \texttt{Action} to every nonterminal in the expansion of \texttt{nonterminal}.

This notation is an abbreviation stating that calling \texttt{Action} on \texttt{nonterminal} causes \texttt{Action} to be called with the same arguments on every nonterminal on the right side of the appropriate expansion of \texttt{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:

\texttt{name: T = expression;}

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

\texttt{name: T \ expansion;}

This notation defines \texttt{name} to be a mutable global value. Its initial value is the result of computing \texttt{expression}, but it may be subsequently altered using an assignment. The value is guaranteed to be a member of the semantic domain \texttt{T}. 
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 \u000A, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character \u000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \u000A 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 \u000A 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 5.13) 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 EndOfFile.
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.*
- One of the non-reserved words *exclude, get, include, set.*
- A punctuator token, which is one of !, !==, %=, &==, &&=, |==, (, ), *=, +, ++, +=, =, -=, -==, . . . , . . . , . . . , . . . , /, /=, ::, :, :, , <, <<, <<=, =, ==, ===, >, >=, >>, >>>, >=, =?, [ ], ^, ^=, ^&, ^^=, {, |, |=, ||, |=, }, ~.
- An **Identifier** token, which carries a **STRING** that is the identifier’s name.
- A **Number** token, which carries a **GENERALNUMBER** that is the number’s value.
- A **NegatedMinLong** token, which carries no value. This token is the result of evaluating 922372036854775808L.
- A **String** token, which carries a **STRING** that is the string’s value.
- A **RegularExpression** token, which carries two **STRINGS** — the regular expression’s body and its flags.

A **LineBreak**, although not considered to be a token, also becomes part of the stream of input elements and guides the process of automatic semicolon insertion (section 6.5). **EndOfInput** signals the end of the source text.

**NOTE**  The lexical grammar discards simple white space and single-line comments. They do not appear in the stream of input elements for the syntactic grammar. Comments spanning several lines become **LineBreaks**.

**TOKEN** is the semantic domain of all tokens. **INPUTELEMENT** is the semantic domain of all input elements, and is defined by:

\[
\text{INPUTELEMENT} = \{ \text{LineBreak}, \text{EndOfInput} \} \sqcup \text{TOKEN}
\]

The lexical grammar has individual characters as its terminal symbols plus the special terminal **End**, which is appended after the last input character. The lexical grammar defines three goal symbols **NextInputElement**\(^a\), **NextInputElement**\(^d\), and **NextInputElement**\(^m\), a set of productions, and instructions for translating the source text into input elements. The choice of the goal symbol depends on the syntactic grammar, which means that lexical and syntactic analyses are interleaved.

**NOTE**  The grammar uses **NextInputElement**\(^m\) if the previous lexed token was a **Number** or **NegatedMinLong**, **NextInputElement**\(^a\) if the previous token was not a **Number** or **NegatedMinLong** and a / should be interpreted as starting a regular expression, and **NextInputElement**\(^d\) if the previous token was not a **Number** or **NegatedMinLong** and a / should be interpreted as a division or division-assignment operator.

The sequence of input elements **inputElements** is obtained as follows:
Let `inputElements` be an empty sequence of input elements.
Let `input` be the input sequence of characters. Append a special placeholder `End` to the end of `input`.
Let `state` be a variable that holds one of the constants `re`, `div`, or `num`. Initialise it to `re`.
Repeat the following steps until exited:
   Find the longest possible prefix `P` of `input` that is a member of the lexical grammar’s language (see section 5.14).
   Use the start symbol `NextInputElement`, `NextInputElement^div`, or `NextInputElement^num` depending on whether `state` is `re`, `div`, or `num`, respectively. If the parse failed, signal a syntax error.
   Compute the action `Lex` on the derivation of `P` to obtain an input element `e`.
   If `e` is `EndOfInput`, then exit the repeat loop.
   Remove the prefix `P` from `input`, leaving only the yet-unprocessed suffix of `input`.
   Append `e` to the end of the `inputElements` sequence.
   If the `inputElements` sequence does not form a valid sentence prefix of the language defined by the syntactic grammar, then:
      If `e` is not `LineBreak`, but the next-to-last element of `inputElements` is `LineBreak`, then insert a `VirtualSemicolon` terminal between the next-to-last element and `e` in `inputElements`.
      If `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 `Number` token, then set `state` to `num`. Otherwise, if the `inputElements` sequence followed by the terminal `/` forms a valid sentence prefix of the language defined by the syntactic grammar, then set `state` to `div`;
      otherwise, set `state` to `re`.
End repeat
   If the `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 `inputElements`.

7.1 Input Elements

Syntax

\[
\begin{align*}
\text{NextInputElement}^* & \rightarrow \text{WhiteSpace InputElement}^* \quad \text{(WhiteSpace: 7.2)} \\
\text{NextInputElement}^\div & \rightarrow \text{WhiteSpace InputElement}^\div \\
\text{NextInputElement}^\num & \rightarrow [\text{lookahead} \rightarrow \text{ContinuingIdentifierCharacter}, \ \text{\}] \text{WhiteSpace InputElement}^\div \\
\text{InputElement}^* & \rightarrow \text{LineBreaks} \quad \text{(LineBreaks: 7.3)} \\
| & \rightarrow \text{IdentifierOrKeyword} \quad \text{(IdentifierOrKeyword: 7.5)} \\
| & \rightarrow \text{Punctuator} \quad \text{(Punctuator: 7.6)} \\
| & \rightarrow \text{NumericLiteral} \quad \text{(NumericLiteral: 7.7)} \\
| & \rightarrow \text{StringLiteral} \quad \text{(StringLiteral: 7.8)} \\
| & \rightarrow \text{RegExpLiteral} \quad \text{(RegExpLiteral: 7.9)} \\
| & \rightarrow \text{EndOfInput} \\
\text{InputElement}^\div & \rightarrow \text{LineBreaks} \quad \text{(LineBreaks: 7.3)} \\
| & \rightarrow \text{IdentifierOrKeyword} \quad \text{(IdentifierOrKeyword: 7.5)} \\
| & \rightarrow \text{Punctuator} \quad \text{(Punctuator: 7.6)} \\
| & \rightarrow \text{DivisionPunctuator} \quad \text{(DivisionPunctuator: 7.6)} \\
| & \rightarrow \text{NumericLiteral} \quad \text{(NumericLiteral: 7.7)} \\
| & \rightarrow \text{StringLiteral} \quad \text{(StringLiteral: 7.8)} \\
| & \rightarrow \text{EndOfInput} \\
\text{EndOfInput} & \rightarrow \text{End} \\
| & \rightarrow \text{LineComment} \quad \text{(LineComment: 7.4)}
\end{align*}
\]
Semantics

The grammar parameter `n` can be either `re` or `div`.

\[
\text{Lex}[\text{NextInputElement}^{n}] = \text{InputElement};
\]
\[
\text{Lex}[\text{NextInputElement}^{re} \cdot \text{WhiteSpace} \cdot \text{InputElement}^{re}] = \text{Lex}[\text{InputElement}^{re}];
\]
\[
\text{Lex}[\text{NextInputElement}^{div} \cdot \text{WhiteSpace} \cdot \text{InputElement}^{div}] = \text{Lex}[\text{InputElement}^{div}];
\]
\[
\text{Lex}[\text{NextInputElement}^{num} \cdot \{\text{lookahead} : \text{ContinuingIdentifierCharacter}, \} \cdot \text{WhiteSpace} \cdot \text{InputElement}^{div}] = \text{Lex}[\text{InputElement}^{div}];
\]

7.2 White space

Syntax

\[
\text{WhiteSpace} = \text{«empty»} | \text{WhiteSpace} \cdot \text{WhiteSpaceCharacter} | \text{WhiteSpace} \cdot \text{SingleLineBlockComment} \quad (\text{SingleLineBlockComment}: 7.4)
\]
\[
\text{WhiteSpaceCharacter} = \text{«TAB»} | \text{«VT»} | \text{«FF»} | \text{«SP»} | \text{«u00A0»} | \text{Any other character in category Zs in the Unicode Character Database}
\]

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

\[
\text{LineBreak} = \text{LineTerminator} | \text{LineComment} \cdot \text{LineTerminator} | \text{MultiLineBlockComment} \quad (\text{LineComment}: 7.4)
\]
\[
\text{LineBreaks} = \text{LineBreak} | \text{LineBreaks} \cdot \text{WhiteSpace} \cdot \text{LineBreak} \quad (\text{WhiteSpace}: 7.2)
\]
\[
\text{LineTerminator} = \text{«LF»} | \text{«CR»} | \text{«u2028»} | \text{«u2029»}
\]

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

```
LineComment [] // LineCommentCharacters

LineCommentCharacters []
  «empty»
  | LineCommentCharacters NonTerminator

SingleLineBlockComment [] /* BlockCommentCharacters */

BlockCommentCharacters []
  «empty»
  | BlockCommentCharacters NonTerminatorOrSlash
  | PreSlashCharacters /

PreSlashCharacters []
  «empty»
  | BlockCommentCharacters NonTerminatorOrAsteriskOrSlash
  | PreSlashCharacters /

MultiLineBlockComment [] /* MultiLineBlockCommentCharacters BlockCommentCharacters */

MultiLineBlockCommentCharacters []
  BlockCommentCharacters LineTerminator (LineTerminator: 7.3)
  | MultiLineBlockCommentCharacters BlockCommentCharacters LineTerminator

UnicodeCharacter [] Any Unicode character

NonTerminator [] UnicodeCharacter except LineTerminator

NonTerminatorOrSlash [] NonTerminator except /

NonTerminatorOrAsteriskOrSlash [] NonTerminator except * /
```

NOTE Comments can be either line comments or block comments. Line comments start with a // 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 LineTerminator. That LineTerminator is not considered to be part of that line comment; it is recognised separately and becomes a LineBreak. A block comment that actually spans more than one line is also considered to be a LineBreak.

7.5 Keywords and Identifiers

Syntax

```
IdentifierOrKeyword [] IdentifierName
```
Semantics

**Lex[IdentifierOrKeyword[ IdentifierName]]: INPUTELEMENT**

```plaintext
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","finally","for","function","get","goto","if","implements","import","in","include","instanceof","interface","is","namespace","native","new","null","package","private","protected","public","return","set","static","super","switch","synchronized","this","throw","throws","transient","true","try","typeof","use","var","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;
```

**NOTE** Even though the lexical grammar treats exclude, get, include, 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[IdentifierName [], IdentifierName, ContinuingIdentifierCharacterOrEscape] = LexName[IdentifierName] ⊗ [LexChar[ContinuingIdentifierCharacterOrEscape]];
LexName[IdentifierName, NullEscape] = LexName[IdentifierName];

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;

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;

The characters in the specified categories in version 3.0 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

Punctuator []

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

DivisionPunctuator []

/ { lookahead[ /, + ] }
| / =

Semantics

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

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

7.7 Numeric literals

Syntax

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
  | DecimalDigits 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 ≤ 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;
Lex[NumericLiteral []  IntegerLiteral LetterU LetterL]
  begin
    i: INTEGER []  LexNumber[IntegerLiteral];
    if i ≤ 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 [] DecimalIntegerLiteral] = 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 s [] NonZeroDecimalDigits s ASCIIDigit]
  = 10LexNumber[NonZeroDecimalDigits s] + DecimalValue[ASCIIDigit];

LexNumber[Fraction [] DecimalDigits]: RATIONAL = LexNumber[DecimalDigits]/10NDigits[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 s [] DecimalDigits s ASCIIDigit]
  = 10LexNumber[DecimalDigits s] + DecimalValue[ASCIIDigit];

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

LexNumber[HexIntegerLiteral]: INTEGER;
LexNumber[HexIntegerLiteral [] 0 LetterX HexDigit] = HexValue[HexDigit];
LexNumber[HexIntegerLiteral [] 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} \n  |  " StringChars^{double} \n
StringChars^{q} \[
  «empty» \n  |  StringChars^{q} StringChar^{q} \n  |  StringChars^{q} NullEscape \n
StringChar^{q} \[
  LiteralStringChar^{q} \n  |  \ StringStringChar^{q} \n
LiteralStringChar^{single} \[
  UnicodeCharacter except ' | \ | LineTerminator \n
LiteralStringChar^{double} \[
  UnicodeCharacter except " | \ | LineTerminator \n
StringEscape \[
  ControlEscape \n  |  ZeroEscape \n  |  HexEscape \n  |  IdentityEscape \n
IdentityEscape \ NonTerminator except _ | UnicodeAlphanumeric \n
ControlEscape \ b | f | n | r | t | v \n
ZeroEscape \ 0 [lookahead{ASCIIDigit}] \n
HexEscape \ x HexDigit HexDigit \n  |  u HexDigit HexDigit HexDigit HexDigit \n```

Semantics

```
Lex[StringLiteral]: TOKEN;
  Lex[StringLiteral \* StringChars^{single}] = a String token with the value LexString[StringChars^{single}];
  Lex[StringLiteral \ " StringChars^{double}] = a String token with the value LexString[StringChars^{double}];

LexString[StringChars^{q}]: STRING;
  LexString[StringChars^{q} «empty»] = "",
  LexString[StringChars^{q} StringChars^{q} StringChar^{q}] = LexString[StringChars^{q}] $\oplus$ LexChar[StringChar^{q}];
  LexString[StringChars^{q} StringChars^{q} NullEscape] = LexString[StringChars^{q}];

LexChar[StringChar^{q}]: CHARACTER;
  LexChar[StringChar^{q} LiteralStringChar^{q}] = LiteralStringChar^{q};
  LexChar[StringChar^{q} \ StringEscape] = LexChar[StringEscape];
```
LexChar[StringEscape]: CHARACTER;
LexChar[StringEscape ControlEscape] = LexChar[ControlEscape];
LexChar[StringEscape ZeroEscape] = LexChar[ZeroEscape];
LexChar[StringEscape HexEscape] = LexChar[HexEscape];
LexChar[StringEscape IdentityEscape] = IdentityEscape;

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

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 { ASCIIDigit}]]: CHARACTER = ‘NUL’;

LexChar[HexEscape]: CHARACTER;
LexChar[HexEscape x HexDigit1 HexDigit2]
    = codeToCharacter(16.HexValue[HexDigit1] + HexValue[HexDigit2]);
LexChar[HexEscape u HexDigit1 HexDigit2 HexDigit3 HexDigit4]

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

RegExpLiteral [ RegExpBody RegExpFlags

RegExpFlags [ «empty»
| RegExpFlags ContinuingIdentifierCharacterOrEscape
| RegExpFlags NullEscape (ContinuingIdentifierCharacterOrEscape: 7.5)

RegExpBody [ / [lookahead { * } ] RegExpChars /

RegExpChars [ RegExpChar
| RegExpChars RegExpChar

RegExpChar [ OrdinaryRegExpChar
| \ NonTerminator (NonTerminator: 7.4)

OrdinaryRegExpChar [ NonTerminator except \ /]
Semantics

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

LexString[RegExpFlags]: STRING;
LexString[RegExpFlags [] «empty»] = «””»;
LexString[RegExpFlags0 [] RegExpFlags1 ContinuingIdentifierCharacterOrEscape]
   = LexString[RegExpFlags1] ⊕ [LexChar[ContinuingIdentifierCharacterOrEscape]];
LexString[RegExpFlags0 [] RegExpFlags1 NullEscape] = LexString[RegExpFlags1];

LexString[RegExpBody [] / [lookahead[] { * }] RegExpChars / ]: STRING = LexString[RegExpChars];

LexString[RegExpChars]: STRING;
LexString[RegExpChars [] RegExpChar] = LexString[RegExpChar];
LexString[RegExpChars0 [] RegExpChars1 RegExpChar]
   = LexString[RegExpChars1] ⊕ LexString[RegExpChar];

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

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 undefined, 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 simple instance, a method closure, a date, a regular expression, or a package object. These kinds of objects are described in the subsections below.
**OBJECT** is the semantic domain of all possible objects and is defined as:

```
OBJECT = UNDEFINED NULL BOOLEAN LONG ULONG FLOAT32 FLOAT64 CHARACTER STRING
       Namespace NAME CompoundAttribute CLASS SimpleInstance MethodClosure Date RegExp Package;
```

A **PRIMITIVEOBJECT** is either **undefined, null**, a Boolean, a signed or unsigned 64-bit integer, a single or double-precision floating-point number, a character, or a string:

```
PRIMITIVEOBJECT = UNDEFINED NULL BOOLEAN LONG ULONG FLOAT32 FLOAT64 CHARACTER STRING;
```

A **BINDINGOBJECT** is an object that can bind local properties:

```
BINDINGOBJECT = CLASS SIMPLEINSTANCE REGEXP DATE PACKAGE;
```

The semantic domain **OBJECTOPT** consists of all objects as well as the tag **none** which denotes the absence of an object or a variable that has yet to be initialised. **none** is not a value visible to ECMAScript programmers.

```
OBJECTOPT = OBJECT {none};
```

The semantic domain **INTEGEROPT** consists of all integers as well as **none**:

```
INTEGEROPT = INTEGER {none};
```

### 9.1.1 Undefined

There is exactly one **undefined** value. The semantic domain **UNDEFINED** consists of that one value.

```
UNDEFINED = {undefined}
```

### 9.1.2 Null

There is exactly one **null** value. The semantic domain **NULL** consists of that one value.

```
NULL = {null}
```

### 9.1.3 Booleans

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

The semantic domain **BOOLEANOPT** consists of the tags **true,** **false,** and **none**:

```
BOOLEANOPT = BOOLEAN {none};
```

### 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.

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.

```
STRINGOPT = STRING {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.

```
Field  Contents  Note
```
name STRING The namespace’s name used by `toString`

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>STRING</td>
<td>The namespace qualifier</td>
</tr>
<tr>
<td>id</td>
<td>STRING</td>
<td>The name</td>
</tr>
</tbody>
</table>

The semantic notation `ns::id` is a shorthand for QualifiedName[namespace: ns, id: id]]

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>STRING</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>enumerable</td>
<td>BOOLEAN</td>
<td>true if the enumerable 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, virtual, or final if one of these attributes has been given; none if not.</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 without arguments was given; none if the override attribute was not given. OVERRIDEMODIFIER = {none, true, false, undefined}</td>
</tr>
<tr>
<td>prototype</td>
<td>BOOLEAN</td>
<td>true if the prototype attribute has been given</td>
</tr>
<tr>
<td>unused</td>
<td>BOOLEAN</td>
<td>true if the unused attribute has been given</td>
</tr>
</tbody>
</table>

NOTE An implementation that supports host-defined attributes will add other fields to the tuple above

\[
\text{Attribute} = \text{Boolean}[] \text{Namespace}[] \text{CompoundAttribute}
\]

\[
\text{AttributeOptNotFalse} = \{\text{none, true}\}[] \text{Namespace}[] \text{CompoundAttribute}
\]

9.1.8 Classes

Programmer-visible class objects are represented as CLASS 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>localBindings</td>
<td>LOCALBINDING[]</td>
<td>Map of qualified names to static members defined in this class section *****</td>
</tr>
<tr>
<td>super</td>
<td>CLASSOpt</td>
<td>This class’s immediate superclass or null if none</td>
</tr>
</tbody>
</table>
instanceMembers: `INSTANCEMEMBER {}`

Map of qualified names to instance members defined or overridden in this class.

complete: `BOOLEAN`

`true` after all members of this class have been added to this `CLASS` record.

name: `STRING`

This class’s name.

prototype: `OBJECTOpt`

The default value of the `super` field of newly created simple instances of this class; `none` for most classes.

typeOfString: `STRING`

A string to return if `typeof` is invoked on this class’s instances.

privateNamespace: `NAMESPACE`

This class’s `private` namespace.

dynamic: `BOOLEAN`

`true` if this class or any of its ancestors was defined with the `dynamic` attribute.

final: `BOOLEAN`

`true` if this class cannot be subclassed.

defaultValue: `OBJECTOpt`

When a variable whose type is this class is defined but not explicitly initialised, the variable’s initial value is `defaultValue`, which must be an instance of this class. The class `Never` has no values, so that class’s (and only that class’s) `defaultValue` is `none`.

bracketRead: `OBJECT CLASS OBJECT Phase`

A procedure to call when this class is used in a call expression. The parameters are the `this` argument, the list of arguments, and the phase of evaluation (section 9.4).

bracketWrite: `OBJECT CLASS OBJECT Phase`

A procedure to call when this class is used in a `new` expression. The parameters are the list of arguments and the phase of evaluation (section 9.4).

bracketDelete: `OBJECT CLASS OBJECT Phase`

delete: `OBJECT CLASS MULTINAME ObjectPhase ObjectOpt`

A procedure to call to initialise a newly created instance of this class or `none` if no special initialisation is needed. `init` is called by `construct`.

enumerate: `OBJECT Object`

A procedure to call to determine whether a given object is an instance of this class.

call: `OBJECT Object Phase`

A procedure to call when this class is used in a call expression. The parameters are the `this` argument, the list of arguments, and the phase of evaluation (section 9.4).

write: `OBJECT CLASS MULTINAME EnvironmentOpt Phase ObjectOpt`

A procedure to call when this class is used in a `new` expression. The parameters are the list of arguments and the phase of evaluation (section 9.4).

read: `OBJECT CLASS MULTINAME Environment Opt`

A procedure to call to initialise a newly created instance of this class or `none` if no special initialisation is needed. `init` is called by `construct`.
A procedure to call when a value is assigned to a varia parameter, or result whose type is this class. The argument implicitCoerce can be any value, which may or may not be instance of this class; the result must be an instance of this clas the coercion is not appropriate, implicitCoerce should throw exception if its second argument is false or return null (as lon null is an instance of this class) if its second argument is true.

9.1.9 Simple Instances

Instances of programmer-defined classes as well as of some built-in classes are represented as SimpleInstance records (see section 5.11) with the fields below. Prototype-based objects are also SimpleInstance records.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING{}</td>
<td>Map of qualified names to local properties (including dynamic properties, if any) of this instance</td>
</tr>
<tr>
<td>super</td>
<td>OBJECTOPT</td>
<td>Optional link to the next object in this instance’s prototype chain</td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td>If true, no more local properties may be added to this instance</td>
</tr>
<tr>
<td>type</td>
<td>CLASS</td>
<td>This instance’s type</td>
</tr>
<tr>
<td>slots</td>
<td>SLOT{ }</td>
<td>A set of slots that hold this instance’s fixed property values</td>
</tr>
<tr>
<td>call</td>
<td>(OBJECT {} SIMPLEINSTANCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>) OBJECT {} PHASE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>) OBJECT {} {none}</td>
<td>Either none or a procedure to call when this instance is used in a call expression. The procedure takes an OBJECT (the this value), a SIMPLEINSTANCE (the called instance), a list of OBJECT argument values, and a PHASE (see section 9.4) and produces an OBJECT result</td>
</tr>
<tr>
<td>construct</td>
<td>(SIMPLEINSTANCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>) OBJECT {} PHASE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>) OBJECT {} {none}</td>
<td>Either none or a procedure to call when this instance is used in a new expression. The procedure takes a SIMPLEINSTANCE (the instance on which new was invoked), a list of OBJECT argument values, and a PHASE (see section 9.4) and produces an OBJECT result</td>
</tr>
<tr>
<td>env</td>
<td>ENVIRONMENTOPT</td>
<td>Either none or the environment in which call or construct should look up non-local variables</td>
</tr>
</tbody>
</table>

9.1.10 Uninstantiated Functions

An UninstantiatedFunction record (see section 5.11) has the fields below. It is not an instance in itself but creates a SimpleInstance when instantiated with an environment. UninstantiatedFunction records represent functions with
variables inherited from their enclosing environments; supplying the environment turns such a function into a `SIMPLEINSTANCE`.

<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>Values to be transferred into the generated <code>SIMPLEINSTANCE</code>’s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>corresponding fields</td>
</tr>
<tr>
<td>buildPrototype</td>
<td><code>BOOLEAN</code></td>
<td>If true, the generated <code>SIMPLEINSTANCE</code> gets a separate</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>prototype</code> property with its own prototype object</td>
</tr>
<tr>
<td>length</td>
<td><code>INTEGER</code></td>
<td>The value to store in the generated <code>SIMPLEINSTANCE</code>’s <code>length</code></td>
</tr>
<tr>
<td>call</td>
<td><code>OBJECT</code></td>
<td>Values to be transferred into the generated <code>SIMPLEINSTANCE</code>’s</td>
</tr>
<tr>
<td></td>
<td><code>SIMPLEINSTANCE</code></td>
<td>corresponding fields</td>
</tr>
<tr>
<td></td>
<td><code>OBJECT</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>PHASE</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>{none}</code></td>
<td></td>
</tr>
<tr>
<td>construct</td>
<td><code>SIMPLEINSTANCE</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>OBJECT</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>PHASE</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>{none}</code></td>
<td></td>
</tr>
<tr>
<td>instantiations</td>
<td><code>SIMPLEINSTANCE{}</code></td>
<td>Set of prior instantiations. This set serves only to precisely</td>
</tr>
<tr>
<td></td>
<td></td>
<td>specify the closure sharing optimization and would not be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>needed in any actual implementation.</td>
</tr>
</tbody>
</table>

### 9.1.11 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>

### 9.1.12 Dates

Instances of the `Date` class are represented as `DATE` 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>localBindings</td>
<td><code>LOCALBINDING{}</code></td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>super</td>
<td><code>OBJECTOPT</code></td>
<td></td>
</tr>
<tr>
<td>sealed</td>
<td><code>BOOLEAN</code></td>
<td></td>
</tr>
<tr>
<td>timeValue</td>
<td><code>INTEGER</code></td>
<td>The date expressed as a count of milliseconds from January 1, 1970 UTC</td>
</tr>
</tbody>
</table>

### 9.1.13 Regular Expressions

Instances of the `RegExp` class are represented as `REGEXP` 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>localBindings</td>
<td><code>LOCALBINDING{}</code></td>
<td>Same as in <code>SIMPLEINSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>super</td>
<td><code>OBJECTOPT</code></td>
<td></td>
</tr>
<tr>
<td>sealed</td>
<td><code>BOOLEAN</code></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td><code>STRING</code></td>
<td>This regular expression’s source pattern</td>
</tr>
</tbody>
</table>
lastIndex | INTEGER | The string position at which to start the next regular expression match
--- | --- | ---
global | BOOLEAN | true if the regular expression flags included the flag \( g \)
ignoreCase | BOOLEAN | true if the regular expression flags included the flag \( i \)
multiline | BOOLEAN | true if the regular expression flags included the flag \( m \)

### 9.1.14 Packages and Global Objects

Programmer-visible packages and global objects 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>localBindings</td>
<td>LOCAL_BINDING{}</td>
<td>Same as in <code>SIMPLE_INSTANCES</code> (section 9.1.9)</td>
</tr>
<tr>
<td>super</td>
<td>OBJECTOPT</td>
<td></td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td></td>
</tr>
<tr>
<td>internalNamespace</td>
<td>NAMESPACE</td>
<td>This package’s or global object’s <code>internal</code> namespace</td>
</tr>
</tbody>
</table>

### 9.2 Objects with Limits

A `LIMITED_INSTANCE` 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>OBJECT</td>
<td>The value of <code>expr</code> to which the <code>super</code> subexpression was applied; if <code>expr</code> wasn’t given, defaults to the value of <code>this</code>. The value of <code>instance</code> is always an instance of one of the <code>limit</code> class’s descendants.</td>
</tr>
<tr>
<td>limit</td>
<td>CLASS</td>
<td>The immediate superclass of the class inside which the <code>super</code> subexpression was applied</td>
</tr>
</tbody>
</table>

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

`OBJ_OPTIONAL_LIMIT` is the result of a subexpression that can produce either an `OBJECT` or a `LIMITED_INSTANCE`:

\[
\text{OBJ_OPTIONAL_LIMIT} = \text{OBJECT} \sqcup \text{LIMITED_INSTANCE}
\]

### 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.

\[
\text{REFERENCE} = \text{LEXICAL_REFERENCE} \sqcup \text{DOT_REFERENCE} \sqcup \text{BRACKET_REFERENCE};
\]

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

\[
\text{OBJ_OR_REF} = \text{OBJECT} \sqcup \text{REFERENCE}
\]

A `LEXICAL_REFERENCE` 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. `LEXICAL_REFERENCE` 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>
<tbody>
<tr>
<td>env</td>
<td>ENVIRONMENT</td>
<td>The environment in which the reference was created.</td>
</tr>
<tr>
<td>variableMultiname</td>
<td>MULTINAME</td>
<td>A nonempty set of qualified names to which this reference can refer</td>
</tr>
</tbody>
</table>
The semantic domain is defined functions. 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 phases of expression evaluation:

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

9.5 Contexts

A CONTEXT record (see section 5.11) 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 was in effect at the point where the reference was created</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>

9.6 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.

\[ \text{LABEL} = \text{STRING} \cup \{ \text{default} \} \]

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.
Field  | Contents | Note
--- | --- | ---
breakTargets | LABEL{} | The set of labels that are valid destinations for a `break` statement
continueTargets | LABEL{} | The set of labels that are valid destinations for a `continue` statement

### 9.7 Semantic Exceptions

All values thrown by the semantics’ `throw` steps and caught by `try-catch` steps (see section 5.13.3) are members of the semantic domain `SEMANTICEXCEPTION`, defined as follows:

\[
\text{SEMANTICEXCEPTION} = \text{OBJECT} \mid \text{CONTROLTRANSFER};
\]

\[
\text{CONTROLTRANSFER} = \text{BREAK} \mid \text{CONTINUE} \mid \text{RETURN};
\]

The semantics `throw` four different kinds of values:
- An `OBJECT` is thrown as a result of encountering an error or evaluating an ECMAScript `throw` statement
- A `BREAK` tuple is thrown as a result of evaluating an ECMAScript `break` statement
- A `CONTINUE` tuple is thrown as a result of evaluating an ECMAScript `continue` statement
- A `RETURN` tuple is thrown as a result of evaluating an ECMAScript `return` statement

A `BREAK` tuple (see section 5.10) has the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT</td>
<td>The value produced by the last statement to be executed before the <code>break</code></td>
</tr>
<tr>
<td>label</td>
<td>LABEL</td>
<td>The label that is the target of the <code>break</code></td>
</tr>
</tbody>
</table>

A `CONTINUE` tuple (see section 5.10) has the fields below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT</td>
<td>The value produced by the last statement to be executed before the <code>continue</code></td>
</tr>
<tr>
<td>label</td>
<td>LABEL</td>
<td>The label that is the target of the <code>continue</code></td>
</tr>
</tbody>
</table>

A `RETURN` tuple (see section 5.10) has the field below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT</td>
<td>The value of the expression in the <code>return</code> statement or <code>undefined</code> if omitted</td>
</tr>
</tbody>
</table>

### 9.8 Function Support

The `FUNCTIONKIND` semantic domain encodes a general kind of a function:

\[
\text{FUNCTIONKIND} = \{\text{plainFunction}, \text{uncheckedFunction}, \text{prototypeFunction}, \text{instanceFunction}, \text{constructorFunction}\};
\]

These kinds represent the following:
- A `plainFunction` is a static function whose signature is checked when it is called. This function is not a prototype-based constructor and cannot be used in a `new` expression.
- A `prototypeFunction` is a static function whose signature is checked when it is called. This function is also a prototype-based constructor and may be used in a `new` expression.
- An `uncheckedFunction` is a static function whose signature is not checked when it is called. This function is also a prototype-based constructor and may be used in a `new` expression.
- An `instanceFunction` is an instance method whose signature is checked when it is called.
- A `constructorFunction` is a class constructor whose signature is checked when it is called.

The subset of static function kinds has its own semantic domain `STATICFUNCTIONKIND`:
\texttt{STATICFUNCTIONKIND} = \{\texttt{plainFunction}, \texttt{uncheckedFunction}, \texttt{prototypeFunction}\};

Two of the above five function kinds, plain and instance functions, can be defined either normally or as getters or setters. This distinction is encoded by the \texttt{HANDLING} semantic domain:

\texttt{HANDLING} = \{\texttt{normal}, \texttt{get}, \texttt{set}\};

### 9.9 Environment Frames

Environments contain the bindings that are visible from a given point in the source code. An \texttt{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 \texttt{SYSTEMFRAME}. The next-to-last frame is always a \texttt{PACKAGE}. A \texttt{WITHFRAME} is always preceded by a \texttt{LOCALFRAME}, so the first frame is never a \texttt{WITHFRAME}.

\texttt{ENVIRONMENT} = \texttt{FRAME[]}

The semantic domain \texttt{ENVIRONMENTOPT} consists of all environments as well as the tag \texttt{none} which denotes the absence of an environment:

\texttt{ENVIRONMENTOPT} = \texttt{ENVIRONMENT} [] \{\texttt{none}\};

A frame contains bindings defined at a particular scope in a program. A frame is either the top-level system frame, a package, a function parameter frame, a class, a local (block) frame, or a \texttt{with} statement frame:

\texttt{FRAME} = \texttt{NONWITHFRAME} [] \texttt{WITHFRAME};

\texttt{NONWITHFRAME} = \texttt{SYSTEMFRAME} [] \texttt{PACKAGE} [] \texttt{PARAMETERFRAME} [] \texttt{CLASS} [] \texttt{LOCALFRAME};

Some frames hold the runtime values of variables and other definitions; these frames are called \textit{instantiated frames}. Other frames, called \textit{uninstantiated frames}, are used as templates for making (instantiating) instantiated frames. The static analysis done by \texttt{Validate} generates instantiated frames for a few top-level scopes and uninstantiated frames for other scopes; the \texttt{preinst} parameter to \texttt{Validate} governs whether it generates instantiated or uninstantiated frames.

### 9.9.1 System Frame

The top-level frame containing predefined constants, functions, and classes is represented as a \texttt{SYSTEMFRAME} record (see section 5.11) with the field below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING[]</td>
<td>Map of qualified names to definitions in this frame</td>
</tr>
</tbody>
</table>

### 9.9.2 Function Parameter Frames

Frames holding bindings for invoked functions are represented as \texttt{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>localBindings</td>
<td>LOCALBINDING[]</td>
<td>Map of qualified names to definitions in this function</td>
</tr>
<tr>
<td>kind</td>
<td>FUNCTIONKIND</td>
<td>See section 9.8</td>
</tr>
<tr>
<td>handling</td>
<td>HANDLING</td>
<td>See section 9.8</td>
</tr>
<tr>
<td>callsSuperconstructor</td>
<td>BOOLEAN</td>
<td>A flag that indicates whether a call to the superclass’s constructor has been detected during static analysis of a class constructor. Always false if \texttt{kind} is not \texttt{constructorFunction}.</td>
</tr>
<tr>
<td>superconstructorCalled</td>
<td>BOOLEAN</td>
<td>If \texttt{kind} is a \texttt{constructorFunction}, this flag indicates whether the superclass’s constructor has been called yet during execution of this constructor. Always true if \texttt{kind} is not \texttt{constructorFunction}.</td>
</tr>
<tr>
<td>this</td>
<td>OBJECTOPT</td>
<td>The value of this; \texttt{none} if this function doesn’t define this or it defines this but the value is not available because this function hasn’t</td>
</tr>
</tbody>
</table>
been called yet

parameters PARAMETER[] List of this function’s parameters

rest VARIABLEOPT The parameter variable for collecting any extra arguments that may be passed or none if no extra arguments are allowed

returnType CLASS The function’s declared return type, which defaults to Object if not provided

PARAMETERFRAMEOPT consists of all parameter frames as well as none:

PARAMETERFRAMEOPT = PARAMETERFRAME [] {none};

9.9.2.1 Parameters

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>VARIABLE [] DYNAMICVar</td>
<td>The local variable that will hold this parameter’s value</td>
</tr>
<tr>
<td>default</td>
<td>OBJECTOPT</td>
<td>This parameter’s default value; if none, this parameter is required</td>
</tr>
</tbody>
</table>

9.9.3 Local Frames

Frames holding bindings for blocks and other statements that can hold local bindings are represented as LOCALFRAME records (see section 5.11) with the field below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>localBindings</td>
<td>LOCALBINDING[]</td>
<td>Map of qualified names to definitions in this frame</td>
</tr>
</tbody>
</table>

9.9.4 With Frames

Frames holding bindings for with statements are represented as WITHFRAME records (see section 5.11) with the field below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECTOPT</td>
<td>The value of the with statement’s expression or none if not evaluated yet</td>
</tr>
</tbody>
</table>

9.10 Environment Bindings

In general, accesses of members are either read or write operations. The tags read and write indicate these respectively. The semantic domain ACCESS consists of these two tags:

ACCESS = {read, write};

Some members are visible only for read or only for write accesses; other members are visible to both read and write accesses. The tag readWrite indicates that a member is visible to both kinds of accesses. The semantic domain ACCESSSET consists of the three possible access visibilities:

ACCESSSET = {read, write, readWrite};

NOTE Access sets indicate visibility, not permission to perform the desired access. Immutable members generally have the access readWrite but an attempt to write one results in an error. Trying to write to member with the access read would not even find the member, and the write would proceed to search an object’s parent hierarchy for another matching member.

9.10.1 Static Bindings

A LOCALBINDING 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).
A local member is either **forbidden**, a variable, a dynamic variable, a getter, or a setter:

```
LOCAL_MEMBER = {forbidden} □ VARIABLE □ DYNAMIC_VAR □ GETTER □ SETTER;
```

A **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 **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>CLASS</td>
<td>Type of values that may be stored in this variable</td>
</tr>
<tr>
<td>value</td>
<td>VARIABLE_VALUE</td>
<td>This variable’s current value; <strong>future</strong> if the variable has not been declared yet; <strong>uninitialised</strong> if the variable must be written before it can be read</td>
</tr>
<tr>
<td>immutable</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this variable’s value may not be changed once set</td>
</tr>
<tr>
<td>setup</td>
<td>(() □ CLASSOPT)</td>
<td>A semantic procedure that performs the <strong>Setup</strong> action on the variable or constant definition. <strong>none</strong> if the action has already been performed; <strong>busy</strong> if the action is in the process of being performed and should not be reentered.</td>
</tr>
<tr>
<td>initialiser</td>
<td>INITIALISER</td>
<td>A semantic procedure that computes a variable’s initialiser specified by the programmer. <strong>none</strong> if no initialiser was given or if it has already been evaluated; <strong>busy</strong> if the initialiser is being evaluated now and should not be reentered.</td>
</tr>
<tr>
<td>initialiserEnv</td>
<td>ENVIRONMENT</td>
<td>The environment to provide to <strong>initialiser</strong> if this variable is a compile-time constant</td>
</tr>
</tbody>
</table>

The semantic domain **VARIABLEOPT** consists of all variables as well as **none**:

```
VARIABLEOPT = VARIABLE □ {none};
```

A variable’s value can be either an object, **none** (used when the variable has not been initialised yet), or an uninstantiated function (compile time only):

```
VARIABLE_VALUE = {none} □ OBJECT □ UNINSTANTIATED_FUNCTION;
```

A **INITIALISER** is a semantic procedure that takes environment and phase parameters and computes a variable’s initial value.

```
INITIALISER = ENVIRONMENT □ PHASE □ OBJECT;
```

```
INITIALISEROPT = INITIALISER □ {none};
```

A **DYNAMIC_VAR** record (see section 5.11) has the fields below and describes one hoisted or dynamic variable.

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>OBJECT □ UNINSTANTIATED_FUNCTION</td>
<td>This variable’s current value; may be an uninstantiated function at compile time</td>
</tr>
<tr>
<td>sealed</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this variable cannot be deleted using the <strong>delete</strong> operator</td>
</tr>
</tbody>
</table>

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

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

\[
\text{INSTANCEMEMBER} = \text{INSTANCEVARIABLE} \mid \text{INSTANCEMETHOD} \mid \text{INSTANCEGETTER} \mid \text{INSTANCESETTER};
\]

\[
\text{INSTANCEMEMBEROPT} = \text{INSTANCEMEMBER} \mid \{\text{none}\};
\]

An **INSTANCEVARIABLE** record (see section 5.11) has the fields below and describes one instance variable or constant definition. This record is also used as a key to look up an instance’s **SLOT** (see section 9.1.9.1).

<table>
<thead>
<tr>
<th>Field</th>
<th>Contents</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this instance variable</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this instance variable may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this instance variable’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>type</td>
<td>CLASS</td>
<td>Type of values that may be stored in this variable</td>
</tr>
<tr>
<td>defaultValue</td>
<td>OBJECTOPT</td>
<td>This variable’s default value; <strong>none</strong> if not provided</td>
</tr>
<tr>
<td>immutable</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this variable’s value may not be changed once set</td>
</tr>
</tbody>
</table>

The semantic domain **INSTANCEVARIABLEOPT** consists of all instance variables as well as **none**:

\[
\text{INSTANCEVARIABLEOPT} = \text{INSTANCEVARIABLE} \mid \{\text{none}\};
\]

An **INSTANCEMETHOD** 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>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this instance method</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this instance method may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td><strong>true</strong> if this instance method’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>signature</td>
<td>PARAMETERFRAME</td>
<td>This method’s signature encoded in the <strong>PARAMETERFRAME</strong>’s parameters, rest, and returnType fields</td>
</tr>
<tr>
<td>call</td>
<td>OBJECT [OBJECT] [PHASE] [OBJECT]</td>
<td>A procedure to call when this instance method is invoked. The procedure takes a <strong>this OBJECT</strong>, a list of argument <strong>OBJECTs</strong>, and a <strong>PHASE</strong> (see section 9.4) and produces an <strong>OBJECT</strong> result</td>
</tr>
</tbody>
</table>

An **INSTANCEGETTER** 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>
</table>
multiname  MULTINAME  The set of qualified names for this getter
final  BOOLEAN  true if this getter may not be overridden in subclasses
enumerable  BOOLEAN  true if this getter’s public name should be visible in a for-in statement
signature  PARAMETERFRAME  This getter’s signature encoded in the PARAMETERFRAME’s parameters, rest, and returnType fields
call  OBJECT  PHASE  OBJECT  A procedure to call to read the value, passing it the this value and the current mode of expression evaluation

An INSTANCESETTER 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>multiname</td>
<td>MULTINAME</td>
<td>The set of qualified names for this setter</td>
</tr>
<tr>
<td>final</td>
<td>BOOLEAN</td>
<td>true if this getter may not be overridden in subclasses</td>
</tr>
<tr>
<td>enumerable</td>
<td>BOOLEAN</td>
<td>true if this getter’s public name should be visible in a for-in statement</td>
</tr>
<tr>
<td>signature</td>
<td>PARAMETERFRAME</td>
<td>This setter’s signature encoded in the PARAMETERFRAME’s parameters, rest, and returnType fields</td>
</tr>
<tr>
<td>call</td>
<td>OBJECT  PHASE  ()</td>
<td>A procedure to call to write the value, passing it the this value, the value being written, and the current mode of expression evaluation</td>
</tr>
</tbody>
</table>

10 Data Operations

This chapter describes core algorithms defined on the values in chapter 9. The algorithms here are not ECMAScript 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.

\[
\text{proc unsignedWrap32}(i: \text{INTEGER}): \{0 \ldots 2^{32} - 1\} \\
\text{return bitwiseAnd}(i, 0xFFFFFFFF) \\
\text{end proc;}
\]

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

\[
\text{proc signedWrap32}(i: \text{INTEGER}): \{-2^{31} \ldots 2^{31} - 1\} \\
j: \text{INTEGER}  \\
\text{bitwiseAnd}(i, 0xFFFFFFFF); \\
\text{if } j \geq 2^{31} \text{ then } j \leftarrow 2^{32} - 1 \text{ end if; } \\
\text{return } j \\
\text{end proc;}
\]

unsignedWrap64(i) returns \( i \) converted to a value between 0 and \( 2^{64} - 1 \) inclusive, wrapping around modulo \( 2^{64} \) if necessary.

\[
\text{proc unsignedWrap64}(i: \text{INTEGER}): \{0 \ldots 2^{64} - 1\} \\
\text{return bitwiseAnd}(i, 0xFFFFFFFFFFFFFFFF) \\
\text{end proc;}
\]

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

\[
\text{proc signedWrap64}(i: \text{INTEGER}): \{-2^{63} \ldots 2^{63} - 1\} \\
j: \text{INTEGER}  \\
\text{bitwiseAnd}(i, 0xFFFFFFFFFFFFFFFF); \\
\text{if } j \geq 2^{63} \text{ then } j \leftarrow 2^{64} - 1 \text{ end if; } \\
\text{return } j \\
\text{end proc;}
\]
proc signedWrap64(i: INTEGER): \{-2^{63} \ldots 2^{63} - 1\}
    j: INTEGER \text{ bitwiseAnd}(i, 0xFFFFFFFFFFFFFFFF);
    if j \geq 2^{63} then j = j - 2^{64} end if;
    return j
end proc;

proc truncateToInteger(x: GENERALNUMBER): INTEGER
    case x of
        \{+∞_{132}, +∞_{64}, -∞_{132}, -∞_{64}, NaN_{132}, NaN_{64}\} do return 0;
        FINITEFLOAT32 do return truncateFiniteFloat32(x);
        FINITEFLOAT64 do return truncateFiniteFloat64(x);
        LONG \text{ ULONG} do return x.value
    end case
end proc;

proc checkInteger(x: GENERALNUMBER): INTEGEROPT
    case x of
        \{NaN_{132}, NaN_{64}, +∞_{132}, +∞_{64}, -∞_{132}, -∞_{64}\} do return none;
        \{+zero_{132}, +zero_{64}, -zero_{132}, -zero_{64}\} do return 0;
        LONG \text{ ULONG} do return x.value;
        NONZEROFINITEFLOAT32 \text{ NONZEROFINITEFLOAT64} do
            r: RATIONAL \text{ x.value};
            if r \text{ INTEGER} then return none end if;
            return r
        end case
end proc;

proc integerToLong(i: INTEGER): GENERALNUMBER
    if \{-2^{63} \leq i < 2^{63} - 1\} then return i.long
    elsif \{2^{63} \leq i < 2^{64} - 1\} then return i ulong
    else return realToFloat64(i) end if
end proc;

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

proc rationalToLong(q: RATIONAL): GENERALNUMBER
    if q \text{ INTEGER} then return integerToLong(q)
    elsif \{q \leq 2^{53}\} then return realToFloat64(q)
    elsif \{q < -2^{63} - 1/2 \text{ or} q \geq 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 \{-2^{63} \leq i \leq 2^{64} - 1\};
        if \{i < 2^{63}\} then return i long else return i ulong end if
    end if
end proc;
proc rationalToULong(q: RATIONAL): GENERALNUMBER
  if q \in INTEGER then return integerToULong(q)
  elsif |q| \leq 2^{53} then return realToFloat64(q)
  elsif q < -2^{63} - 1/2 or q \geq 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 -2^{63} \leq i \leq 2^{64} - 1;
    if i \geq 0 then return iulong else return ilong 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 NONZEROFINITEFLOAT64 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 \in \{NaN_{f32}, NaN_{f64}\} or y \in \{NaN_{f32}, NaN_{f64}\} then return unordered
  elsif x \in \{+\infty_{f32}, +\infty_{f64}\} and y \in \{+\infty_{f32}, +\infty_{f64}\} then return equal
  elsif x \in \{-\infty_{f32}, -\infty_{f64}\} and y \in \{-\infty_{f32}, -\infty_{f64}\} then return equal
  elsif x \in \{+\infty_{f32}, +\infty_{f64}\} or y \in \{-\infty_{f32}, -\infty_{f64}\} then return greater
  elsif x \in \{-\infty_{f32}, -\infty_{f64}\} or y \in \{+\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

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 Void;
    null do return null;
    boolean do return boolean;
    long do return long;
    ulong do return ulong;
    float32 do return float32;
    float64 do return float64;
    character do return character;
    string do return string;
    namespace do return namespace;
    compoundattribute do return attribute;
    class do return class;
    simpleinstance do return o.type;
    methodclosure do return function;
    date do return date;
    regexp do return regexp;
    package do return package
  end case
end proc;

10.2.2 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 do return true;
    boolean do return o;
    long ulong do return o.value ≠ 0;
    float32 do return o {+zero, -zero, NaN} ;
    float64 do return o {+zero, -zero, NaN} ;
    string do return o ≠ "" ;
    character namespace compoundattribute class simpleinstance methodclosure package do
      date regexp package do
      throw a TypeError exception;
    end case
    return true
  end case
end proc;

10.2.3 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
    undefined do return NaN;
    null {false} do return +zero;
    {true} do return 1.0;
    generalnumber do return o;
    character namespace compoundattribute class simpleinstance methodclosure package do
      date regexp package do
      throw a TypeError exception;
    end case
    return NaN
  end case
end proc;
10.2.4 `toString`

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

```plaintext
proc toString(o: Object, phase: Phase): String
  case o of
    UNDEFINED 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 ???
    COMPOUNDATTRIBUTE do ???
    CLASS do ???
    METHODATTRIBUTE do ???
    DATE do ???
    REGEXP do ???
    PACKAGE do ???
  end case
end proc;
```

`integerToString()` 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.

```plaintext
proc integerToString(i: Integer): String
  if i < 0 then return ["-"] ⊕ integerToString(-i) end if;
  q: Integer [] i/10;
  r: Integer [] i - q*10;
  c: Character [] codeToCharacter(r + characterToCode(‘0’));
  if q = 0 then return [c] else return integerToString(q) ⊕ [c] end if
end proc;
```

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

```plaintext
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 \( 10^{21} \) 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 \( -\text{zero}_{32} \) would become \( +\text{zero}_{32} \)).
proc float32ToString(x: FLOAT32): STRING
  case x of
    {NaN[32]} do return "NaN";
    {+zero[32], -zero[32]} 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, 10^{k−1} ≤ s ≤ 10^k, realToFloat32(10^{n−k}) = x, and k is as small as possible.
      note k is the number of digits in the decimal representation of s, s is not divisible by 10, and the least significant digit of s is not necessarily uniquely determined by the above 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 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 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 10^{21} 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 −zero[64] would become +zero[64]).
**10.2.7 toQualifiedName**

`toQualifiedName(o, phase)` coerces an object `o` to a qualified name. If `phase` is `compile`, only compile-time conversions are permitted.

```plaintext
proc toQualifiedName(o: OBJECT, phase: PHASE): QUALIFIEDNAME
    return public::(toString(o, phase))
end proc;
```

**10.2.6 toPrimitive**

`toPrimitive(o: OBJECT, hint: OBJECT, phase: PHASE): PRIMITIVEOBJECT`

```plaintext
proc toPrimitive(o: OBJECT, hint: OBJECT, phase: PHASE): PRIMITIVEOBJECT
    case o of
        PRIMITIVEOBJECT do return o;
        NAMESPACE [] COMPOUNDATTRIBUTE [] CLASS [] SIMPLEINSTANCE [] METHODCLOSURE [] REGEXP []
            PACKAGE do
                return toString(o, phase);
            end case
    end case
end proc;
```

**10.2.7 toClass**

`toClass(o: OBJECT): CLASS`

```plaintext
proc toClass(o: OBJECT): CLASS
    if o [] CLASS then return o else throw a TypeError exception end if
end proc;
```
10.2.8 Attributes

*combineAttributes*(a, b) returns the attribute that results from concatenating the attributes a and b.

```plaintext
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, enumerable: 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
    note At this point both a and b are compound attributes.
    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 an AttributeError exception — attributes a and b have conflicting contents
    else
        return COMPOUNDATTRIBUTE{namespaces: a.namespaces [] b.namespaces,
            explicit: a.explicit or b.explicit, enumerable: a.enumerable or b.enumerable,
            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.

```plaintext
proc toCompoundAttribute(a: ATTRIBUTEOptNotFalse): COMPOUNDATTRIBUTE
    case a of
        {none, true} do
            return COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false, dynamic: false, memberMod: none, overrideMod: none, prototype: false, unused: false}
        NAMESPACE do
            return COMPOUNDATTRIBUTE{namespaces: {a}, explicit: false, enumerable: false, dynamic: false, memberMod: none, overrideMod: none, prototype: false, unused: false}
        COMPOUNDATTRIBUTE do return a
    end case
end proc;
```

10.3 Access Utilities

```plaintext
proc accessesOverlap(accesses1: ACCESSSET, accesses2: ACCESSSET): BOOLEAN
    return accesses1 = accesses2 or accesses1 = readWrite or accesses2 = readWrite
end proc;
```
proc objectSupers(o: OBJECT): OBJECT
    { }
    if o[] BINDINGOBJECT then return { } end if;
    super: OBJECTOPT [] o.super;
    if super = none then return { } end if;
    return {super} [] objectSupers(super)
end proc;

proc findSlot(o: OBJECT, id: INSTANCEDVARIABLE): SLOT
    note o must be a SIMPLEINSTANCE.
    matchingSlots: SLOT[] {s | s.s [] o.slots such that s.id = id};
    return the one element of matchingSlots
end proc;

setupVariable(v) runs Setup and initialises the type of the variable v, making sure that Setup is done at most once and does not reenter itself.

proc setupVariable(v: VARIABLE)
    setup: () [] CLASSOPT [] {none, busy} [] v.setup;
    case setup of
        () [] CLASSOPT do
            v.setup [] busy;
            type: CLASSOPT [] setup();
            if type = none then type [] Object end if;
            v.type [] type;
            v.setup [] none;
            {none} do nothing;
            {busy} do
                throw a ConstantError exception — a constant’s type or initialiser cannot depend on the value of that constant
        end case
end proc;

proc writeVariable(v: VARIABLE, newValue: OBJECT, clearInitialiser: BOOLEAN): OBJECT
    coercedValue: OBJECT [] v.type.implicitCoerce(newValue, false);
    if clearInitialiser then v.initialiser [] none end if;
    if v.immutable and (v.value ≠ none or v.initialiser ≠ none) then
        throw a ReferenceError exception — cannot initialise a const variable twice
    end if;
    v.value [] coercedValue;
    return coercedValue
end proc;

10.4 Environmental Utilities

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;

If env is from within a function’s body, getEnclosingParameterFrame(env) returns the PARAMETERFRAME for the innermost such function; otherwise, it returns none.
proc getEnclosingParameterFrame(env: ENVIRONMENT): PARAMETERFRAMEOPT

  for each frame \[\text{env}\] do
    case frame of
      LOCALFRAME \[\text{WITHFRAME}\] do nothing;
      PARAMETERFRAME do return frame;
      SYSTEMFRAME \[\text{PACKAGE}\] \[\text{CLASS}\] do return none
    end case
  end for each;
  return none
end proc;

getRegionalEnvironment(env) returns all frames in \[\text{env}\] up to and including the first regional frame. A regional frame is either any frame other than a with frame or local block frame, a local block frame directly enclosed in a class, or a local block frame directly enclosed in a with frame directly enclosed in a class.

proc getRegionalEnvironment(env: ENVIRONMENT): FRAME[]

  i: INTEGER \[\text{0}\];
  while \[\text{env}\][i] \[\text{LOCALFRAME}\] \[\text{WITHFRAME}\] do i \[\text{i + 1}\] end while;
  if \[\text{env}\][i] \[\text{CLASS}\] then while i \[\text{\neq 0}\] and \[\text{env}\][i] \[\text{LOCALFRAME}\] do i \[\text{i - 1}\] end while end if;
  return \[\text{env}\][0 ... i]
end proc;

getRegionalFrame(env) returns the most specific regional frame in \[\text{env}\].

proc getRegionalFrame(env: ENVIRONMENT): FRAMEx regionalEnv: FRAME[] \[\text{getRegionalEnvironment}\[\text{env}\]];
  return regionalEnv[\text{regionalEnv} - 1]
end proc;

proc getPackageFrame(env: ENVIRONMENT): PACKAGE

  pkg: FRAMEx env[\|\text{env}\| - 2];
  note The penultimate frame \[\text{pkg}\] is always a \text{PACKAGE}.
  return pkg
end proc;

10.5 Property Lookup

proc findLocalMember(o: NONWITHFRAME \[\text{SIMPLEINSTANCE}\] \[\text{REGEXP}\] \[\text{DATE}\], multiline: MULTINAME, access: ACCESS): LOCALMEMBEROPT

  matchingLocalBindings: LOCALBINDING[] \[\text{\{} b | \text{\!\!b \in o.localBindings such that}
    b.qname \[\text{multiline}\] \[\text{and accessesOverlap(b.accesses, access)}\]\text{\}\};

  note If the same member was found via several different bindings \[\text{b}\], then it will appear only once in the set \text{matchingLocalMembers}.

  matchingLocalMembers: LOCALMEMBER[] \[\text{\{} b.content | \text{\!\!b \in matchingLocalBindings}\};

  if matchingLocalMembers = \text{\{} \text{then return none}
  elsif \|matchingLocalMembers\| = 1 \text{then return the one element of matchingLocalMembers}
  else
    throw a \text{ReferenceError} exception — this access is ambiguous because the bindings it found belong to several different local members
  end if
end proc;
getDerivedInstanceMember(c, mBase, accesses) returns the most derived instance member whose name includes that of mBase and whose access includes accesses. The caller of getDerivedInstanceMember ensures that such a member always exists. If accesses is readWrite then it is possible that this search could find both a getter and a setter defined in the same class; in this case either the getter or the setter is returned at the implementation’s discretion.
proc getDerivedInstanceMember(c: CLASS, mBase: INSTANCEMEMBER, accesses: ACCESSSET): INSTANCEMEMBER
if some m ∈ c.instanceMembers satisfies mBase.multiname ⊈ m.multiname and
   accessesOverlap(instanceMemberAccesses(m), accesses) then
   return m
else return getDerivedInstanceMember(c.super, mBase, accesses)
end if
end proc;

proc lookupInstanceMember(c: CLASS, gname: QUALIFIEDNAME, access: ACCESS): INSTANCEMEMBEROPT
mBase: INSTANCEMEMBEROPT [] findBaseInstanceMember(c, {gname}, access);
if mBase = none then return none end if;
return getDerivedInstanceMember(c, mBase, access)
end proc;

proc readImplicitThis(env: ENVIRONMENT): OBJECT
frame: PARAMETERFRAMEOPT [] getEnclosingParameterFrame(env);
if frame = none then
   throw a ReferenceError exception — can’t access instance members outside an instance method without supplying
   an instance object
end if;
this: OBJECTOPT [] frame.this;
if this = none then
   throw a ReferenceError exception — can’t access instance members inside a non-instance method without
   supplying an instance object
end if;
if frame.kind ⊈ {instanceFunction, constructorFunction} then
   throw a ReferenceError exception — can’t access instance members inside a non-instance method without
   supplying an instance object
end if;
if not frame.superconstructorCalled then
   throw an UninitializedError exception — can’t access instance members from within a constructor before the
   superconstructor has been called
end if;
return this
end proc;

10.6 Reading
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
result: OBJECTOPT;
case r of
   OBJECT do result [] r;
   LEXICALREFERENCE do result [] lexicalRead(r.env, r.variableMultiname, phase);
   DOTREFERENCE do
      result [] r.limit.read(r.base, r.limit, r.propertyMultiname, none, phase);
   BRACKETREFERENCE do result [] r.limit.bracketRead(r.base, r.limit, r.args, phase)
end case;
if result ≠ none then return result
else
   throw a ReferenceError exception — property not found, and no default value is available
end if
end proc;

dotRead(o, multiname, phase) is a simplified interface to read the multiname property of o.
proc dotRead(o: Object, multiname: Multiname, phase: Phase): Object
limit: Class [] objectType(o);
result: ObjectOpt [] limit.read(o, limit, multiname, none, phase);
if result = none then
    throw a ReferenceError exception — property not found, and no default value is available
end if;
return result
end proc;

proc indexRead(o: Object, i: Integer, phase: Phase): ObjectOpt
if i < 0 or i ≥ arrayLimit then throw a RangeError exception end if;
limit: Class [] objectType(o);
return limit.bracketRead(o, limit, i, ulong, phase)
end proc;

proc defaultBracketRead(o: Object, limit: Class, args: Object[], phase: Phase): ObjectOpt
if args ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
end if;
qname: QualifiedName [] toQualifiedName(args[0], phase);
return limit.read(o, limit, qname, none, phase)
end proc;

proc lexicalRead(env: Environment, multiname: Multiname, phase: Phase): Object
i: Integer = 0;
while i < |env| do
    frame: Frame [] env[i];
    result: ObjectOpt [] none;
case frame of
    Package [] Class do
        limit: Class [] objectType(frame);
        result [] limit.read(frame, limit, multiname, env, phase);
    SystemFrame [] ParameterFrame [] LocalFrame do
        m: LocalMemberOpt [] findLocalMember(frame, multiname, read);
        if m ≠ none then result [] readLocalMember(m, phase) end if;
    WithFrame do
        value: ObjectOpt [] frame.value;
        if value = none then
            case phase of
            {compile} do
                throw a ConstantError exception — cannot read a with statement’s frame from a constant expression;
            {run} do
                throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s expression has been evaluated
            end case
        end if;
        limit: Class [] objectType(value);
        result [] limit.read(value, limit, multiname, env, phase)
    end case;
    if result ≠ none then return result end if;
i [] i + 1
end while;
throw a ReferenceError exception — no variable found with the name multiname
end proc;
mBase: InstanceMemberOpt = findBaseInstanceMember(limit, multiname, read);
if mBase ≠ none then return readInstanceMember(o, limit, mBase, phase) end if;
if limit ≠ objectType(o) then return none end if;
m: {none} LOCALMember INSTANCEMember findCommonMember(o, multiname, read, false);
case m of
  {none} do
    if env = none and o = SimpleInstance DATE RegExp Package and not o.sealed then
      case phase of
        {compile} do
          throw a ConstantError exception — constant expressions cannot read dynamic properties;
        end case
      end if
    else return none
  end if;
  LOCALMember do return readLocalMember(m, phase);
  INSTANCEMember do
    if o = Class or env = none then
      throw a ReferenceError exception — cannot read an instance member without supplying an instance
    end if;
    this: Object = readImplicitThis(env);
    return readInstanceMember(this, objectType(this), m, phase)
  end case
end proc;

readInstanceProperty(o, qname, phase) is a simplified interface to defaultReadProperty used to read to instance members that are known to exist.

proc readInstanceProperty(o: Object, qname: QualifiedName, phase: Phase): Object
  c: Class = objectType(o);
  mBase: InstanceMemberOpt = findBaseInstanceMember(c, {qname}, read);
  note readInstanceProperty is only called in cases where the instance property is known to exist, so mBase cannot be none here.
  return readInstanceMember(o, c, mBase, phase)
end proc;
proc readInstanceMember(this: OBJECT, c: CLASS, mBase: INSTANCEMEMBER, phase: PHASE): OBJECT
m: INSTANCEMEMBER [] getDerivedInstanceMember(c, mBase, read);
case m of
INSTANCEVARIABLE do
  if phase = compile and not m.immutable then
    throw a ConstantError exception — constant expressions cannot read mutable variables
  end if;
v: OBJECTOPT [] findSlot(this, m).value;
if v = none then
  case phase of
    {compile} do
      throw a ConstantError exception — cannot read an uninitialised const variable from a constant expression;
    {run} do
      throw an UninitializedError exception — cannot read a const instance variable before it is initialised
    end case
  end if;
return v;
INSTANCEMETHOD do return METHODCLOSURE[this: this, method: m[]]
INSTANCEGETTER do return m.call(this, phase);
INSTANCESETTER do
  m cannot be an INSTANCESETTER because these are only represented as write-only members.
end case
end proc;
proc readLocalMember(m: LOCALMEMBER, phase: PHASE): OBJECT
  case m of
    {forbidden} do
      throw a ReferenceError exception — cannot access a definition from an outer scope if any block inside the current region shadows it;
  DYNAMIC_VAR do
    if phase = compile then
      throw a ConstantError exception — constant expressions cannot read mutable variables
    end if;
    value: OBJECT [] UNINSTANTIATED_FUNCTION [] m.value;
    note value can be an UNINSTANTIATED_FUNCTION only during the compile phase, which was ruled out above.
    return value;
  VARIABLE do
    if phase = compile and not m.immutable then
      throw a ConstantError exception — constant expressions cannot read mutable variables
    end if;
    value: VARIABLE_VALUE [] m.value;
    case value of
      OBJECT do return value;
      {none} do
        if not m.immutable then
          case phase of
            {compile} do
              throw a ConstantError exception — cannot read a mutable variable from a constant expression;
            {run} do throw an UninitializedError exception
          end case
          end if;
        note Try to run a const variable’s initialiser if there is one.
        setupVariable(m);
        initialiser: INITIALISER [] {none, busy} [] m.initialiser;
        if initialiser [] {none, busy} then
          case phase of
            {compile} do
              throw a ConstantError exception — a constant expression cannot access a constant with a missing or recursive initialiser;
            {run} do throw an UninitializedError exception
          end case
        end if;
        m.initialiser [] busy;
        coercedValue: OBJECT;
        try
          newValue: OBJECT [] initialiser(m.initialiserEnv, compile);
          coercedValue [] writeVariable(m, newValue, true)
        catch x: SEMANTIC_EXCEPTION do
          note If initialisation failed, restore m.initialiser to its original value so it can be tried later.
          m.initialiser [] initialiser;
          throw x
        end try;
        return coercedValue;
      UNINSTANTIATED_FUNCTION do
        note An uninstantiated function can only be found when phase = compile.
        throw a ConstantError exception — an uninstantiated function is not a constant expression
      end case;
    Getter do
      env: ENVIRONMENT_OPT [] m.env;
      if env = none then
        note An uninstantiated getter can only be found when phase = compile.
throw a ConstantException exception — an uninstantiated getter is not a constant expression
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;

10.7 Writing

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

proc writeReference(r: OBR, newValue: OBJECT, phase: {run})
  result: {none, ok};
case r of
  Object do
    throw a ReferenceError exception — a non-reference is not a valid target of an assignment;
  LexicalReference do
    lexicalWrite(r.env, r.variableMultiname, newValue, not r.strict, phase);
    result [] ok;
  DotReference do
    result [] r.limit.write(r.base, r.limit, r.propertyMultiname, none, true, newValue, phase);
  BracketReference do
    result [] r.limit.bracketWrite(r.base, r.limit, r.args, newValue, phase)
end case;
if result = none then
  throw a ReferenceError exception — property not found and could not be created
end if
end proc;

dotWrite(o, multiname, newValue, phase) is a simplified interface to write newValue into the multiname property of o.

proc dotWrite(o: OBJECT, multiname: MULTINAME, newValue: OBJECT, phase: {run})
  limit: CLASS [] objectType(o);
  result: {none, ok} [] limit.write(o, limit, multiname, none, true, newValue, phase);
  if result = none then
    throw a ReferenceError exception — property not found and could not be created
  end if
end proc;

proc indexWrite(o: OBJECT, i: INTEGER, newValue: OBJECT, phase: {run})
if i < 0 or i ≥ arrayLimit then throw a RangeException exception end if;
  limit: CLASS [] objectType(o);
  result: {none, ok} [] limit.bracketWrite(o, limit, [i_long], newValue, phase);
  if result = none then
    throw a ReferenceError exception — property not found and could not be created
  end if
end proc;

proc defaultBracketWrite(o: OBJECT, limit: CLASS, args: OBJECT[], newValue: OBJECT, phase: {run}): {none, ok}
if args ≠ 1 then
  throw an ArgumentException exception — exactly one argument must be supplied
end if;
qname: QualifiedName [] toQualifiedName(args[0], phase);
return limit.write(o, limit, {qname}, none, true, newValue, phase)
end proc;
proc lexicalWrite(env: ENVIRONMENT, multiname: MULTINAME, newValue: OBJECT, createIfMissing: BOOLEAN, 
phase: {run})
i: INTEGER [] 0;
while i < |env| do
    frame: FRAME [] env[i];
    result: {none, ok} [] none;
    case frame of
        PACKAGE [] CLASS do
            limit: CLASS [] objectType(frame);
            result [] limit.write(frame, limit, multiname, env, false, newValue, phase);
        SYSTEMFRAME [] PARAMETERFRAME [] LOCALFRAME do
            m: LOCALMEMBEROPT [] findLocalMember(frame, multiname, write);
            if m ≠ none then writeLocalMember(m, newValue, phase); result [] ok
        end if;
        WITHFRAME do
            value: OBJECTOPT [] frame.value;
            if value = none then
                throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s
expression has been evaluated
            end if;
            limit: CLASS [] objectType(value);
            result [] limit.write(value, limit, multiname, env, false, newValue, phase)
        end case;
        if result = ok then return end if;
    i [] i + 1
end while;
if createIfMissing then
    pkg: PACKAGE [] getPackageFrame(env);
    note Try to write the variable into pkg again, this time allowing new dynamic bindings to be created dynamically.
    limit: CLASS [] objectType(pkg);
    result: {none, ok} [] limit.write(pkg, limit, multiname, env, true, newValue, phase);
    if result = ok then return end if
end if;
    throw a ReferenceError exception — no existing variable found with the name multiname and one could not be created
end proc;
The caller must make sure that the created property does not already exist and does not conflict with any other property.

```javascript
proc defaultWriteProperty(o: Object, limit: Class, multiname: Multiname, env: EnvironmentOpt, createIfMissing: Boolean, newValue: Object, phase: {run}): {none, ok} mBase: InstanceMemberOpt [] findBaseInstanceMember(limit, multiname, write);
if mBase ≠ none then writeInstanceMember(o, limit, mBase, newValue, phase); return ok end if;
if limit ≠ objectType(o) then return none end if;
m: {none} [] LocalMember [] InstanceMember [] findCommonMember(o, multiname, write, true);
case m of
{none} do
if createIfMissing and o [] SimpleInstance [] Date [] RegExp [] Package and not o.sealed and
(some qname [] multiname satisfies qname.namespace = public) then
note Before trying to create a new dynamic property named qname, check that there is no read-only fixed
property with the same name.
if findBaseInstanceMember(objectType(o), {qname}, read) = none and
findCommonMember(o, {qname}, read, true) = none then
createDynamicProperty(o, qname, false, true, newValue);
return ok end if
end if;
return none;
LocalMember do writeLocalMember(m, newValue, phase); return ok;
InstanceMember do
if o [] Class or env = none then
throw a ReferenceError exception — cannot write an instance member without supplying an instance end if;
this: Object [] readImplicitThis(env);
writeInstanceMember(this, objectType(this), m, newValue, phase);
return ok end case
end proc;
```

The caller must make sure that the created property does not already exist and does not conflict with any other property.

```javascript
proc createDynamicProperty(o: SimpleInstance [] Date [] RegExp [] Package, qname: QualifiedName, sealed: Boolean, enumerable: Boolean, newValue: Object) dv: DynamicVar [] newValue: DynamicVar [] value: newValue, sealed: sealed, o.localBindings [] o.localBindings [] {LocalBinding [] name: qname, accesses: readWrite, content: dv, explicit: false, enumerable: enumerable} end proc;
```

```javascript
proc writeInstanceMember(this: Object, c: Class, mBase: InstanceMember, newValue: Object, phase: {run}) m: InstanceMember [] getDerivedInstanceMember(c, mBase, write);
case m of
InstanceVariable do
s: Slot [] findSlot(this, m);
coercedValue: Object [] m.type.implicitCoerce(newValue, false);
if m.immutable and s.value ≠ none then
throw a ReferenceError exception — cannot initialise a const instance variable twice end if;
s.value [] coercedValue;
InstanceMethod do
throw a ReferenceError exception — cannot write to an instance method;
InstanceGetter do
m cannot be an InstanceGetter because these are only represented as read-only members.
InstanceSetter do m.call(this, newValue, phase)
end case
end proc;
```
proc writeLocalMember(m: LOCALMEMBER, newValue: OBJECT, phase: {run})
  case m of
    {forbidden} do
      throw a ReferenceError exception — cannot access a definition from an outer scope if any block inside the current region shadows it;
    VARIABLE do writeVariable(m, newValue, false);
    DYNAMICVAR do m.value = newValue;
    GETTER do
      m cannot be a GETTER because these are only represented as read-only members.
    SETTER do
      env: ENVIRONMENTOPT = m.env;
      note All instances are resolved for the run phase, so env ≠ none.
      m.call(newValue, env, phase)
    end case
  end
end proc;

10.8 Deleting

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: OBJORREF, strict: BOOLEAN, phase: {run}): BOOLEAN
  result: BOOLEAN
  case r of
    OBJECT do
      if strict then
        throw a ReferenceError exception — a non-reference is not a valid target for delete in strict mode
      else result = true
      end if;
    LEXICALREFERENCE do result = lexicalDelete(r.env, r.variableMultiname, phase);
    DOTREFERENCE do
      result = r.limit.delete(r.base, r.limit, r.propertyMultiname, none, phase);
    BRACKETREFERENCE do
      result = r.limit.bracketDelete(r.base, r.limit, r.args, phase)
    end case;
    if result ≠ none then return result else return true end if
  end proc;

proc defaultBracketDelete(o: OBJECT, limit: CLASS, args: OBJECT[], phase: {run}): BOOLEAN
  if args ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  qname: QUALIFIEDNAME = toQualifiedName(args[0], phase);
  return limit.delete(o, limit, {qname}, none, phase)
proc lexicalDelete(env: ENVIRONMENT, multiname: MULTINAME, phase: {run}): BOOLEAN
i: INTEGER = 0;
while i < |env| do
    frame: FRAME = env[i];
    result: BOOLEANOpt = none;
    case frame of
    PACKAGE » CLASS do
        limit: CLASS = objectType(frame);
        result = limit.delete(frame, limit, multiname, env, phase);
    SYSTEMFRAME » PARAMETERFRAME » LOCALFRAME do
        if findLocalMember(frame, multiname, write) ≠ none then result = false;
    end if;
    WITHFRAME do
        value: OBJECTOpt = frame.value;
        if value = none then
            throw an UninitializedError exception — cannot read a with statement’s frame before that statement’s expression has been evaluated
        end if;
        limit: CLASS = objectType(value);
        result = limit.delete(value, limit, multiname, env, phase);
    end case;
    if result ≠ none then return result end if;
    i = i + 1
end while;
return true
end proc;

proc defaultDeleteProperty(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOpt, phase: {run}): BOOLEANOpt
if findBaseInstanceMember(limit, multiname, write) ≠ none then return false end if;
if limit ≠ objectType(o) then return none end if;
m: {none} » LOCALMEMBER » INSTANCEMEMBER = findCommonMember(o, multiname, write, true);
    case m of
    {none} do return none;
    {forbidden} do
        throw a ReferenceError exception — cannot access a definition from an outer scope if any block inside the current region shadows it;
    VARIABLE » GETTER » SETTER do return false;
    DYNAMICVAR do
        if m.sealed then return false
    else
        o.localBindings = {b | b in o.localBindings such that b.qname = multiname or b.content ≠ m};
        return true
    end if;
    INSTANCEMEMBER do
        if o CLASS or env = none then return false end if;
        readImplicitThis(env);
        return false
    end case
end proc;
10.9 Enumerating

```plaintext
proc defaultEnumerate(o: OBJECT): OBJECT {
  e1: OBJECT() enumerateInstanceMembers(objectType(o));
  e2: OBJECT() enumerateCommonMembers(o);
  return e1 » e2;
}
```

```plaintext
proc enumerateInstanceMembers(c: CLASS): OBJECT {
  e: OBJECT() enumerateInstanceMembers(c.instanceMembers)
  for each m ∈ c.instanceMembers do
    if m.enumerable then
      e » e {qname.id | qnameŒm.multiname such that qname.namespace = public}
    end if
  end for each;
  super: CLASSOPT = c.super;
  if super = none then return e else return e » enumerateInstanceMembers(super) end if
}
```

```plaintext
proc enumerateCommonMembers(o: OBJECT): OBJECT {
  e: OBJECT() enumerateCommonMembers(o)
  for each s ∈ {o} enumerateInstanceMembers(s) do
    for each b ∈ s.localBindings do
      if b.enumerable and b.qnamenamespace = public then e » e {b.qname.id}
    end if
  end for each
  return e
}
```

10.10 Creating Instances

```plaintext
proc createSimpleInstance(c: CLASS, super: OBJECTOPT, call: (OBJECT Œ SIMPLEINSTANCE Œ OBJECT Œ PHASE Œ OBJECT) Œ {none}, construct: (SIMPLEINSTANCE Œ OBJECT Œ PHASE Œ OBJECT) Œ {none}, env: ENVIRONMENTOPT): SIMPLEINSTANCE {
  slots: SLOT() for each s ∈ ancestors(c) do
    for each m ∈ s.instanceMembers do
      if m INSTANCEVARIABLE then
        slot: SLOT() new SLOT{Id: m, value: m.defaultValue}
        slots » slots {slot}
      end if
    end for each
  end for each;
}
```
10.11 Adding Local Definitions

```javascript
proc defineLocalMember(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE{});
  overrideMod: OVERRIDE_MODIFIER, explicit: BOOLEAN, accesses: ACCESS_SET, m: LOCAL_MEMBER): MULTINAME
innerFrame: NON_WITH_FRAME [] env[0];
if overrideMod ≠ none then
  throw an AttributeError exception — a local definition cannot have the override attribute
end if;
if explicit and innerFrame [] PACKAGE then
  throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
end if;
namespaces2: NAMESPACE{} [] namespaces;
if namespaces2 = { } then namespaces2 [] { public } end if;
multiname: MULTINAME { ns::id | ns namespaces2 };
regionalEnv: FRAME[] getRegionalEnvironment(env);
if some b innerFrame.localBindings satisfies
  b.qname [] multiname and accessesOverlap(b.accesses, accesses) then
  throw a DefinitionError exception — duplicate definition in the same scope
end if;
if innerFrame [] CLASS and id = innerFrame.name then
  throw a DefinitionError exception — a static member of a class cannot have the same name as the class,
regardless of the namespace
end if;
for each frame [] regionalEnv[1 ...] do
  if frame [] WITH_FRAME and (some b [] frame.localBindings satisfies b.qname [] multiname and
accessesOverlap(b.accesses, accesses) and b.content ≠ forbidden) then
    throw a DefinitionError exception — this definition would shadow one defined in an outer scope within the
same region
  end if;
end for each;
newBindings: LOCAL_BINDING{} [] { LOCAL_BINDING[] name: qname, accesses: accesses, content: m,
explicit: explicit, enumerable: true[] qname [] multiname};
innerFrame.localBindings [] innerFrame.localBindings [] newBindings;
```

**Note** 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.

```javascript
newForbiddenBindings: LOCAL_BINDING{} [] { LOCAL_BINDING[] name: qname, accesses: accesses,
content: forbidden, explicit: true, enumerable: true[] qname [] multiname};
for each frame [] regionalEnv[1 ...] do
  if frame [] WITH_FRAME then
    frame.localBindings [] frame.localBindings [] newForbiddenBindings
  end if;
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 package 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. A `var` hoisted variable may be hoisted into the
`PARAMETER_FRAME` if there is already a parameter with the same name; a `function` hoisted variable is never hoisted into the
`PARAMETER_FRAME` and will shadow a parameter with the same name for compatibility with ECMAScript Edition 3. If
there are multiple `function` definitions, the initial value is the last `function` definition.
proc defineHoistedVar(env: ENVIRONMENT, id: STRING, initialValue: OBJECT [] UninstantiatedFunction): DYNAMICVAR
  qname: QualifiedName [] public::id;
  regionalEnv: FRAME [] getRegionalEnvironment(env);
  regionalFrame: FRAME [] regionalEnv[regionalEnv[regionalEnv] – 1];
  note  env is either a PACKAGE or a PARAMETERFRAME because hoisting only occurs into package or function scope.
  existingBindings: LOCALBINDING {} [] {b | b regionalFrame.localBindings such that b.qname = qname};
  if (existingBindings = {} or initialValue ≠ undefined) and regionalFrame [] PARAMETERFRAME and
      |regionalEnv| ≥ 2 then
    regionalFrame [] regionalEnv[regionalEnv[regionalEnv] – 2];
    existingBindings [] {b | b regionalFrame.localBindings such that b.qname = qname}
  end if;
  if existingBindings ≠ {} then
    v: DYNAMICVAR [] new DYNAMICVAR[]value: initialValue, sealed: true[]
    regionalFrame.localBindings [] regionalFrame.localBindings [] {LOCALBINDING[name: qname, accesses: readWrite, content: v, explicit: false, enumerable: true];
      return v
  elsif existingBindings ≠ 1 then
    throw a DefinitionError exception — a hoisted definition conflicts with a non-hoisted one
  else
    b: LOCALBINDING [] the one element of existingBindings;
    m: LOCALMEMBER [] b.content;
    if b.accesses ≠ readWrite or m DYNAMICVAR then
      throw a DefinitionError exception — a hoisted definition conflicts with a non-hoisted one
    end if;
    note  At this point 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 end if;
    m.sealed [] true;
    regionalFrame.localBindings [] regionalFrame.localBindings – {b};
    regionalFrame.localBindings [] regionalFrame.localBindings [] {LOCALBINDING[enumerable: true], other fields from b[];
      return m
  end if
end proc;

10.12 Adding Instance Definitions

proc searchForOverrides(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEMEMBEROPT
  mBase: INSTANCEMEMBEROPT [] none;
  s: CLASSOPT [] c.super;
  if s ≠ none then
    for each qname [] multiname do
      m: INSTANCEMEMBEROPT [] findBaseInstanceMember(s, {qname}, accesses);
      if mBase = none then mBase [] m
      elsif m ≠ none and m ≠ mBase then
        throw a DefinitionError exception — cannot override two separate superclass methods at the same time
      end if
    end for each
  end if;
return mBase
end proc;
proc `defineInstanceMember(c: CLASS, ctx: CONTEXT, id: STRING, namespaces:_NAMESPACE{},
    overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, m: INSTANCEMEMBER): INSTANCEMEMBEROPT
if explicit then
    throw an AttributeError exception — the explicit attribute can only be used at the top level of a package
end if;
accesses: ACCESSSET[] instanceMemberAccesses(m);
requestedMultiname: MULTINAME[] {ns::id | ns openNamespaces};
openMultiname: MULTINAME[] {ns::id | ns ctx.openNamespaces};
definedMultiname: MULTINAME;
searchedMultiname: MULTINAME;
if requestedMultiname = {} then
    definedMultiname = public::id;
    searchedMultiname = openMultiname;
    note definedMultiname = searchedMultiname because the public namespace is always open.
else definedMultiname = requestedMultiname, searchedMultiname = requestedMultiname
end if;
mBase: INSTANCEMEMBEROPT[] searchForOverrides(c, searchedMultiname, accesses);
mOverridden: INSTANCEMEMBEROPT[] none;
if mBase ≠ none then
    mOverridden = getDerivedInstanceMember(c, mBase, accesses);
definedMultiname = mOverridden.multiname;
if not (requestedMultiname = definedMultiname) then
    throw a DefinitionError exception — cannot extend the set of a member’s namespaces when overriding it
end if;
goodKind: BOOLEAN;
case m of
    INSTANCEVARIABLE do goodKind = mOverridden = INSTANCEVARIABLE;
    INSTANCEGETTER do
goodKind = mOverridden = INSTANCEVARIABLE = INSTANCEGETTER;
    INSTANCESETTER do
goodKind = mOverridden = INSTANCEVARIABLE = INSTANCESETTER;
    INSTANCEMETHOD do goodKind = mOverridden = INSTANCEMETHOD
end case;
if not goodKind then
    throw a DefinitionError exception — a method can override only another method, a variable can override only another variable, a getter can override only a getter or a variable, and a setter can override only a setter or a variable
end if;
if mOverridden.final then
    throw a DefinitionError exception — cannot override a final member
end if;
end if;
if some m2 ∈ c.instanceMembers satisfies m2.multiname ≠ {} and accessesOverlap(instanceMemberAccesses(m2), accesses) then
    throw a DefinitionError exception — duplicate definition in the same scope
end if;
case overrideMod of
    none do
    if mBase ≠ none then
        throw a DefinitionError exception — a definition that overrides a superclass’s member must be marked with the override attribute
    end if;
    if searchForOverrides(c, openMultiname, accesses) ≠ none then
        throw a DefinitionError exception — this definition is hidden by one in a superclass when accessed without a namespace qualifier; in the rare cases where this is intentional, use the override(false) attribute
    end if;
    false do

if mBase ≠ none then
    throw a DefinitionError exception — this definition is marked with override(false) but it overrides a superclass’s member
end if;
{true} do
    if mBase = none then
        throw a DefinitionError exception — this definition is marked with override or override(true) but it doesn’t override a superclass’s member
    end if;
    {undefined} do nothing
end case;
m.multiname [] definedMultiname;
c.instanceMembers [] c.instanceMembers [] {m};
return mOverridden
end proc;

10.13 Instantiation

proc instantiateFunction(uf: UNINSTANTIATEDFUNCTION, env: ENVIRONMENT): SIMPLEINSTANCE
    c: CLASS [] uf.type;
i: SIMPLEINSTANCE [] createSimpleInstance(c, c.prototype, uf.call, uf.construct, env);
dotWrite(i, {public::"length"}, realToFloat64(uf.length), run);
if uf.buildPrototype then
    prototype: OBJECT [] Prototype.construct([], run);
dotWrite(prototype, {public::"constructor"}, i, run);
dotWrite(i, {public::"prototype"}, prototype, run)
end if;
instantiations: SIMPLEINSTANCE {} [] uf.instantiations;
if instantiations != {} then
    Suppose that instantiateFunction were to choose at its discretion some element i2 of instantiations, assign i2.env [] env, and return i. If the behaviour of doing that assignment were observationally indistinguishable by the rest of the program from the behaviour of returning i without modifying i2.env, then the implementation may, but does not have to, return i2 now, discarding (or not even bothering to create) the value of i.
    note The above rule allows an implementation to avoid creating a fresh closure each time a local function is instantiated if it can show that the closures would behave identically. This optimisation is not transparent to the programmer because the instantiations will be === to each other and share one set of properties (including the prototype property, if applicable) rather than each having its own. ECMAScript programs should not rely on this distinction.
end if;
uf.instantiations [] instantiations [] {i};
return i
end proc;
proc instantiateMember(m: LOCALMEMBER, env: ENVIRONMENT): LOCALMEMBER
  case m of
    {forbidden} do return m;
    VARIABLE do
      note m.setup = none because Setup must have been called on a frame before that frame can be instantiated.
      value: VARIABLEVALUE m.value;
      if value : UNINSTANTIATEDFUNCTION then
        value [] instantiateFunction(value, env)
      end if;
      return new VARIABLE[]type: m.type, value: value, immutable: m.immutable, setup: none,
        initialiser: m.initialiser, initialiserEnv: env[]
    DYNAMICVAR do
      value: OBJECT[] UNINSTANTIATEDFUNCTION m.value;
      if value : UNINSTANTIATEDFUNCTION then
        value [] instantiateFunction(value, env)
      end if;
      return new DYNAMICVAR[]value: value, sealed: m.sealed[]
    GETTER do
      case m.env of
        ENVIRONMENT do return m;
        {none} do return new GETTER[]call: m.call, env: env[]
      end case;
    SETTER do
      case m.env of
        ENVIRONMENT do return m;
        {none} do return new SETTER[]call: m.call, env: env[]
      end case
    end case
  end case
end proc;

tuple MEMBERTRANSLATION
  from: LOCALMEMBER,
  to: LOCALMEMBER
end tuple;

proc instantiateLocalFrame(frame: LOCALFRAME, env: ENVIRONMENT): LOCALFRAME
  instantiatedFrame: LOCALFRAME [] new LOCALFRAME[]localBindings: {[]} pluralMembers: LOCALMEMBER{} [] {b.content | b [] frame.localBindings};
  memberTranslations: MEMBERTRANSLATION{}
    {MEMBERTRANSLATION[]from: m, to: instantiateMember(m, [instantiatedFrame] ⊕ env[]) | m [] pluralMembers};
proc translateMember(m: LOCALMEMBER): LOCALMEMBER
  mi: MEMBERTRANSLATION[] the one element mi [] memberTranslations that satisfies mi.from = m;
  return mi.to
end proc;

instantiatedFrame.localBindings [] {LOCALBINDING[]content: translateMember(b.content), other fields from b[] | b [] frame.localBindings};
return instantiatedFrame
end proc;
proc instantiateParameterFrame(frame: PARAMETERFRAME, env: ENVIRONMENT, singularThis: OBJECTOPT):
  PARAMETERFRAME

  note  frame.superconstructorCalled must be true if and only if frame.kind is not constructorFunction.
  instantiatedFrame: PARAMETERFRAME [] new PARAMETERFRAME.localBindings: [], kind: frame.kind,
  handling: frame.handling, callsSuperconstructor: frame.callsSuperconstructor,
  superconstructorCalled: frame.superconstructorCalled, this: singularThis, returnType: frame.returnType[]

  note  pluralMembers will contain the set of all LOCALMEMBER records found in the frame.
  pluralMembers: LOCALMEMBER{} [] [b.content | b[] frame.localBindings];

  note  If any of the parameters (including the rest parameter) are anonymous, their bindings will not be present in
  frame.localBindings. In this situation, the following steps add their LOCALMEMBER records to pluralMembers.
  for each p [] frame.parameters do pluralMembers [] pluralMembers [] {p.var}
  end for each;
  rest: VARIABLEOPT [] frame.rest;
  if rest = none then pluralMembers [] pluralMembers [] {rest} end if;
  memberTranslations: MEMBERTRANSLATION{} []
  {MEMBERTRANSLATION[from: m, to: instantiateMember(m, [instantiatedFrame] ⊗ env)]}
  [] m [] pluralMembers;
  proc translateMember(m: LOCALMEMBER): LOCALMEMBER
    mi: MEMBERTRANSLATION [] the one element mi [] memberTranslations that satisfies mi.from = m;
    return mi.to
  end proc;
  instantiatedFrame.localBindings [] {LOCALBINDING[b.content: translateMember(b.content), other fields from b[]]
  [] b [] frame.localBindings};
  instantiatedFrame.parameters [] {PARAMETER[op: translateMember(op.var), default: op.default[]]
  [] op [] frame.parameters};
  if rest = none then instantiatedFrame.rest [] none
  else instantiatedFrame.rest [] translateMember(rest)
  end if;
  return instantiatedFrame
end proc;

11 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.

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 OBJORREF 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, 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
```

Semantics

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

12.2 Qualified Identifiers

Syntax

```
Qualifier []
  Identifier
  | public
  | private

SimpleQualifiedIdentifier []
  Identifier
  | Qualifier :: Identifier

ExpressionQualifiedIdentifier []
  ParenExpression :: Identifier

QualifiedIdentifier []
  SimpleQualifiedIdentifier
  | ExpressionQualifiedIdentifier
```

Validation

```
OpenNamespaces[Qualifier]: NAMESPACE{};
```
proc Validate[Qualifier] (ext: CONTEXT, env: ENVIRONMENT)
[Qualifier [] Identifier] do OpenNamespaces[Qualifier] [] ext.openNamespaces;
[Qualifier [] public] do nothing;
[Qualifier [] private] do
    c: CLASSOPT getEnclosingClass(env);
    if c = none then
        throw a SyntaxError exception — private is meaningful only inside a class
    end if
end proc;

OpenNamespaces[SimpleQualifiedIdentifier]: NAMESPACE{};

proc Validate[SimpleQualifiedIdentifier] (ext: CONTEXT, env: ENVIRONMENT)
[SimpleQualifiedIdentifier [] Identifier] do
    OpenNamespaces[SimpleQualifiedIdentifier] [] ext.openNamespaces;
    Validate[Qualifier [] Identifier] do ext, env
end proc;

proc Validate[ExpressionQualifiedIdentifier [] ParenExpression : : Identifier] (ext: CONTEXT, env: ENVIRONMENT)
    Validate[ParenExpression](ext, env)
end proc;

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

Setup

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

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

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

Evaluation

proc Eval[Qualifier] (env: ENVIRONMENT, phase: PHASE): NAMESPACE
[Qualifier [] Identifier] do
    multiname: MULTINAME [] {ns:(Name[Identifier]) [] ns [] OpenNamespaces[Qualifier]};
    a: OBJECT [] lexicalRead(env, multiname, phase);
    if a [] NAMESPACE then
        throw a TypeError exception — the qualifier must be a namespace
    end if;
    return a;
[Qualifier [] public] do return public;
[Qualifier [] private] do
    c: CLASSOPT getEnclosingClass(env);
    note Validate already ensured that c ≠ none.
    return c.privateNamespace
end proc;
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
        frame: PARAMETERFRAMEOPT getEnclosingParameterFrame(env);
        if frame = none then
            if ctx.strict then
                throw a SyntaxError exception — this can be used outside a function only in non-strict mode
            end if
        elsif frame.kind = plainFunction then
            throw a SyntaxError exception — this function does not define this
        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 public;
[PrimaryExpression [] Number] do return Value[Number];
[PrimaryExpression [] String] do return Value[String];
[PrimaryExpression [] this] do
    frame: PARAMETERFRAMEOPT [] getEnclosingParameterFrame(env);
    if frame = none then return getPackageFrame(env) end if;
    note Validate ensured that frame.kind ≠ plainFunction at this point.
    this: OBJECTOPT [] frame.this;
    if this = none then
        note If Validate passed, this can be uninitialised only when phase = compile.
        throw a ConstantError exception — a constant expression cannot read an uninitialised this parameter
    end if;
    if not frame.superconstructorCalled then
        throw an UninitializedError exception — can’t access this from within a constructor before
        the superconstructor has been called
    end if;
    return this;
[PrimaryExpression [] RegularExpression] do ????;
[PrimaryExpression [] ParentListExpression] 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;

proc Eval[ParenExpression [] (AssignmentExpressionallowin)] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    return Eval[AssignmentExpressionallowin](env, phase)
end proc;


[ParenListExpression [] ParenExpression] do return Eval[ParenExpression](env, phase);
[ParenListExpression [] ParenExpression] do return Eval[ParenExpression](env, phase);
[ParenListExpression [] (ListExpressionallowin, AssignmentExpressionallowin)] do
    readReference(Eval[ListExpressionallowin, AssignmentExpressionallowin](env, phase), phase);
    return readReference(Eval[AssignmentExpressionallowin](env, phase), phase)
end proc;

proc EvalAsList[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJECT[]

[ParenListExpression [] ParenExpression] do
    elt: OBJECT [] readReference(Eval[ParenExpression](env, phase), phase);
    return [elt];
[ParenListExpression [] (ListExpressionallowin, AssignmentExpressionallowin)] do
    elts: OBJECT [] EvalAsList[ListExpressionallowin](env, phase);
    elt: OBJECT [] readReference(Eval[AssignmentExpressionallowin](env, phase), phase);
    return elts ⊕ [elt]
end proc;
12.4 Function Expressions

Syntax

\[\text{FunctionExpression} \equiv \]
\[
\quad \text{function FunctionCommon} \\
\quad | \quad \text{function Identifier FunctionCommon}
\]

Validation

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

\text{proc Validate[FunctionExpression]} (\text{ext: CONTEXT, env: ENVIRONMENT})
\[
[\text{FunctionExpression} \equiv \text{function FunctionCommon}] \text{ do}
\]
\[
\quad \text{kind: STATICFUNCTIONKind} \equiv \text{plainFunction} ;
\]
\[
\quad \text{if not ext.strict and Plain[FunctionCommon] then kind} \equiv \text{uncheckedFunction } \text{ end if;}
\]
\[
\quad \text{F[FunctionExpression]} \equiv \text{ValidateStaticFunction[FunctionCommon]}(\text{ext, env, kind}) ;
\]
\[\text{end proc;}\]

Setup

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

Evaluation

\text{proc Eval[FunctionExpression]} (\text{env: ENVIRONMENT, phase: PHASE}): OBJORREF
\[
[\text{FunctionExpression} \equiv \text{function FunctionCommon}] \text{ do}
\]
\[
\quad \text{if phase} = \text{compile} \text{ then}
\]
\[
\quad \quad \text{throw a ConstantError exception — a function expression is not a constant expression because it can evaluate to different values}
\]
\[\text{end if;}
\]
\[
\quad \text{return instantiateFunction(F[FunctionExpression], env)} ;
\]
FunctionExpression [] function Identifier FunctionCommon] do
  if phase = compile then
    throw a ConstantError exception — a function expression is not a constant expression because it can evaluate to different values
  end if;

v: VARIABLE [] new VARIABLE [type: Function, value: none, immutable: true, setup: none, initialiser: none[[]]
b: LOCALBINDING [] LOCALBINDING [name: public::(Name[Identifier]), accesses: readWrite, content: v, explicit: false, enumerable: true[]
runtimeFrame: LOCALFRAME [] new LOCALFRAME [localBindings: {b}[]]
f2: SIMPLEINSTANCE [] instantiateFunction(F[FunctionExpression], [runtimeFrame] ⊕ env);
  v.value [] f2;
return f2;
end proc;

12.5 Object Literals

Syntax

ObjectLiteral [] { FieldList }

FieldList []
  «empty»
  | NonemptyFieldList

NonemptyFieldList []
  LiteralField
  | LiteralField , NonemptyFieldList

LiteralField [] FieldName : AssignmentExpression

FieldName []
  QualifiedIdentifier
  | String
  | Number
  | ParenExpression

Validation

proc Validate[ObjectLiteral [] { FieldList }] (ext: CONTEXT, env: ENVIRONMENT)
  Validate[FieldList](ext, env)
end proc;

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

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

proc Validate[LiteralField [] FieldName : AssignmentExpression] (ext: CONTEXT, env: ENVIRONMENT)
  Validate[FieldName](ext, env);
  Validate[AssignmentExpression](ext, env)
end proc;

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

```
proc Setup[ObjectLiteral [] { FieldList }]()
    Setup[FieldList]();
end proc;
```

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

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

```
proc Setup[LiteralField [] FieldName : AssignmentExpressionallowIn]()
    Setup[FieldName]();
    Setup[AssignmentExpressionallowIn]();
end proc;
```

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

Evaluation

```
    if phase = compile then
        throw a ConstantError exception — an object literal is not a constant expression because it evaluates to a new
        object each time it is evaluated
    end if;
    o: OBJECT [] Prototype.construct([], phase);
    Eval[FieldList](env, o, phase);
    return o
end proc;
```

Eval[FieldList](env: ENVIRONMENT, o: OBJECT, phase: {run}) propagates the call to Eval to every nonterminal in the expansion of FieldList.

```
Eval[NonemptyFieldList](env: ENVIRONMENT, o: OBJECT, phase: {run}) propagates the call to Eval to every
nonterminal in the expansion of NonemptyFieldList.
```

```
proc Eval[LiteralField [] FieldName : AssignmentExpressionallowIn](env: ENVIRONMENT, o: OBJECT, phase: {run})
    multiline: MULTINAME [] Eval[FieldName](env, phase);
    value: OBJECT [] readReference(Eval[AssignmentExpressionallowIn](env, phase), phase);
    dotWrite(o, multiline, value, phase)
end proc;
```

```
proc Eval[FieldName](env: ENVIRONMENT, phase: PHASE): MULTINAME
    [FieldName [] QualifiedIdentifier] do return Eval[QualifiedIdentifier](env, phase);
    [FieldName [] String] do return {toQualifiedName(Value[String], phase)};
    [FieldName [] Number] do return {toQualifiedName(Value[Number], phase)};
    [FieldName [] ParenExpression] do
        a: OBJECT [] readReference(Eval[ParenExpression](env, phase), phase);
        return {toQualifiedName(a, phase)}
end proc;
```

12.6 Array Literals

Syntax

```
ArrayLiteral [] [ ElementList ]
```
ElementList []
  «empty»
  | LiteralElement
  | , ElementList
  | LiteralElement , ElementList

LiteralElement [] AssignmentExpression

Validation

proc Validate[ArrayLiteral [] [ ElementList ]]
  Validate[ElementList](ext, env)
end proc;

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

proc Validate[LiteralElement [] AssignmentExpression]
  Validate[AssignmentExpression](ext, env)
end proc;

Setup

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

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

proc Setup[LiteralElement [] AssignmentExpression]
  Setup[AssignmentExpression]()
end proc;

Evaluation

proc Eval[ArrayLiteral [] [ ElementList ]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  if phase = compile then
    throw a ConstantError exception — an array literal is not a constant expression because it evaluates to a new object each time it is evaluated
  end if;
  o: OBJECT [] Array.construct([], phase);
  length: INTEGER [] Eval[ElementList](env: 0, o, phase);
  if length > arrayLimit then throw a RangeError exception end if;
  dotWrite(o, {arrayPrivate::"length"}, length_ulong, phase);
  return o
end proc;

proc Eval[ElementList] (env: ENVIRONMENT, length: INTEGER, o: OBJECT, phase: {run}): INTEGER
  [ElementList [] «empty»] do return length;
  [ElementList [] LiteralElement] do
    Eval[LiteralElement](env, length, o, phase);
    return length + 1;
  [ElementList0 [] , ElementList1] do
    return Eval[ElementList1](env, length + 1, o, phase);
\[
\text{ElementList} = \text{LiteralElement}, \text{ElementList}
\]
do
Eval[\text{LiteralElement}](\text{env}, \text{length}, o, \text{phase});
return Eval[\text{ElementList}](\text{env}, \text{length} + 1, o, \text{phase})
end proc;

proc Eval[\text{LiteralElement} \ [ \text{AssignmentExpression}]]
(\text{env}: \text{ENVIRONMENT}, \text{length}: \text{INTEGER}, o: \text{OBJECT}, \text{phase}: \{\text{run}\})
value: \text{OBJECT} \ [ \text{readReference}(\text{Eval}[\text{AssignmentExpression}](\text{env}, \text{phase}), \text{phase})]
indexWrite(o, \text{length}, \text{value}, \text{phase})
end proc;

12.7 Super Expressions

Syntax

\[
\text{SuperExpression} \ [\]
\]
\text{super}
| \text{super} \text{ParenExpression}

Validation

proc Validate[\text{SuperExpression}](\text{ctx}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT})
\text{SuperExpression} \ [\text{super}]
do
c: \text{CLASSOPT} \ getEnclosingClass(\text{env});
if c = \text{none} then
  throw a \text{SyntaxError} exception — a \text{super} expression is meaningful only inside a class
end if;
frame: \text{PARAMETERFRAMEOPT} \ getEnclosingParameterFrame(\text{env});
if frame = \text{none} or frame.\text{kind} \ \text{STATICFUNCTIONKIND} then
  throw a \text{SyntaxError} exception — a \text{super} expression without an argument is meaningful only inside an
  instance method or a constructor
end if;
if c.\text{super} = \text{none} then
  throw a \text{SyntaxError} exception — a \text{super} expression is meaningful only if the enclosing class has a superclass
end if;
\text{SuperExpression} \ [\text{super} \text{ParenExpression}]
do
c: \text{CLASSOPT} \ getEnclosingClass(\text{env});
if c = \text{none} then
  throw a \text{SyntaxError} exception — a \text{super} expression is meaningful only inside a class
end if;
if c.\text{super} = \text{none} then
  throw a \text{SyntaxError} exception — a \text{super} expression is meaningful only if the enclosing class has a superclass
end if;
Validate[\text{ParenExpression}](\text{ctx}, \text{env})
end proc;

Setup

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

proc Eval(SuperExpression) (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT

[SuperExpression] super do
    frame: PARAMETERFRAMEOPT getEnclosingParameterFrame(env);
    note Validate ensured that frame ≠ none and frame.kind ≠ STATICFUNCTIONKIND at this point.
    this: OBJECTOPT frame.this;
    if this = none then
        note If Validate passed, this can be uninitialised only when phase = compile.
        throw a ConstantError exception — a constant expression cannot read an uninitialised this parameter end if;
    if not frame.superconstructorCalled then
        throw an UninitializedError exception — can’t access super from within a constructor before the superconstructor has been called end if;
    return makeLimitedInstance(this, getEnclosingClass(env), phase);
[SuperExpression] super ParenExpression do
    r: OBJORREF Eval(ParenExpression)(env, phase);
    return makeLimitedInstance(r, getEnclosingClass(env), phase)
end proc;

proc makeLimitedInstance(r: OBJORREF, c: CLASS, phase: PHASE): OBJOPTIONALLIMIT

  o: OBJECT readReference(r, phase);
  limit: CLASSOPT c.super;
  note Validate ensured that limit cannot be none at this point.
  coerced: OBJECT limit.implicitCoerce(o, false);
  if coerced = null then return null end if;
  return LIMITEDINSTANCE instance: coerced, 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] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of PostfixExpression.

Strict[AttributeExpression]: BOOLEAN;

proc Validate[AttributeExpression] (ctx: CONTEXT, env: ENVIRONMENT)
  [AttributeExpression [] SimpleQualifiedIdentifier] do
    Validate[SimpleQualifiedIdentifier](ctx, env);
    Strict[AttributeExpression] [] ctx.strict;
  [AttributeExpression0 [] AttributeExpression1 MemberOperator] do
    Validate[AttributeExpression0](ctx, env);
    Validate[MemberOperator](ctx, env);
    [AttributeExpression0 [] AttributeExpression1 Arguments] do
      Validate[AttributeExpression0](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);
  [FullPostfixExpression0 [] FullPostfixExpression1 MemberOperator] do
    Validate[FullPostfixExpression1](ctx, env);
    Validate[MemberOperator](ctx, env);
  [FullPostfixExpression [] SuperExpression MemberOperator] do
    Validate[SuperExpression](ctx, env);
    Validate[MemberOperator](ctx, env);
  [FullPostfixExpression0 [] FullPostfixExpression1 Arguments] do
    Validate[FullPostfixExpression1](ctx, env);
    Validate[Arguments](ctx, env);
  [FullPostfixExpression [] PostfixExpression] do
    Validate[PostfixExpression](ctx, env);
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);
    SuperExpression MemberOperator do
        Validate[SuperExpression](ctx, env);
        Validate[MemberOperator](ctx, env)
end proc;

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

Validate[ShortNewSubexpression] (ctx: 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
        m: MULTINAME [] Eval[SimpleQualifiedIdentifier](env, phase);
        return LEXICALREFERENCE(env, variableMultiname: m, strict: Strict)[AttributeExpression][][]
    [AttributeExpression [], AttributeExpression, MemberOperator] do
        a: OBJECT [] readReference(Eval[AttributeExpression], env, phase, phase);
        return Eval[MemberOperator](env, a, phase);
    [AttributeExpression, AttributeExpression, Arguments] do
        r: OBJORREF [] Eval[AttributeExpression](env, phase);
        f: OBJECT [] readReference(r, phase);
        base: OBJECT;
        case r of
            OBJECT [] LEXICALREFERENCE do base [] null;
            DOTREFERENCE [] BRACKETREFERENCE do base [] r.base
        end case;
        args: OBJECT[] [] 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
        m: MULTINAME [] Eval[ExpressionQualifiedIdentifier](env, phase);
        return LEXICALREFERENCE(env, variableMultiname: m, strict: Strict)[FullPostfixExpression][][]
    [FullPostfixExpression [] FullNewExpression] do
        return Eval[FullNewExpression](env, phase);
    [FullPostfixExpression, FullPostfixExpression, MemberOperator] do
        a: OBJECT [] readReference(Eval[FullPostfixExpression], env, phase, phase);
        return Eval[MemberOperator](env, a, phase);
    [FullPostfixExpression, SuperExpression MemberOperator] do
        a: OBJOPTIONALLIMIT [] Eval[SuperExpression](env, phase);
        return Eval[MemberOperator](env, a, phase);
    [FullPostfixExpression, FullPostfixExpression, Arguments] do
        r: OBJORREF [] Eval[FullPostfixExpression](env, phase);
        f: OBJECT [] readReference(r, phase);
        base: OBJECT;
        case r of
            OBJECT [] LEXICALREFERENCE do base [] null;
            DOTREFERENCE [] BRACKETREFERENCE do base [] r.base
        end case;
        args: OBJECT[] [] Eval[Arguments](env, phase);
        return call(base, f, args, phase);
[FullPostfixExpression]  PostfixExpression [no line break] ++ do
  if phase = compile then
    throw a ConstantError exception ++ cannot be used in a constant expression
  end if;
  r: ObjectRef [] 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 b;
[FullPostfixExpression]  PostfixExpression [no line break] -- do
  if phase = compile then
    throw a ConstantError exception -- cannot be used in a constant expression
  end if;
  r: ObjectRef [] Eval[PostfixExpression](env, phase);
  a: Object [] readReference(r, phase);
  b: Object [] plus(a, phase);
  c: Object [] subtract(b, 1.0_64, phase);
  writeReference(r, c, phase);
  return b;
end proc;

proc Eval[FullNewExpression [] new FullNewSubexpression Arguments](env: Environment, phase: Phase): ObjectRef
  f: Object [] readReference(Eval[FullNewSubexpression](env, phase), phase);
  args: Object [] Eval[Arguments](env, phase);
  return construct(f, args, phase)
end proc;

proc Eval[FullNewSubexpression](env: Environment, phase: Phase): ObjectRef
  [FullNewSubexpression [] PrimaryExpression] do
    return Eval[PrimaryExpression](env, phase);
  [FullNewSubexpression [] QualifiedIdentifier] do
    m: Multiname [] Eval[QualifiedIdentifier](env, phase);
    return LexicalReference[env: env, variableMultiname: m, strict: Strict][FullNewSubexpression]
  [FullNewSubexpression [] FullNewExpression] do
    return Eval[FullNewExpression](env, phase);
  [FullNewSubexpression, MemberOperator] do
    a: Object [] readReference(Eval[FullNewSubexpression], env, phase), phase);
    return Eval[MemberOperator](env, a, phase);
  [FullNewSubexpression [] SuperExpression MemberOperator] do
    a: Object [] Eval[SuperExpression](env, phase);
    return Eval[MemberOperator](env, a, phase)
end proc;

proc Eval[ShortNewExpression [] new ShortNewSubexpression](env: Environment, phase: Phase): ObjectRef
  f: Object [] readReference(Eval[ShortNewSubexpression](env, phase), phase);
  return construct(f, [], phase)
end proc;

proc Eval[ShortNewSubexpression](env: Environment, phase: Phase): ObjectRef
  [ShortNewSubexpression [] FullNewSubexpression] do
    return Eval[FullNewSubexpression](env, phase);
[ShortNewSubexpression [] ShortNewExpression] do
  return Eval[ShortNewExpression](env, phase)
end proc;

proc call(this: Object, a: Object, args: Object[], phase: Phase): Object
  case a of
    Undefined [] Null [] Boolean [] GeneralNumber [] Character [] String [] Namespace []
    CompoundAttribute [] Date [] RegExp [] Package do
      throw a TypeError exception;
    end case
    Class do return a.call(this, args, phase);
    SimpleInstance do
      f: (Object [] SimpleInstance [] Object[] [] Phase [] Object) [] {none} [] a.call;
      if f = none then throw a TypeError exception end if;
      return f(this, a, args, phase);
    end case
    MethodClosure do
      m: InstanceMethod [] a.method;
      return m.call(a.this, args, phase)
    end case
  end case
end proc;

proc construct(a: Object, args: Object[], phase: Phase): Object
  case a of
    Undefined [] Null [] Boolean [] GeneralNumber [] Character [] String [] Namespace []
    CompoundAttribute [] MethodClosure [] Date [] RegExp [] Package do
      throw a TypeError exception;
    end case
    Class do return a.construct(args, phase);
    SimpleInstance do
      f: (SimpleInstance [] Object[] [] Phase [] Object) [] {none} [] a.construct;
      if f = none then throw a TypeError exception end if;
      return f(a, args, phase)
    end case
  end case
end proc;

12.9 Member Operators

Syntax

MemberOperator []
  . QualifiedIdentifier
  | Brackets

Brackets []
  [ ]
  | [ ListExpression Allow ]
  | [ ExpressionsWithRest ]

Arguments []
  ( )
  | ParenListExpression
  | ( ExpressionsWithRest )

ExpressionsWithRest []
  RestExpression
  | ListExpression Allow , RestExpression

RestExpression [] . . . AssignmentExpression Allow


Validation

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

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

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

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

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

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

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

Evaluation

    [MemberOperator . QualifiedIdentifier] do
        m: MULTINAME ⇒ Eval[QualifiedIdentifier](env, phase);
        case base of
            OBJECT do
                return DOTREFERENCE[base, limit: objectType(base), propertyMultiname: m[]]
            LIMITEDINSTANCE do
                return DOTREFERENCE[base: base . instance, limit: base . limit, propertyMultiname: m[]]
            end case;
    [MemberOperator Brackets] do
        args: OBJECT[] ⇒ Eval[Brackets](env, phase);
        case base of
            OBJECT do
                return BRACKETREFERENCE[base, limit: objectType(base), args: args[]]
            LIMITEDINSTANCE do
                return BRACKETREFERENCE[base: base . instance, limit: base . limit, args: args[]]
            end case
        end case
    end proc;

proc Eval[Brackets] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
    [Brackets [ ] ] do return [];
    [Brackets [ ListExpression]] do
        return EvalAsList[ListExpression](env, phase);
Brackets do return Eval[ExpressionsWithRest](env, phase)
end proc;

proc Eval[Arguments] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
Arguments ( ) do return [];
Arguments ParenListExpression do
  return EvalAsList[ParenListExpression](env, phase);
Arguments ( ExpressionsWithRest ) do
  return Eval[ExpressionsWithRest](env, phase)
end proc;

proc Eval[ExpressionsWithRest] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
ExpressionsWithRest RestExpression do return Eval[RestExpression](env, phase);
ExpressionsWithRest ListExpressionallowIn, RestExpression do
  args1: OBJECT[] EvalAsList[ListExpressionallowIn](env, phase);
  args2: OBJECT[] Eval[RestExpression](env, phase);
  return args1 ⊕ args2
end proc;

proc Eval[RestExpression ... AssignmentExpressionallowIn] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
a: OBJECT readReference(Eval[AssignmentExpressionallowIn](env, phase), phase);
if not Array.is(a) then throw a TypeError exception — the ... operand must be an Array
end if;
length: ULONG readInstanceProperty(a, arrayPrivate::"length", phase);
i: INTEGER 0;
args: OBJECT[] [];
while i ≠ length.value do
  arg: OBJECTOPT indexRead(a, i, phase);
  if arg = none
    An implementation may, at its discretion, either throw a ReferenceError or treat the hole as a missing argument, substituting the called function’s default parameter value if there is one, undefined if the called function is unchecked, or throwing an ArgumentError exception otherwise. An implementation must not replace such a hole with undefined except when the called function is unchecked or happens to have undefined as its default parameter value.
  end if;
  args ⊕ arg, i + 1
end while;
return args
end proc;
12.10 Unary Operators

Syntax

\[
\text{UnaryExpression} \equiv \\
\text{PostfixExpression} \mid \text{delete PostfixExpression} \mid \text{void UnaryExpression} \mid \text{typeof UnaryExpression} \mid ++ \text{ PostfixExpression} \mid -- \text{ PostfixExpression} \mid + \text{ UnaryExpression} \mid - \text{ UnaryExpression} \mid -\text{ NegatedMinLong} \mid \sim \text{ UnaryExpression} \mid ! \text{ UnaryExpression}
\]

Validation

\[
\text{Strict[UnaryExpression]} : \text{BOOLEAN};
\]

\[
\text{proc Validate[UnaryExpression] (cxt : CONTEXT, env : ENVIRONMENT)} \equiv \\
\text{[UnaryExpression] do Validate[PostfixExpression] (cxt, env);} \\
\text{[UnaryExpression] delete PostfixExpression do Validate[PostfixExpression] (cxt, env);} \\
\text{Strict[UnaryExpression] = cxt.strict;} \\
\text{[UnaryExpression] void UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{[UnaryExpression] typeof UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{[UnaryExpression] ++ PostfixExpression do Validate[PostfixExpression] (cxt, env);} \\
\text{[UnaryExpression] -- PostfixExpression do Validate[PostfixExpression] (cxt, env);} \\
\text{[UnaryExpression] + UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{[UnaryExpression] - UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{[UnaryExpression] - NegatedMinLong do nothing;} \\
\text{[UnaryExpression] ~ UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{[UnaryExpression] ! UnaryExpression do Validate[UnaryExpression] (cxt, env);} \\
\text{end proc;}
\]

Setup

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

Evaluation

\[
\text{proc Eval[UnaryExpression] (env : ENVIRONMENT, phase : PHASE) : OBJORREF} \equiv \\
\text{[UnaryExpression] PostfixExpression do return Eval[PostfixExpression] (env, phase);} \\
\text{[UnaryExpression] delete PostfixExpression do if phase = compile then} \\
\text{throw a ConstantError exception --- delete cannot be used in a constant expression} \\
\text{end if;}
\]

\[
r : OBJORREF \equiv \text{ Eval[PostfixExpression] (env, phase);} \\
\text{return deleteReference}(r, \text{Strict[UnaryExpression], phase});
\]
[UnaryExpression, void UnaryExpression] do
  readReference(Eval(UnaryExpression, env, phase), phase);
  return undefined;
[UnaryExpression, typeof UnaryExpression] do
  a: OBJECT readReference(Eval(UnaryExpression, env, phase), phase);
  c: CLASS objectType(a);
  return c.typeOfString;
[UnaryExpression, ++ PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — ++ cannot be used in a constant expression
  end if;
  r: OBJORREF Eval(PostfixExpression, env, phase);
  a: OBJECT readReference(r, phase);
  b: OBJECT plus(a, phase);
  c: OBJECT add(b, 1.0f64, phase);
  writeReference(r, c, phase);
  return c;
[UnaryExpression, -- PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — -- cannot be used in a constant expression
  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
  a: OBJECT readReference(Eval(UnaryExpression, env, phase), phase);
  return plus(a, phase);
[UnaryExpression, - UnaryExpression] do
  a: OBJECT readReference(Eval(UnaryExpression, env, phase), phase);
  return minus(a, phase);
[UnaryExpression, - NegatedMinLong] do return (-2^63)_long;
[UnaryExpression, ~ UnaryExpression] do
  a: OBJECT readReference(Eval(UnaryExpression, env, phase), phase);
  return bitNot(a, phase);
[UnaryExpression, ! UnaryExpression] do
  a: OBJECT readReference(Eval(UnaryExpression, env, phase), 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 □ toGeneralNumber(a, phase);
  case x of
    LONG do i: {-2^63 ... 2^63 - 1} □ x.value; return bitwiseXor(i, -1)_long;
    ULONG do
      i: {0 ... 2^64 - 1} □ x.value;
      return bitwiseXor(i, 0xFFFFFFFFFFFFFFF)_long;
    FLOAT32 □ FLOAT64 do
      i: {-2^31 ... 2^31 - 1} □ signedWrap32(truncateToInteger(x));
      return realToFloat64(bitwiseXor(i, -1))
  end case
end proc;

logicalNot(a, phase) returns the value of the unary expression !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

MultiplicativeExpression □
  UnaryExpression
  | MultiplicativeExpression * UnaryExpression
  | MultiplicativeExpression / UnaryExpression
  | MultiplicativeExpression % UnaryExpression

Validation

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

Setup

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

Evaluation

  [MultiplicativeExpression □ UnaryExpression] do
    return Eval[UnaryExpression](env, phase);
[MultiplicativeExpression, MultiplicativeExpression, UnaryExpression] do
  a: OBJECT readReference(Eval[MultiplicativeExpression, env, phase], phase);
  b: OBJECT readReference(Eval[UnaryExpression, env, phase], phase);
  return multiply(a, b, phase);

[MultiplicativeExpression, MultiplicativeExpression, UnaryExpression] do
  a: OBJECT readReference(Eval[MultiplicativeExpression, env, phase], phase);
  b: OBJECT readReference(Eval[UnaryExpression, env, phase], phase);
  return divide(a, b, phase);

[MultiplicativeExpression, MultiplicativeExpression, UnaryExpression] do
  a: OBJECT readReference(Eval[MultiplicativeExpression, env, phase], phase);
  b: OBJECT readReference(Eval[UnaryExpression, env, phase], phase);
  return remainder(a, b, phase)
end proc;

proc multiply(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER toGeneralNumber(a, phase);
  y: GENERALNUMBER toGeneralNumber(b, phase);
  if x ‹ LONG or y ‹ LONG then
    i: INTEGEROPT checkInteger(x);
    j: INTEGEROPT checkInteger(y);
    if i ≠ none and j ≠ none then
      k: INTEGER while y = LONG or y = ULONG then
        integerToULong(k)
      end if
    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 or y ‹ LONG then
    i: INTEGEROPT checkInteger(x);
    j: INTEGEROPT checkInteger(y);
    if i ≠ none and j ≠ none and j ≠ 0 then
      q: RATIONAL rationalToULong(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 i / j;
      k: Integer q ≥ 0 ? [k] : [0];
      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] (ctx: 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

proc Eval[AdditiveExpression] (env: ENVIRONMENT, phase: Phase): ObjOrRef
  [AdditiveExpression □ MultiplicativeExpression] do
    return Eval[MultiplicativeExpression](env, phase);
  end Eval[AdditiveExpression]

  [AdditiveExpression0 □ AdditiveExpression1 + MultiplicativeExpression] do
    a: Object readReference(Eval[AdditiveExpression0](env, phase), phase);
    b: Object readReference(Eval[MultiplicativeExpression](env, phase), phase);
    return add(a, b, phase);
  end Eval[AdditiveExpression]

  [AdditiveExpression0 □ AdditiveExpression1 - MultiplicativeExpression] do
    a: Object readReference(Eval[AdditiveExpression0](env, phase), phase);
    b: Object readReference(Eval[MultiplicativeExpression](env, phase), phase);
    return subtract(a, b, phase)
  end Eval[AdditiveExpression]
proc add(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
ap: PRIMITIVEOBJECT ▶ toPrimitive(a, null, phase);
bp: PRIMITIVEOBJECT ▶ toPrimitive(b, null, phase);
if ap ▶ STRING or bp ▶ 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
return float64Add(toFloat64(x), toFloat64(y))
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
return float64Subtract(toFloat64(x), toFloat64(y))
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
    a: OBJECT  readReference(Eval[ShiftExpression1](env, phase), phase);
    b: OBJECT  readReference(Eval[AdditiveExpression](env, phase), phase);
    return shiftLeft(a, b, phase);
  [ShiftExpression0  ShiftExpression1 >> AdditiveExpression] do
    a: OBJECT  readReference(Eval[ShiftExpression1](env, phase), phase);
    b: OBJECT  readReference(Eval[AdditiveExpression](env, phase), phase);
    return shiftRight(a, b, phase);
  [ShiftExpression0  ShiftExpression1 >>> AdditiveExpression] do
    a: OBJECT  readReference(Eval[ShiftExpression1](env, phase), phase);
    b: OBJECT  readReference(Eval[AdditiveExpression](env, phase), 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} \ldots 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} \ldots 2^{63} – 1\}  \[ signedWrap64(bitwiseShift(x.value, count));
        return i_long;
    ULONG do
      count  bitwiseAnd(count, 0x3F);
      i: {0 ... 2^{64} – 1}  \[ unsignedWrap64(bitwiseShift(x.value, count));
        return i_ulong
    end case
  end proc;
proc shiftRight(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 ▣ bitwiseShift(i, –count);
return realToFloat64(i);
LONG do
  count ▣ bitwiseAnd(count, 0x3F);
i: {-2^{63} ... 2^{63} – 1} ▣ bitwiseShift(x.value, –count);
return i.long;
ULONG do
  count ▣ bitwiseAnd(count, 0x3F);
i: {-2^{64} ... 2^{64} – 1} ▣ bitwiseShift(signedWrap64(x.value), –count);
return (unsignedWrap64(i)).ulong
end case
end proc;

proc shiftRightUnsigned(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: {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 (signedWrap64(i)).long;
ULONG do
  count ▣ bitwiseAnd(count, 0x3F);
i: {0 ... 2^{64} – 1} ▣ bitwiseShift(x.value, –count);
return i.ulong
end case
end proc;

12.14 Relational Operators

Syntax

RelationalExpression | ShiftExpression
--------------------|---------------------
RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression
| RelationalExpression | ShiftExpression

instanceof
RelationalExpression\textsuperscript{nonin} \[\]
ShiftExpression
| RelationalExpression\textsuperscript{nonin} < ShiftExpression
| RelationalExpression\textsuperscript{nonin} > ShiftExpression
| RelationalExpression\textsuperscript{nonin} <= ShiftExpression
| RelationalExpression\textsuperscript{nonin} >= ShiftExpression
| RelationalExpression\textsuperscript{nonin} as ShiftExpression
| RelationalExpression\textsuperscript{nonin} instanceof ShiftExpression

Validation

\texttt{Validate}[\texttt{RelationalExpression}^{\textit{fr}}] (\textit{ext}: \texttt{CONTEXT}, \textit{env}: \texttt{ENVIRONMENT}) propagates the call to \texttt{Validate} to every nonterminal in the expansion of \texttt{RelationalExpression}^{\textit{fr}}.

Setup

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

Evaluation

\begin{verbatim}
proc \texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}] (\textit{env}: \texttt{ENVIRONMENT}, \textit{phase}: \texttt{PHASE}): \texttt{OBJORREF}
[\texttt{RelationalExpression}^{\textit{fr}} \[\]
\texttt{ShiftExpression}] do
  return \texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase});
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 < \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  return \texttt{isLess}(a, b, \textit{phase});
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 > \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  return \texttt{isLess}(b, a, \textit{phase});
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 <= \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  return \texttt{isLessOrEqual}(a, b, \textit{phase});
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 >= \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  return \texttt{isLessOrEqual}(b, a, \textit{phase});
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 \texttt{is} \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  c: \texttt{CLASS} \[\]
  \texttt{toClass}(b);
  return \texttt{c.is}(a);
[\texttt{RelationalExpression}^{\textit{fr}}_0 \[\]
\texttt{RelationalExpression}^{\textit{fr}}_1 \texttt{as} \texttt{ShiftExpression}] do
  a: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{RelationalExpression}^{\textit{fr}}_1](\textit{env}, \textit{phase}), \textit{phase});
  b: \texttt{OBJECT} \[\]
  \texttt{readReference}(\texttt{Eval}[\texttt{ShiftExpression}](\textit{env}, \textit{phase}), \textit{phase});
  c: \texttt{CLASS} \[\]
  \texttt{toClass}(b);
  return \texttt{c.implicitCoerce}(a, \texttt{true});
\end{verbatim}
12.15 Equality Operators

Syntax

\[
\text{EqualityExpression}^{(5)} \mid \text{RelationalExpression}^{(3)} \\
\| \text{EqualityExpression}^{(3)} == \text{RelationalExpression}^{(3)} \\
\| \text{EqualityExpression}^{(3)} != \text{RelationalExpression}^{(3)} \\
\| \text{EqualityExpression}^{(3)} \&\& \text{RelationalExpression}^{(3)} \\
\| \text{EqualityExpression}^{(3)} \|\| \text{RelationalExpression}^{(3)}
\]

Validation

\text{Validate}[\text{EqualityExpression}^{(5)}](\text{ctx: CONTEXT, env: ENVIRONMENT}) propagates the call to \text{Validate} to every nonterminal in the expansion of \text{EqualityExpression}^{(5)}.

Setup

\text{Setup}[\text{EqualityExpression}^{(5)}]() propagates the call to \text{Setup} to every nonterminal in the expansion of \text{EqualityExpression}^{(5)}. 

[RelationalExpression]^{(5)} \| \text{RelationalExpression}^{(3)} \text{ in ShiftExpression} \text{ do}
\begin{align*}
a: \text{OBJECT} & \rightarrow \text{readReference}(\text{Eval}[\text{RelationalExpression}^{(5)}](\text{env, phase}, \text{phase})); \\
b: \text{OBJECT} & \rightarrow \text{readReference}(\text{Eval}[\text{ShiftExpression}](\text{env, phase}, \text{phase})); \\
qname: \text{QUALIFIEDNAME} & \rightarrow \text{toQualifiedName}(a, \text{phase}); \\
c: \text{CLASS} & \rightarrow \text{objectType}(b); \\
\text{return} \text{findBaseInstanceMember}(c, \{qname\}, \text{read} \neq \text{null} \text{ or} \\
\text{findBaseInstanceMember}(c, \{qname\}, \text{write} \neq \text{null} \text{ or} \\
\text{findCommonMember}(b, \{qname\}, \text{read, false} \neq \text{null} \text{ or} \\
\text{findCommonMember}(b, \{qname\}, \text{write, false} \neq \text{null});
\end{align*}

[RelationalExpression]^{(5)} \| \text{RelationalExpression}^{(3)}, \text{instanceof ShiftExpression} \text{ do}
\begin{align*}
a: \text{OBJECT} & \rightarrow \text{readReference}(\text{Eval}[\text{RelationalExpression}^{(5)}](\text{env, phase}, \text{phase})); \\
b: \text{OBJECT} & \rightarrow \text{readReference}(\text{Eval}[\text{ShiftExpression}](\text{env, phase}, \text{phase})); \\
\text{if not} \text{PrototypeFunction}.\text{is}(b) & \text{ then throw a TypeError exception end if;}
\text{prototype: OBJECT} & \rightarrow \text{dotRead}(b, \{\text{public::"prototype"}, \text{phase})); \\
\text{return} \text{prototype} \rightarrow \text{objectSupers}(a)
\end{align*}
end proc;

\text{proc} \text{isLess}(a: \text{OBJECT, b: OBJECT, phase: PHASE}): \text{BOOLEAN}
\begin{align*}
ap: \text{PRIMITIVEOBJECT} & \rightarrow \text{toPrimitive}(a, \text{null, phase}); \\
bp: \text{PRIMITIVEOBJECT} & \rightarrow \text{toPrimitive}(b, \text{null, phase}); \\
\text{if} \ ap \rightarrow \text{STRING} \text{ and } bp \rightarrow \text{STRING} \text{ then} \\
\text{return} \text{toString}(ap, \text{phase}) < \text{toString}(bp, \text{phase})
\end{align*}
end proc;

\text{proc} \text{isLessOrEqual}(a: \text{OBJECT, b: OBJECT, phase: PHASE}): \text{BOOLEAN}
\begin{align*}
ap: \text{PRIMITIVEOBJECT} & \rightarrow \text{toPrimitive}(a, \text{null, phase}); \\
bp: \text{PRIMITIVEOBJECT} & \rightarrow \text{toPrimitive}(b, \text{null, phase}); \\
\text{if} \ ap \rightarrow \text{STRING} \text{ and } bp \rightarrow \text{STRING} \text{ then} \\
\text{return} \text{toString}(ap, \text{phase}) \leq \text{toString}(bp, \text{phase})
\end{align*}
end proc;
Proc Eval[EqualityExpression^0] (env: ENVIRONMENT, phase: PHASE): OBJORREF

EqualityExpression^0 reflexivity

proc Eval[EqualityExpression^0] (env: ENVIRONMENT, phase: PHASE): OBJORREF

EqualityExpression^0 reflexivity

return Eval[RelationalExpression^0](env, phase);

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

do

EqualityExpression^0 reflexivity

EqualityExpression^0 reflexive

end proc;
proc isEqual(a: Object, b: Object, phase: Phase): Boolean
    case a of
      Undefined | Null do return b | Undefined | Null;
      Boolean do
        if b | Boolean then return a = b
        else return isEqual(toGeneralNumber(a, phase), b, phase)
      end if;
      GeneralNumber do
        bp: PrimitiveObject | toPrimitive(b, null, phase);
        case bp of
          Undefined | Null do return false;
          Boolean | GeneralNumber do
            return generalNumberCompare(toGeneralNumber(a, phase), toGeneralNumber(bp, phase)) = equal
          end case;
          Character | String do
            return generalNumberCompare(toGeneralNumber(a, phase), toGeneralNumber(bp, phase)) = equal
          end case;
        end case
      end case
    end case
end proc;

proc isStrictlyEqual(a: Object, b: Object, phase: Phase): Boolean
    if a | GeneralNumber and b | GeneralNumber then
      return generalNumberCompare(a, b) = equal
    else return a = b
  end if
end proc;

12.16 Binary Bitwise Operators

Syntax

BitwiseAndExpression | EqualityExpression | BitwiseAndExpression
| BitwiseXorExpression | BitwiseAndExpression

BitwiseXorExpression | BitwiseAndExpression
| BitwiseXorExpression

ECMAScript Edition 4 Draft
BitwiseOrExpression[^1] []
BitwiseXorExpression[^1]
| BitwiseOrExpression[^1] | BitwiseXorExpression[^1]

Validation

Validate[BitwiseAndExpression[^2]] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of BitwiseAndExpression[^1].

Validate[BitwiseXorExpression[^2]] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of BitwiseXorExpression[^1].

Validate[BitwiseOrExpression[^2]] (ctx: CONTEXT, env: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of BitwiseOrExpression[^1].

Setup

Setup[BitwiseAndExpression[^3]] () propagates the call to Setup to every nonterminal in the expansion of BitwiseAndExpression[^1].

Setup[BitwiseXorExpression[^3]] () propagates the call to Setup to every nonterminal in the expansion of BitwiseXorExpression[^1].

Setup[BitwiseOrExpression[^3]] () propagates the call to Setup to every nonterminal in the expansion of BitwiseOrExpression[^1].

Evaluation

proc Eval[BitwiseAndExpression[^4]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[BitwiseAndExpression[^1] [] EqualityExpression[^2]] do
  return Eval[EqualityExpression[^3]](env, phase);
[BitwiseAndExpression[^1] [] BitwiseAndExpression[^1], & EqualityExpression[^1]] do
  a: OBJECT [] readReference(Eval[BitwiseAndExpression[^1]])(env, phase), phase);
  b: OBJECT [] readReference(Eval[EqualityExpression[^1]])(env, phase), phase);
  return bitAnd(a, b, phase)
end proc;

proc Eval[BitwiseXorExpression[^4]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[BitwiseXorExpression[^1] [] BitwiseAndExpression[^1]] do
  return Eval[BitwiseAndExpression[^3]](env, phase);
[BitwiseXorExpression[^1] [] BitwiseXorExpression[^1], ^ BitwiseAndExpression[^1]] do
  a: OBJECT [] readReference(Eval[BitwiseXorExpression[^1]])(env, phase), phase);
  b: OBJECT [] readReference(Eval[BitwiseAndExpression[^1]])(env, phase), phase);
  return bitXor(a, b, phase)
end proc;

proc Eval[BitwiseOrExpression[^4]] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[BitwiseOrExpression[^1] [] BitwiseXorExpression[^1]] do
  return Eval[BitwiseXorExpression[^3]](env, phase);
[BitwiseOrExpression[^1] [] BitwiseOrExpression[^1], | BitwiseXorExpression[^1]] do
  a: OBJECT [] readReference(Eval[BitwiseOrExpression[^1]])(env, phase), phase);
  b: OBJECT [] readReference(Eval[BitwiseOrExpression[^1]])(env, phase), 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} \to signedWrap64(truncateToInteger(x));
  j: {-2^{63} ... 2^{63} - 1} \to signedWrap64(truncateToInteger(y));
  k: {-2^{63} ... 2^{63} - 1} \to bitwiseAnd(i, j);
else return k_{long}
end if
else
  i: {-2^{31} ... 2^{31} - 1} \to signedWrap32(truncateToInteger(x));
  j: {-2^{31} ... 2^{31} - 1} \to 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} \to signedWrap64(truncateToInteger(x));
  j: {-2^{63} ... 2^{63} - 1} \to signedWrap64(truncateToInteger(y));
  k: {-2^{63} ... 2^{63} - 1} \to bitwiseXor(i, j);
else return k_{long}
end if
else
  i: {-2^{31} ... 2^{31} - 1} \to signedWrap32(truncateToInteger(x));
  j: {-2^{31} ... 2^{31} - 1} \to 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} \to signedWrap64(truncateToInteger(x));
  j: {-2^{63} ... 2^{63} - 1} \to signedWrap64(truncateToInteger(y));
  k: {-2^{63} ... 2^{63} - 1} \to bitwiseOr(i, j);
else return k_{long}
end if
else
  i: {-2^{31} ... 2^{31} - 1} \to signedWrap32(truncateToInteger(x));
  j: {-2^{31} ... 2^{31} - 1} \to 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[] BitwiseOrExpression] do
  return Eval[BitwiseOrExpression](env, phase);
[LogicalAndExpression, & BitwiseOrExpression] do
  a: OBJECT[] readReference(Eval[LogicalAndExpression](env, phase), phase);
  if toBoolean(a, phase) then
    return readReference(Eval[BitwiseOrExpression](env, phase), phase)
  else return a
end if
end proc;

proc Eval[LogicalXorExpression](env: ENVIRONMENT, phase: PHASE): OBJORREF
[LogicalXorExpression[] LogicalAndExpression] do
  return Eval[LogicalAndExpression](env, phase);
[LogicalXorExpression[^ LogicalAndExpression]] do
  a: OBJECT readReference(Eval[LogicalXorExpression](env, phase), phase);
  b: OBJECT readReference(Eval[LogicalAndExpression](env, phase), 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[^ LogicalXorExpression]] do
    return Eval[LogicalXorExpression](env, phase);
  [LogicalOrExpression[^ LogicalAndExpression]] do
    a: OBJECT readReference(Eval[LogicalOrExpression],(env, phase), phase);
    if toBoolean(a, phase) then return a
    else return readReference(Eval[LogicalXorExpression](env, phase), phase)
  end if
end proc;

12.18 Conditional Operator

Syntax

ConditionalExpression[^]
  LogicalOrExpression[^]
  | LogicalOrExpression? AssignmentExpression[^] : AssignmentExpression[^]

NonAssignmentExpression[^]
  LogicalOrExpression[^]
  | LogicalOrExpression? 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] () propagates the call to Setup to every nonterminal in the expansion of ConditionalExpression[^].

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

Evaluation

  [ConditionalExpression[^ LogicalOrExpression]] do
    return Eval[LogicalOrExpression](env, phase);
[ConditionalExpression[^ ConditionalOrExpression[^ ? AssignmentExpression[^ ] : AssignmentExpression[^ ]] do
  a: OBJECT[^ readReference(Eval[ConditionalOrExpression[^ ]](env, phase), phase);
  if toBoolean(a, phase) then
    return readReference(Eval[AssignmentExpression[^ ]](env, phase), phase)
  else return readReference(Eval[AssignmentExpression[^ ]](env, phase), phase)
end if
end proc;

proc Eval[NonAssignmentExpression[^ ]](env: ENVIRONMENT, phase: PHASE): OBJORREF
  Eval[ConditionalOrExpression[^ ] do
    a: OBJECT[^ readReference(Eval[ConditionalOrExpression[^ ]](env, phase), phase);
    if toBoolean(a, phase) then
      return readReference(Eval[NonAssignmentExpression[^ ]](env, phase), phase)
    else return readReference(Eval[NonAssignmentExpression[^ ]](env, phase), phase)
end if
end proc;

12.19 Assignment Operators

Syntax

AssignmentExpression[^ ConditionalExpression[^ ]
  | PostfixExpression = AssignmentExpression[^ ]
  | PostfixExpression CompoundAssignment AssignmentExpression[^ ]
  | PostfixExpression LogicalAssignment AssignmentExpression[^ ]

CompoundAssignment[^ ]
  *=
  /=
  %=
  +=
  -=
  <<=
  >>=
  >>>=
  &=
  ^=
  |=

LogicalAssignment[^ ]
  & &=
  ^ ^=
  | |=

Semantics

tag andEq;
tag xorEq;
tag orEq;
Validation

\[
\text{proc } \text{Validate}[\text{AssignmentExpression}^b] (\text{cntx}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT}) \\
\text{[AssignmentExpression}^b \rightarrow \text{ConditionalExpression}^b] \text{ do} \\
\text{Validate}[\text{ConditionalExpression}^b](\text{cntx}, \text{env}); \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression} = \text{AssignmentExpression}^b_1] \text{ do} \\
\text{Validate}[\text{PostfixExpression}](\text{cntx}, \text{env}); \\
\text{Validate}[\text{AssignmentExpression}^b_1](\text{cntx}, \text{env}); \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression CompoundAssignment AssignmentExpression}^b_1] \text{ do} \\
\text{Validate}[\text{PostfixExpression}](\text{cntx}, \text{env}); \\
\text{Validate}[\text{AssignmentExpression}^b_1](\text{cntx}, \text{env}); \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression LogicalAssignment AssignmentExpression}^b_1] \text{ do} \\
\text{Validate}[\text{PostfixExpression}](\text{cntx}, \text{env}); \\
\text{Validate}[\text{AssignmentExpression}^b_1](\text{cntx}, \text{env})
\text{ end proc;}
\]

Setup

\[
\text{proc } \text{Setup}[\text{AssignmentExpression}^b] () \\
\text{[AssignmentExpression}^b \rightarrow \text{ConditionalExpression}^b] \text{ do Setup}[\text{ConditionalExpression}^b](); \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression} = \text{AssignmentExpression}^b_1] \text{ do} \\
\text{Setup}[\text{PostfixExpression}]() ; \\
\text{Setup}[\text{AssignmentExpression}^b_1]() ; \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression CompoundAssignment AssignmentExpression}^b_1] \text{ do} \\
\text{Setup}[\text{PostfixExpression}]() ; \\
\text{Setup}[\text{AssignmentExpression}^b_1]() ; \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression LogicalAssignment AssignmentExpression}^b_1] \text{ do} \\
\text{Setup}[\text{PostfixExpression}]() ; \\
\text{Setup}[\text{AssignmentExpression}^b_1]()
\text{ end proc;}
\]

Evaluation

\[
\text{proc } \text{Eval}[\text{AssignmentExpression}^b] (\text{env}: \text{ENVIRONMENT}, \text{phase}: \text{PHASE}): \text{OBJOrREF} \\
\text{[AssignmentExpression}^b \rightarrow \text{ConditionalExpression}^b] \text{ do} \\
\text{return } \text{Eval}[\text{ConditionalExpression}^b](\text{env}, \text{phase}); \\
\text{[AssignmentExpression}^b_0 \rightarrow \text{PostfixExpression} = \text{AssignmentExpression}^b_1] \text{ do} \\
\text{if } \text{phase} = \text{compile} \text{ then} \\
\text{throw a } \text{ConstantError} \text{ exception — assignment cannot be used in a constant expression} \\
\text{end if;} \\
\text{ra: OBJOrREF} \rightarrow \text{Eval}[\text{PostfixExpression}](\text{env}, \text{phase}); \\
b: \text{OBJECT} \rightarrow \text{readReference}(\text{Eval}[\text{AssignmentExpression}^b_1](\text{env}, \text{phase}, \text{phase}); \\
\text{writeReference}(\text{ra}, b, \text{phase}); \\
\text{return } b;
\]
AssignmentExpression PostfixExpression CompoundAssignment AssignmentExpression

if phase = compile then
  throw a ConstantError exception — assignment cannot be used in a constant expression
end if;

rLeft: OBJORRef Eval[PostfixExpression](env, phase);

oLeft: OBJECT readReference(rLeft, phase);

oRight: OBJECT readReference(Eval[AssignmentExpression](env, phase), phase);

result: OBJECT Op[CompoundAssignment](oLeft, oRight, phase);

writeReference(rLeft, result, phase);

return result;

AssignmentExpression PostfixExpression LogicalAssignment AssignmentExpression

if phase = compile then
  throw a ConstantError exception — assignment cannot be used in a constant expression
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](env, phase), phase)
    end if;
  {xorEq} do
    bRight: BOOLEAN toBoolean(readReference(Eval[AssignmentExpression](env, phase), phase), phase);
    result bLeft xor bRight;
  {orEq} do
    if not bLeft then
      result readReference(Eval[AssignmentExpression](env, phase), phase)
    end if;
  end case;

writeReference(rLeft, result, phase);

return result
end proc;

Op[CompoundAssignment]: OBJECT OBJECT PHASE OBJECT;

Op[CompoundAssignment *] = multiply;
Op[CompoundAssignment /] = divide;
Op[CompoundAssignment %] = remainder;
Op[CompoundAssignment +] = add;
Op[CompoundAssignment -] = subtract;
Op[CompoundAssignment <<] = shiftLeft;
Op[CompoundAssignment >>] = shiftRight;
Op[CompoundAssignment >>>] = shiftRightUnsigned;
Op[CompoundAssignment &] = bitAnd;
Op[CompoundAssignment ^] = bitXor;
Op[CompoundAssignment |] = bitOr;

Operator[LogicalAssignment]: {andEq, xorEq, orEq};

Operator[LogicalAssignment &] = andEq;
Operator[LogicalAssignment ^] = xorEq;
Operator[LogicalAssignment |] = orEq;
12.20 Comma Expressions

Syntax

\[ \text{ListExpression} \equiv \text{AssignmentExpression} | \text{ListExpression}, \text{AssignmentExpression} \]

Validation

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

Setup

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

Evaluation

\[ \text{proc Eval} [\text{ListExpression}] (\text{env}: \text{ENVIRONMENT}, \text{phase}: \text{PHASE}): \text{OBJORREF} \]
\[ \text{ListExpression} \equiv \text{AssignmentExpression} \]
\[ \text{do} \]
\[ \text{return Eval}[\text{AssignmentExpression}](\text{env}, \text{phase}); \]
\[ \text{ListExpression}, \text{AssignmentExpression} \]
\[ \text{do} \]
\[ \text{return readReference(Eval}[\text{ListExpression}](\text{env}, \text{phase}), \text{phase}); \]
\[ \text{return readReference(Eval}[\text{AssignmentExpression}](\text{env}, \text{phase}), \text{phase}) \]
\[ \text{end proc}; \]

\[ \text{proc EvalAsList} [\text{ListExpression}] (\text{env}: \text{ENVIRONMENT}, \text{phase}: \text{PHASE}): \text{OBJECT}[] \]
\[ \text{ListExpression} \equiv \text{AssignmentExpression} \]
\[ \text{do} \]
\[ \text{elt: OBJECT} \equiv \text{readReference(Eval}[\text{AssignmentExpression}](\text{env}, \text{phase}), \text{phase}); \]
\[ \text{return [elt];} \]
\[ \text{ListExpression}, \text{AssignmentExpression} \]
\[ \text{do} \]
\[ \text{els: OBJECT} \equiv \text{EvalAsList}[\text{ListExpression}](\text{env}, \text{phase}); \]
\[ \text{elt: OBJECT} \equiv \text{readReference(Eval}[\text{AssignmentExpression}](\text{env}, \text{phase}), \text{phase}); \]
\[ \text{return els} \oplus [elt] \]
\[ \text{end proc}; \]

12.21 Type Expressions

Syntax

\[ \text{TypeExpression} \equiv \text{NonAssignmentExpression} \]

Validation

\[ \text{proc Validate} [\text{TypeExpression} \equiv \text{NonAssignmentExpression}] (\text{ext}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT}) \]
\[ \text{Validate}[\text{NonAssignmentExpression}](\text{ext}, \text{env}) \]
\[ \text{end proc}; \]
Setup and Evaluation

proc SetupAndEval[TypeExpression \[ NonAssignmentExpression \)] (env: ENVIRONMENT): CLASSSetup[NonAssignmentExpression]( );
o: OBJECT \[ readReference(Eval[NonAssignmentExpression](env, compile), compile)\];
return toClass(o)
end proc;

13 Statements

Syntax

\[
\text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Statement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{ExpressionStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{SuperStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Block} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{LabeledStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{IfStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{SwitchStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{DoStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{WhileStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{ForStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{WithStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{ContinueStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{BreakStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{ReturnStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{ThrowStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{TryStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Substatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{EmptyStatement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Statement} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{SimpleVariableDefinition} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Attributes} \text{\{no line break\}} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Substatements} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{\{empty\}} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{SubstatementsPrefix} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{Semicolon} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
; \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{VirtualSemicolon} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{\{empty\}} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{VirtualSemicolon} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]

\[
\text{\{empty\}} \text{\{abbrev, noShortIf, full\}} \text{\{abbrev, noShortIf, full\}}
\]
Semicolon

; VirtualSemicolon

Validation

  [Statement [] ExpressionStatement Semicolon] do
    Validate[ExpressionStatement](ctx, env);
  [Statement [] SuperStatement Semicolon] do Validate[SuperStatement](ctx, env);
  [Statement [] Block] do Validate[Block](ctx, env, jt, preinst);
  [Statement [] LabeledStatement] do Validate[LabeledStatement](ctx, env, sl, jt);
  [Statement [] IfStatement] do Validate[IfStatement](ctx, env, jt);
  [Statement [] SwitchStatement] do Validate[SwitchStatement](ctx, env, jt);
  [Statement [] DoStatement Semicolon] do Validate[DoStatement](ctx, env, sl, jt);
  [Statement [] WhileStatement] do Validate[WhileStatement](ctx, env, sl, jt);
  [Statement [] ForStatement] do Validate[ForStatement](ctx, env, sl, jt);
  [Statement [] WithStatement] do Validate[WithStatement](ctx, env, jt);
  [Statement [] ContinueStatement Semicolon] do Validate[ContinueStatement](jt);
  [Statement [] BreakStatement Semicolon] do Validate[BreakStatement](jt);
  [Statement [] ReturnStatement Semicolon] do Validate[ReturnStatement](ctx, env);
  [Statement [] ThrowStatement Semicolon] do Validate[ThrowStatement](ctx, env);
  [Statement [] TryStatement] do Validate[TryStatement](ctx, env, jt)
end proc;

Enabled[Substatement]: BOOLEAN;

  [Substatement [] EmptyStatement] do nothing;
  [Substatement [] Statement] do Validate[Statement](ctx, env, sl, jt, false);
  [Substatement [] SimpleVariableDefinition Semicolon] do
    Validate[SimpleVariableDefinition](ctx, env);
  [Substatement [] Attributes] no line break { Substatements ] do
    Validate[Attributes](ctx, env);
    Setup[Attributes]();
    attr: ATTRIBUTE [] Eval[Attributes](env, compile);
    if attr: BOOLEAN then
      throw a Type Error exception — attributes other than true and false may be used in a statement but not a substatement
    end if;
    Enabled[Substatement] [] attr;
    if attr then Validate[Substatements](ctx, env, jt) end if
end proc;
proc Validate[Substatements] (ctx: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [Substatements □ «empty»] do nothing;
    [Substatements □ SubstatementsPrefix Substatement<abbrev>] do
        Validate[SubstatementsPrefix](ctx, env, jt);
        Validate[Substatement<abbrev>](ctx, env, {}, jt)
    end proc;

proc Validate[SubstatementsPrefix] (ctx: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
    [SubstatementsPrefix □ «empty»] do nothing;
    [SubstatementsPrefix<0> SubstatementsPrefix<1> Substatement<full>] do
        Validate[SubstatementsPrefix<0>](ctx, env, jt);
        Validate[Substatement<full>](ctx, env, {}, jt)
    end proc;

Setup

Setup[Statement<0>] () propagates the call to Setup to every nonterminal in the expansion of Statement<1>.

proc Setup[Substatement<0>] ()
    [Substatement<0> □ EmptyStatement] do nothing;
    [Substatement<0> □ Statement<1>] do Setup[Statement<1>];
    [Substatement<0> □ SimpleVariableDefinition Semicolon<0>] 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[Semicolon<0>] ()
    [Semicolon<0> □ :] do nothing;
    [Semicolon<0> □ VirtualSemicolon] do nothing;
    [Semicolon<0><abbrev> □ «empty»] do nothing;
    [Semicolon<0><noShort<0> □ «empty»] do nothing
end proc;

Evaluation

proc Eval[Statement<0>] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [Statement<0> □ ExpressionStatement Semicolon<0>] do
        return Eval[ExpressionStatement](env);
EmptyStatement

13.1 Empty Statement

Syntax

EmptyStatement ;
13.2 Expression Statement

Syntax

\[
\text{ExpressionStatement} \rightarrow \text{[lookahead]} \{function, \} \text{ListExpression}\text{allowIn}
\]

Validation

\[
\text{proc Validate}[\text{ExpressionStatement} \rightarrow \text{[lookahead]} \{function, \} \text{ListExpression}\text{allowIn}]
(\text{cxt}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT})
\text{Validate}[\text{ListExpression}\text{allowIn}](\text{cxt, env})
\]
end proc;

Setup

\[
\text{proc Setup}[\text{ExpressionStatement} \rightarrow \text{[lookahead]} \{function, \} \text{ListExpression}\text{allowIn}] ()
\text{Setup}[\text{ListExpression}\text{allowIn}]() 
\]
end proc;

Evaluation

\[
\text{proc Eval}[\text{ExpressionStatement} \rightarrow \text{[lookahead]} \{function, \} \text{ListExpression}\text{allowIn}](\text{env}: \text{ENVIRONMENT}): \text{OBJECT}
\text{return readReference(Eval[\text{ListExpression}\text{allowIn}](\text{env, run}), \text{run})}
\]
end proc;

13.3 Super Statement

Syntax

\[
\text{SuperStatement} \rightarrow \text{super Arguments}
\]

Validation

\[
\text{proc Validate}[\text{SuperStatement} \rightarrow \text{super Arguments}](\text{cxt}: \text{CONTEXT}, \text{env}: \text{ENVIRONMENT})
frame: \text{PARAMETERFRAMEOPT} \rightarrow \text{getEnclosingParameterFrame(env)};
\text{if} frame = \text{none} \text{or} frame.\text{kind} ≠ \text{constructorFunction} \text{then}
\text{throw a SyntaxError exception} -- \text{a super statement is meaningful only inside a constructor}
\text{end if};
\text{Validate[Arguments]}(\text{cxt, env});
frame.\text{callsSuperconstructor} \rightarrow \text{true}
\]
end proc;

Setup

\[
\text{proc Setup}[\text{SuperStatement} \rightarrow \text{super Arguments}] ()
\text{Setup[Arguments]}() 
\]
end proc;
Evaluation

proc Eval[SuperStatement [] super Arguments](env: ENVIRONMENT): OBJECT
frame: PARAMETERFRAMEOPT [] getEnclosingParameterFrame(env);
  note Validate already ensured that frame ≠ none and frame.kind = constructorFunction.
args: OBJECT[] [] Eval[Arguments](env, run);
if frame.superconstructorCalled = true then
  throw a ReferenceError exception — the superconstructor cannot be called twice
end if;
c: CLASS [] getEnclosingClass(env);
this: OBJECTOPT [] frame.this;
  note this SIMPLEINSTANCE;
callInit(this, c.super, args, run);
frame.superconstructorCalled [] true;
return this
end proc;

13.4 Block Statement

Syntax

Block [] { Directives }

Validation

CompileFrame[Block]: LOCALFRAME;
Preinstantiate[Block]: BOOLEAN;

proc ValidateUsingFrame[Block [] { Directives }]
  (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN, frame: FRAME)
localCxt: CONTEXT [] new CONTEXT[]strict: cxt.strict, openNamespaces: cxt.openNamespaces[]
Validate[Directives](localCxt, [frame] @ env, jt, preinst, none)
end proc;

proc Validate[Block [] { Directives }](cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN)
  compileFrame: LOCALFRAME [] new LOCALFRAME[]localBindings: {}[]
CompileFrame[Block] [] compileFrame;
Preinstantiate[Block] [] preinst;
ValidateUsingFrame[Block](cxt, env, jt, preinst, compileFrame)
end proc;

Setup

proc Setup[Block [] { Directives }]()()
  Setup[Directives]()
end proc;
Evaluation

proc Eval[Block \{ Directives \}] (env: ENVIRONMENT, d: OBJECT): OBJECT
  compileFrame: LOCALFRAME □ CompileFrame[Block];
  runtimeFrame: LOCALFRAME;
  if Preinstantiate[Block] then runtimeFrame □ compileFrame
  else runtimeFrame □ instantiateLocalFrame(compileFrame, env)
  end if;
  return Eval[Directives][runtimeFrame ⊕ env, d]
end proc;

proc EvalUsingFrame[Block \{ Directives \}] (env: ENVIRONMENT, frame: FRAME, d: OBJECT): OBJECT
  return Eval[Directives][frame ⊕ env, d]
end proc;

13.5 Labeled Statements

Syntax

LabeledStatement: Identifier : Substatement

Validation

  name: STRING □ Name[Identifier];
  if name \\ jt.breakTargets then
    throw a SyntaxError exception — nesting labeled statements with the same label is not permitted
  end if;
  jt2: JUMPTARGETS □ JUMPTARGETS[breakTargets: jt.breakTargets □ \{ name\},
                          continueTargets: jt.continueTargets]
  Validate[Substatement](ext, env, sl □ \{ name\}, jt2)
end proc;

Setup

proc Setup[LabeledStatement \{ Identifier \} : Substatement] ()
  Setup[Substatement]()
end proc;

Evaluation

  try return Eval[Substatement](env, d)
  catch x: SEMANTICEXCEPTION do
    if x.BREAK and x.label = Name[Identifier] then return x.value
    else throw x
  end if
  end try
end proc;
13.6 If Statement

Syntax

\[
\text{IfStatement}^{\text{abbrev}} \equiv
\begin{align*}
\text{if} & \text{ ParenListExpression} \text{ Substatement}^{\text{abbrev}} \\
\text{if} & \text{ ParenListExpression} \text{ Substatement}^{\text{noShortIf}} \text{ else} \text{ Substatement}^{\text{abbrev}}
\end{align*}
\]

\[
\text{IfStatement}^{\text{full}} \equiv
\begin{align*}
\text{if} & \text{ ParenListExpression} \text{ Substatement}^{\text{full}} \\
\text{if} & \text{ ParenListExpression} \text{ Substatement}^{\text{noShortIf}} \text{ else} \text{ Substatement}^{\text{full}}
\end{align*}
\]

\[
\text{IfStatement}^{\text{noShortIf}} \equiv \text{ if} \text{ ParenListExpression} \text{ Substatement}^{\text{noShortIf}} \text{ else} \text{ Substatement}^{\text{noShortIf}}
\]

Validation

\[
\text{proc Validate[IfStatement]} (\text{cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS})
\]

\[
\text{IfStatement}^{\text{abbrev}} \text{ do}
\]

\[
\text{Validate[ParenListExpression]} (\text{cxt, env});
\]

\[
\text{Validate[Substatement]} (\text{cxt, env, } \{\}, \text{jt});
\]

\[
\text{IfStatement}^{\text{full}} \text{ do}
\]

\[
\text{Validate[ParenListExpression]} (\text{cxt, env});
\]

\[
\text{Validate[Substatement]} (\text{cxt, env, } \{\}, \text{jt});
\]

\[
\text{IfStatement}^{\text{noShortIf}} \text{ do}
\]

\[
\text{Validate[ParenListExpression]} (\text{cxt, env});
\]

\[
\text{Validate[Substatement]} (\text{cxt, env, } \{\}, \text{jt});
\]

\]

end proc;

Setup

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

Evaluation

\[
\text{proc Eval[IfStatement]} (\text{env: ENVIRONMENT, d: OBJECT): OBJECT}
\]

\[
\text{IfStatement}^{\text{abbrev}} \text{ do}
\]

\[
o: \text{OBJECT} \text{ if toBoolean}(o, \text{run}) \text{ then return Eval[Substatement]} (\text{env, d})
\]

\[
\text{else return d}
\]

end if;

\[
\text{IfStatement}^{\text{full}} \text{ do}
\]

\[
o: \text{OBJECT} \text{ if toBoolean}(o, \text{run}) \text{ then return Eval[Substatement]} (\text{env, d})
\]

\[
\text{else return d}
\]

end if;

\[
\text{IfStatement}^{\text{noShortIf}} \text{ do}
\]

\[
o: \text{OBJECT} \text{ if toBoolean}(o, \text{run}) \text{ then return Eval[Substatement]} (\text{env, d})
\]

\[
\text{else return Eval[Substatement[2]} (\text{env, d})
\]

end if;

end proc;
13.7 Switch Statement

Semantics

tuple SWITCHKEY
key: OBJECT
end tuple;

SWITCHGUARD = SWITCHKEY {default} OBJECT;

Syntax

SwitchStatement switch ParenListExpression { CaseElements }

CaseElements «empty»
| CaseLabel
| CaseLabel CaseElementsPrefix CaseElement

CaseElementsPrefix «empty»
| CaseElementsPrefix CaseElement

CaseElement Directive
| CaseLabel

CaseLabel

case ListExpression [default]
| default:

Validation

CompileFrame[SwitchStatement]: LOCALFRAME;

proc Validate[SwitchStatement switch ParenListExpression { CaseElements }]
(ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
if NDefaults[CaseElements] > 1 then
    throw a SyntaxError exception — a case statement may have at most one default clause
end if;
Validate[ParenListExpression](ext, env);
jt2: JUMPTARGETS [ JUMPTARGETS breakTargets: jt.breakTargets {default},
    continueTargets: jt.continueTargets]
    compileFrame: LOCALFRAME new LOCALFRAME[localBindings: {}]
    CompileFrame[SwitchStatement] compileFrame;
    localCxt: CONTEXT [ new CONTEXT[strict: ext.strict, openNamespaces: ext.openNamespaces]
    Validate[CaseElements](localCxt, [compileFrame] env, jt2)
end proc;

NDefaults[CaseElements]: INTEGER;
NDefaults[CaseElements «empty»] = 0;
NDefaults[CaseElements CaseLabel] = NDefaults[CaseLabel];
NDefaults[CaseElements CaseLabel CaseElementsPrefix CaseElement]
    = NDefaults[CaseLabel] + NDefaults[CaseElementsPrefix] + NDefaults[CaseElement]
Validate[CaseElements] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS) propagates the call to Validate to every nonterminal in the expansion of CaseElements.

NDefaults[CaseElementsPrefix]: INTEGER;
NDefaults[CaseElementsPrefix «empty»] = 0;
NDefaults[CaseElementsPrefix0 « CaseElementsPrefix1 CaseElement$^0$] = NDefaults[CaseElementsPrefix1] + NDefaults[CaseElement$^0$];

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

NDefaults[CaseElement$^0$]: INTEGER;
NDefaults[CaseElement$^0$ « Directive$^0$] = 0;
NDefaults[CaseElement$^0$ « CaseLabel] = NDefaults[CaseLabel];

proc Validate[CaseElement$^0$] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
[CaseElement$^0$ « Directive$^0$] do Validate[Directive$^0$](ext, env, jt, false, none);
[CaseElement$^0$ « CaseLabel] do Validate[CaseLabel](ext, env, jt)
end proc;

NDefaults[CaseLabel]: INTEGER;
NDefaults[CaseLabel « case ListExpression$^{allowIn}$] = 0;
NDefaults[CaseLabel « default : ] = 1;

proc Validate[CaseLabel] (ext: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
[CaseLabel « case ListExpression$^{allowIn}$] do
  Validate[ListExpression$^{allowIn}$](ext, env);
[CaseLabel « default : ] do nothing
end proc;

Setup

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

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

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

Setup[CaseElement$^0$] () propagates the call to Setup to every nonterminal in the expansion of CaseElement$^0$.

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

```ecmascript
proc Eval[SwitchStatement] [] switch ParenListExpression { CaseElements } ]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
key: OBJECT [] readReference(Eval[ParenListExpression](env, run), run);
compileFrame: LOCALFRAME [] CompileFrame[SwitchStatement];
runtimeFrame: LOCALFRAME [] instantiateLocalFrame(compileFrame, env);
runtimeEnv: ENVIRONMENT [] runtimeFrame + env;
result: SWITCHGUARD [] Eval[CaseElements](runtimeEnv, SWITCHKEY[default, d]);
if result [] OBJECT then return result end if;
ote result = SWITCHKEY[default, d];
return d
end proc;

[CaseElements [] «empty»] do return guard;
[CaseElements [] CaseLabel do return Eval[CaseLabel](env, guard, d);
[CaseElements [] CaseLabel CaseElementsPrefix CaseElementabbrev] do
guard2: SWITCHGUARD [] Eval[CaseLabel](env, guard, d);
guard3: SWITCHGUARD [] Eval[CaseElementsPrefix](env, guard2, d);
return Eval[CaseElementabbrev](env, guard3, d)
end proc;

[CaseElementsPrefix [] «empty»] do return guard;
[CaseElementsPrefix0 CaseElementsPrefix1 CaseElementabbrev] do
guard2: SWITCHGUARD [] Eval[CaseElementsPrefix1](env, guard, d);
return Eval[CaseElementabbrev](env, guard2, d)
end proc;

[CaseElement [] Directive] do
  case guard of
    SWITCHKEY [] (default) do return guard;
    OBJECT do return Eval[Directive](env, guard)
  end case;
[CaseElement [] CaseLabel do return Eval[CaseLabel](env, guard, d)
end proc;

[CaseLabel [] case ListExpressionabbrev : ] do
case guard of
  [default] [] OBJECT do return guard;
  SWITCHKEY do
    label: OBJECT [] readReference(Eval[ListExpressionabbrev](env, run), run);
    if isStrictlyEqual(guard, key, label, run) then return d
    else return guard
  end if
end case;
```

[CaseLabel [] default :] do
case guard of
  SWITCHKEY [] OBJECT do return guard;
  {default} do return d end case
end proc;

13.8 Do-While Statement

Syntax

DoStatement [] do Substatement\textsuperscript{abbrev} while ParenListExpression

Validation

Labels[DoStatement]: LABEL{};

proc Validate[DoStatement [] do Substatement\textsuperscript{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\textsuperscript{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\textsuperscript{abbrev} while ParenListExpression] (env: ENVIRONMENT, d: OBJECT): OBJECT
  try
d1: OBJECT [] d;
  while true do
    try d1 [] Eval[Substatement\textsuperscript{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;
o: OBJECT [] readReference(Eval[ParenListExpression](env, run), 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 \[ \begin{array}{l}
\text{while} \quad \text{ParenListExpression} \quad \text{Substatement} \\
\end{array} \]

Validation

Labels[WhileStatement] \[ \begin{array}{l}
\text{LABEL} \\
\end{array} \]

proc Validate[WhileStatement \[ \begin{array}{l}
\text{while} \quad \text{ParenListExpression} \quad \text{Substatement} \\
\end{array} \]]
\[ \begin{array}{l}
(cxt: \text{CONTEXT}, \ env: \text{ENVIRONMENT}, \ sl: \text{LABEL}, \ jt: \text{JUMPTARGETS}) \\
\end{array} \]
\[ \begin{array}{l}
\text{continueLabels: Label} \{ \text{default} \}; \\
\text{Labels[WhileStatement]} \{ \text{default} \} \}
\end{array} \]
\[ \begin{array}{l}
\text{jt2: JUMPTARGETS} \}
\end{array} \]
\[ \begin{array}{l}
\text{breakTargets: Label} \{ \text{default} \}, \\
\text{continueTargets: JUMPTARGETS} \}
\end{array} \]
\[ \begin{array}{l}
\text{Validate[ParenListExpression]}(cxt, \ env); \\
\text{Validate[Substatement]}(cxt, \ env, \{} \), \ jt2 \}
\end{array} \]
end proc;

Setup

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

Evaluation

proc Eval[WhileStatement \[ \begin{array}{l}
\text{while} \quad \text{ParenListExpression} \quad \text{Substatement} \\
\end{array} \]] (env: \text{ENVIRONMENT}, \ d: \text{OBJECT}): \text{OBJECT}
\[ \begin{array}{l}
\text{try} \\
\end{array} \]
\[ \begin{array}{l}
\text{d1: OBJECT} \}
\end{array} \]
\[ \begin{array}{l}
\text{while toBoolean(readReference(Eval[ParenListExpression]}(env, \ run, \ run, \ run)) do} \\
\text{try d1 \}
\end{array} \]
\[ \begin{array}{l}
\text{Eval[Substatement]}(env, \ d1) \\
\text{catch x: SEMANTICEXCEPTION do} \\
\text{if x \{ CONTINUE and x.label \}
\end{array} \]
\[ \begin{array}{l}
\text{Labels[WhileStatement]} then} \\
\text{d1 \}
\end{array} \]
\[ \begin{array}{l}
\text{x.value} \\
\text{end if} \\
\text{end try} \\
\text{end while;} \\
\text{return d1} \\
\text{catch x: SEMANTICEXCEPTION do} \\
\text{if x \{ BREAK and x.label = default then return x.value else throw x} \\
\text{end if} \\
\text{end try} \\
\text{end proc;}
\]

13.10 For Statements

Syntax

ForStatement \[ \begin{array}{l}
\text{for} \ ( \text{ForInitialiser} ; \text{OptionalExpression} ; \text{OptionalExpression} ) \quad \text{Substatement} \\
\text{| for} \ ( \text{ForInBinding in} \ \text{ListExpression}) \quad \text{Substatement} \\
\end{array} \]

Validation

Labels[ForStatement];

CompileLocalFrame[ForStatement]: LOCALFRAME;

proc Validate[ForStatement] (ext: CONTEXT, env: ENVIRONMENT, sl: LABEL, jt: JUMPTARGETS)

[ForStatement] for (ForInitialiser, OptionalExpression, OptionalExpression) Substatement do

continueLabels: LABEL sl default;

Labels[ForStatement] continueLabels;

jt2: JUMPTARGETS breakTargets: jt.breakTargets default;

continueTargets: jt.continueTargets continueLabels;

compileLocalFrame: LOCALFRAME new LOCALFRAME localBindings: {}[]

CompileLocalFrame[ForStatement] compileLocalFrame;

compileEnv: ENVIRONMENT [[compileLocalFrame] env];

Validate[ForInitialiser](ext, compileEnv);

Validate[OptionalExpression1](ext, compileEnv);

Validate[OptionalExpression2](ext, compileEnv);

Validate[Substatement](ext, compileEnv, {}, jt2);

[ForStatement] for (ForInBinding in ListExpression Substatement) do

continueLabels: LABEL sl default;

Labels[ForStatement] continueLabels;

jt2: JUMPTARGETS breakTargets: jt.breakTargets default;

continueTargets: jt.continueTargets continueLabels;

Validate[ListExpression](ext, env);

compileLocalFrame: LOCALFRAME new LOCALFRAME localBindings: {}[]

CompileLocalFrame[ForStatement] compileLocalFrame;

compileEnv: ENVIRONMENT [[compileLocalFrame] env];

Validate[ForInBinding](ext, compileEnv);

Validate[Substatement](ext, compileEnv, {}, jt2)
end proc;

Enabled[ForInitialiser]: BOOLEAN;

proc Validate[ForInitialiser] (ext: CONTEXT, env: ENVIRONMENT)

[ForInitialiser] «empty» do nothing;

[ForInitialiser] ListExpression do Validate[ListExpression](ext, env);

[ForInitialiser] VariableDefinition do

Validate[VariableDefinition](ext, env, none);
[ForInitialiser] Attributes [no line break] VariableDefinition

Validate[Attributes](ext, env);
Setup[Attributes]();
attr: ATTRIBUTE [] Eval[Attributes](env, compile);
Enabled[ForInitialiser] [] attr ≠ false;
if attr ≠ false then Validate[VariableDefinition](ext, env, attr) end if
end proc;

proc Validate[ForInBinding](ext: CONTEXT, env: ENVIRONMENT)
[ForInBinding] PostfixExpression do Validate[PostfixExpression](ext, env);
[ForInBinding] VariableDefinition Kind VariableBinding do
Validate[VariableBinding](ext, env, none, Immutable[VariableDefinition Kind], true);
[ForInBinding] Attributes [no line break] VariableDefinition Kind VariableBinding do
Validate[Attributes](ext, env);
Setup[Attributes]();
attr: ATTRIBUTE [] Eval[Attributes](env, compile);
if attr = false then
    throw an AttributeError exception — the false attribute cannot be applied to a for-in variable definition
end if;
Validate[VariableBinding](ext, env, attr, Immutable[VariableDefinition Kind], true)
end proc;

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

Setup

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

proc Setup[ForInitialiser]() [ForInitialiser] «empty» do nothing;
[ForInitialiser] ListExpression do Setup[ListExpression]();
[ForInitialiser] VariableDefinition do Setup[VariableDefinition]();
[ForInitialiser] Attributes [no line break] VariableDefinition do
    if Enabled[ForInitialiser] then Setup[VariableDefinition]() end if
end proc;

proc Setup[ForInBinding]() [ForInBinding] PostfixExpression do Setup[PostfixExpression]();
[ForInBinding] VariableDefinition Kind VariableBinding do
    Setup[VariableBinding]();
[ForInBinding] Attributes [no line break] VariableDefinition Kind VariableBinding do
    Setup[VariableBinding]()
end proc;

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

\[
\text{proc} \text{Eval}[\text{ForStatement}^{(f)}] (env: \text{ENVIRONMENT}, d: \text{OBJECT}) : \text{OBJECT} \\
[\text{ForStatement}^{(f)} \equiv \text{for} (\text{ForInitialiser} ; \text{OptionalExpression}_1 ; \text{OptionalExpression}_2) \text{ Substatement}^{(g)}] \text{ do} \\
\text{runtimeLocalFrame: LOCALFRAME } \equiv \text{ instantiateLocalFrame}(\text{CompileLocalFrame}[\text{ForStatement}^{(f)}, env); \\
\text{runtimeEnv: ENVIRONMENT } \equiv [\text{runtimeLocalFrame} \oplus env; \\
\text{try} \\
\text{Eval}[\text{ForInitialiser}](\text{runtimeEnv}); \\
d1: \text{OBJECT } \equiv d; \\
\text{while} \equiv \text{toBoolean}(\text{readReference}(\text{Eval}[\text{OptionalExpression}_1](\text{runtimeEnv}, \text{run}, \text{run}, \text{run})) \text{ do} \\
\text{try} d1 \equiv \text{Eval}[\text{Substatement}^{(g)}](\text{runtimeEnv}, d1) \\
catch x: \text{SEMANTICEXCEPTION} \text{ do} \\
\text{if} x \equiv \text{CONTINUE} \text{ and } x.\text{label} \equiv \text{Labels}[\text{ForStatement}^{(f)}] \text{ then} \\
d1 \equiv x.\text{value} \\
\text{else} \text{throw } x \\
\text{end if} \\
\text{end try}; \\
\text{readReference}(\text{Eval}[\text{OptionalExpression}_2](\text{runtimeEnv}, \text{run}, \text{run})) \\
\text{end while}; \\
\text{return } d1 \\
catch x: \text{SEMANTICEXCEPTION} \text{ do} \\
\text{if} x \equiv \text{BREAK} \text{ and } x.\text{label} = \text{default} \text{ then return } x.\text{value} \text{ else throw } x \text{ end if} \\
\text{end try};
\]
```javascript
try {
  o: OBJECT readReference(Eval[ListExpression[allowIn](env, run), run]);
  c: CLASS objectType(o);
  oldIndices: OBJECT{ } c.enumerate(o);
  remainingIndices: OBJECT{ } oldIndices;
  d1: OBJECT d;
  while remainingIndices ≠ {} do
    runtimeLocalFrame: LOCALFRAME instantiateLocalFrame(CompileLocalFrame[ForStatement[0]], env);
    runtimeEnv: ENVIRONMENT runtimeLocalFrame ⊕ env;
    index: OBJECT any element of remainingIndices;
    remainingIndices: remainingIndices – {index};
    WriteBinding[ForInBinding][runtimeEnv, index];
    try d1 Eval[Substatement[0]](runtimeEnv, d1)
    catch x: SEMANTICEXCEPTION do
      if x CONTINUE and x.label Labels[ForStatement[0]] then
        d1 x.value
      else throw x
      end if;
      newIndices: OBJECT{ } c.enumerate(o);
      if newIndices ≠ oldIndices then
        The implementation may, at its discretion, add none, some, or all of the objects in the set difference
        newIndices – oldIndices to remainingIndices;
        The implementation may, at its discretion, remove none, some, or all of the objects in the set difference
        oldIndices – newIndices from remainingIndices;
      end if;
      oldIndices: newIndices
    end while;
    return d1
  catch x: SEMANTICEXCEPTION do
    if x BREAK and x.label = default then return x.value else throw x
  end try
} end try
}
```
[ForInBinding Attributes [no line break] VariableDefinitionKind VariableBinding^val^] do
  WriteBinding[VariableBinding^val^](env, newValue)
end proc;

proc Eval[OptionalExpression](env: ENVIRONMENT, phase: PHASE): OBJREF
  [OptionalExpression ListExpression^allow^] do
    return Eval[ListExpression^allow^](env, phase);
  [OptionalExpression «empty»] do return true
end proc;

13.11 With Statement

Syntax

WithStatement^w^ with ParenListExpression Substatement^w^

Validation

CompileLocalFrame[WithStatement^w^]: LOCALFRAME;

proc Validate[WithStatement^w^ with ParenListExpression Substatement^w^](cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  Validate[ParenListExpression](cxt, env);
  compileWithFrame: WITHFRAME new WITHFRAME[value: none]
  compileLocalFrame: LOCALFRAME new LOCALFRAME[localBindings: {}]
  CompileLocalFrame[WithStatement^w^] compileLocalFrame;
  compileEnv: ENVIRONMENT [compileLocalFrame] compileWithFrame env;
  Validate[Substatement^w^](cxt, compileEnv, {}, jt)
end proc;

Setup

Setup[WithStatement^w^] () propagates the call to Setup to every nonterminal in the expansion of WithStatement^w^.

Evaluation

proc Eval[WithStatement^w^ with ParenListExpression Substatement^w^](env: ENVIRONMENT, d: OBJECT): OBJECT
  value: OBJECT readReference(Eval[ParenListExpression](env, run), run);
  runtimeWithFrame: WITHFRAME new WITHFRAME[value: value]
  runtimeLocalFrame: LOCALFRAME
    instantiateLocalFrame(CompileLocalFrame[WithStatement^w^], runtimeWithFrame env);
  runtimeEnv: ENVIRONMENT [runtimeLocalFrame] runtimeWithFrame env;
  return Eval[Substatement^w^](runtimeEnv, d)
end proc;

13.12 Continue and Break Statements

Syntax

ContinueStatement
  continue
  | continue [no line break] Identifier
BreakStatement
d | break [no line break] Identifier

Validation

proc Validate[ContinueStatement] (jt: JUMPTARGETS)
    [ContinueStatement [] continue] do
        if default [] jt.continueTargets then
            throw a SyntaxError exception — there is no enclosing statement to which to continue
        end if;
    [ContinueStatement [] continue] do
        if Name[Identifier] [] jt.continueTargets then
            throw a SyntaxError exception — there is no enclosing labeled statement to which to continue
        end if;
end proc;

proc Validate[BreakStatement] (jt: JUMPTARGETS)
    [BreakStatement [] break] do
        if default [] jt.breakTargets then
            throw a SyntaxError exception — there is no enclosing statement to which to break
        end if;
    [BreakStatement [] break] do
        if Name[Identifier] [] jt.breakTargets then
            throw a SyntaxError exception — there is no enclosing labeled statement to which to break
        end if;
end proc;

Setup

proc Setup[ContinueStatement] ()
    [ContinueStatement [] continue] do nothing;
    [ContinueStatement [] continue] do nothing
end proc;

proc Setup[BreakStatement] ()
    [BreakStatement [] break] do nothing;
    [BreakStatement [] break] do nothing
end proc;

Evaluation

proc Eval[ContinueStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
    [ContinueStatement [] continue] do throw continue[value: d, label: default[]]
    [ContinueStatement [] continue] do throw continue[value: d, label: Name[Identifier]]
end proc;

    [BreakStatement [] break] do throw break[value: d, label: default[]]
    [BreakStatement [] break] do throw break[value: d, label: Name[Identifier]]
end proc;
13.13 Return Statement

Syntax

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

Validation

```plaintext
func Validate[ReturnStatement] (cxt: CONTEXT, env: ENVIRONMENT)
  [ReturnStatement [] return] do
    if getEnclosingParameterFrame(env) = none then
      throw a SyntaxError exception — a return statement must be located inside a function
    end if;
  [ReturnStatement [] return [no line break] ListExpression] do
    if frame = none then
      throw a SyntaxError exception — a return statement must be located inside a function
    end if;
    if cannotReturnValue(frame) then
      throw a SyntaxError exception — a return statement inside a setter or constructor cannot return a value
    end if;
  Validate[ListExpression](cxt, env)
end proc;
```

Setup

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

Evaluation

```plaintext
proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
  [ReturnStatement [] return] do throw RETURN[] value: undefined[]
  [ReturnStatement [] return [no line break] ListExpression] do
    a: OBJECT [] readReference(Eval[ListExpression](env, run), run);
    throw RETURN[] value: a[]
end proc;
```

`cannotReturnValue(frame)` returns `true` if the function represented by `frame` cannot return a value because it is a setter or constructor.

```plaintext
proc cannotReturnValue(frame: PARAMETERFRAME): BOOLEAN
  return frame.kind = constructorFunction or frame.handling = set
end proc;
```

13.14 Throw Statement

Syntax

```plaintext
ThrowStatement [] throw [no line break] ListExpression
```
Validation

proc Validate[ThrowStatement [] throw [no line break] ListExpression allowIn] (ext: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpression allowIn](ext, 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
  a: OBJECT [] readReference(Eval[ListExpression allowIn](env, run), run);
  throw 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, false);
    Validate[CatchClauses](ext, env, jt);
  [TryStatement [] try Block1 CatchClausesOpt finally Block2] do
    Validate[Block1](ext, env, jt, false);
    Validate[CatchClausesOpt](ext, env, jt);
    Validate[Block2](ext, env, jt, false)
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.

CompileEnv[CatchClause]: ENVIRONMENT;
**CompileFrame**[CatchClause]:: LOCALFRAME;

**proc** Validate[CatchClause [] catch ( Parameter ) Block] (**ext**: CONTEXT, **env**: ENVIRONMENT, **jt**: JUMPTARGETS)

**compileFrame**: LOCALFRAME [] new LOCALFRAME[localBindings: {}]

**compileEnv**: ENVIRONMENT [] [compileFrame] ⊕ env;

**CompileFrame**[CatchClause] [] compileFrame;

**CompileEnv**[CatchClause] [] compileEnv;

**Validate**[Parameter](**ext**, compileEnv, compileFrame);

**Validate**[Block](**ext**, compileEnv, **jt**, false)

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]()

**Setup**[Parameter](CompileEnv[CatchClause], CompileFrame[CatchClause], none);

**Setup**[Block]();

end proc;

**Evaluation**

**proc** Eval[TryStatement] (**env**: ENVIRONMENT, **d**: OBJECT): OBJECT

[TryStatement [] try Block CatchClauses] do

try return Eval[Block](**env**, **d**)

catch x: SEMANTICEXCEPTION do

if x [] CONTROLTRANSFER then throw x

else

r: OBJECT [] (reject) [] Eval[CatchClauses](**env**, x);

if r ≠ reject then return r else throw x end if

end if

end try;
TryStatement

```javascript
[try Block1, CatchClausesOpt finally Block2] do
  result: OBJECTOpt none;
  exception: SEMANTICException {none} none;
try result [] Eval[Block1](env, d)
catch x: SEMANTICException do exception x
end try;
note At this point exactly one of result and exception has a non-none value.
if exception {} OBJECT then
  try
    r: OBJECT {} {reject} Eval[CatchClausesOpt](env, exception);
    if r ≠ reject then
      note The exception has been handled, so clear it.
      result r;
      exception none
    end if
  catch x: SEMANTICException do
    note The catch clause threw another exception or CONTROLTRANSFER x, so replace the original exception with x.
    exception x
  end try
end if;
note The finally clause is executed even if the original block exited due to a CONTROLTRANSFER (break,
continue, or return).
note The finally clause is not inside a try-catch semantic statement, so if it throws another exception or
CONTROLTRANSFER, then the original exception or CONTROLTRANSFER exception is dropped.
Eval[Block2](env, undefined);
note At this point exactly one of result and exception has a non-none value.
if exception ≠ none then throw exception else return result end if
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);
  [CatchClauses CatchClauses, CatchClause] do
    r: OBJECT {} {reject} Eval[CatchClauses1](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}
  compileFrame: LOCALFRAME [] CompileFrame[catchClause];
  runtimeFrame: LOCALFRAME [] instantiateLocalFrame(compileFrame, env);
  runtimeEnv: ENVIRONMENT [] [runtimeFrame] @ env;
  qname: QUALIFIEDNAME [] public::(Name[Parameter]);
  v: LOCALMEMBEROPT [] findLocalMember(runtimeFrame, {qname}, write);
  note Validate created one local variable with the name in qname, so v [] VARIABLE.
  if v.type.is(exception) then
    writeLocalMember(v, exception, run);
    return Eval[Block](runtimeEnv, undefined)
  else return reject
end if
end proc;

14 Directives

Syntax

Directive[]
  EmptyStatement
  | Statement[]
  | AnnotatableDirective[]
  | Attributes [no line break] AnnotatableDirective[]
  | Attributes [no line break] { Directives }
  |Pragma Semicolon[]

AnnotatableDirective[]
  VariableDefinition abstr Semicolon[]
  | FunctionDefinition
  | ClassDefinition
  | NamespaceDefinition Semicolon[]
  | UseDirective Semicolon[]

Directives[]
  «empty»
  | DirectivesPrefix Directive[abbrev]

DirectivesPrefix[]
  «empty»
  | DirectivesPrefix Directive[full]

Validation

Enabled[Directive[]]: BOOLEAN;

  [Directive[] EmptyStatement] do nothing;
  [Directive[] Statement[]] do
    if attr [] {none, true} then
      throw an AttributeError exception — an ordinary statement only permits the attributes true and false
    end if;
  Validate[Statement[]](ext, env, {}, jt, preinst);
AnnotatableDirective

AnnotatableDirective

proc Validate[AnnotatableDirective](ext, env, preinst, attr); Validate[Attributes](ext, env); Setup[Attributes]();

attr2: ATTRIBUTE Eval[Attributes](env, compile);
attr3: ATTRIBUTE combineAttributes(attr, attr2);
if attr3 = false then Enabled[Directives] [] false
else
    Enabled[Directives] [] true;
    Validate[AnnotatableDirective](ext, env, preinst, attr3)
end if;

Directives

Attributes

Attributes

if attr3 = false then Enabled[Directives] [] false
else
    Enabled[Directives] [] true;
    local Cxt: CONTEXT new CONTEXT strict: ext.strict, openNamespaces: ext.openNamespaces() Validate[Directives](local Cxt, env, jt, preinst, attr3)
end if;

Pragma Semicolon

if attr [] {none, true} then Validate[Pragma](ext)
else
    throw an AttributeError exception — a pragma directive only permits the attributes true and false
end if
end proc;


Validate[VariableDefinition Semicolon](ext, env, attr);
Validate[FunctionDefinition](ext, env, preinst, attr);
Validate[ClassDefinition](ext, env, preinst, attr);
Validate[NamespaceDefinition Semicolon](ext, env, preinst, attr);
Validate[UseDirectives Semicolon](ext, env)
if attr [] {none, true} then Validate[Usedirective](ext, env)
else
    throw an AttributeError exception — a use directive only permits the attributes true and false
end if
end proc;

Validate[Directives](ext: CONTEXT, env: ENVIRONMENT, jt: JUMP TARGETS, preinst: BOOLEAN,
attr: ATTRIBUTE OptNotFalse) propagates the call to Validate to every nonterminal in the expansion of Directives.
Validate[DirectivesPrefix](ext: CONTEXT, env: ENVIRONMENT, jr: JUMPTARGETS, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE) propagates the call to Validate to every nonterminal in the expansion of DirectivesPrefix.

**Setup**

```
proc Setup[Directive^]() {
    if Enabled[Directive^] then Setup[AnnotatableDirective^]() end if;
    if Enabled[Directive^] then Setup[Directives]() end if;
  [Directive^] Pragma Semicolon^ do nothing
end proc;

proc Setup[AnnotatableDirective^]() {
  [AnnotatableDirective^] VariableDefinition^ 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**

```
proc Eval[Directive^](env: ENVIRONMENT, d: OBJECT): OBJECT {
  [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;
    if Enabled[Directive^] then return Eval[Directives](env, d) else return d end if;
  [Directive^] Pragma Semicolon^ do return d
end proc;

proc Eval[AnnotatableDirective^](env: ENVIRONMENT, d: OBJECT): OBJECT {
  [AnnotatableDirective^] VariableDefinition^ do
    return Eval[VariableDefinition^](env, d);
```
14.1 Attributes

Syntax

Attributes → Attribute | AttributeCombination

AttributeCombination → Attribute | 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] (cxt: CONTEXT, env: ENVIRONMENT)
  [NonexpressionAttribute [] final] do nothing;
  [NonexpressionAttribute [] private] do
    if getEnclosingClass(env) = none then
      throw a SyntaxError exception — private is meaningful only inside a class
    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
    a: OBJECT [] readReference(Eval[AttributeExpression](env, phase), phase);
    if a [] ATTRIBUTE then throw an AttributeError exception end if;
    return a;
  [Attribute [] true] do return true;
  [Attribute [] false] do return false;
  [Attribute [] public] do return public;
  [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, enumerable: false, dynamic: false, memberMod: final, overrideMod: none, prototype: false, unused: false}]
[NonexpressionAttribute [] private] do
c: CLASSOPT [] getEnclosingClass(env);
    note Validate ensured that c cannot be none at this point.
    return c.privateNamespace;
[NonexpressionAttribute [] static] do
    return COMPOUNDATTRIBUTE[{namespaces: {}, explicit: false, enumerable: false, dynamic: false, memberMod: static, overrideMod: none, prototype: false, unused: false}]
end proc;

14.2 Use Directive

Syntax

UseDirective [] use namespace ParenListExpression

Validation

    Validate[ParenListExpression](ext, env);
    Setup[ParenListExpression]();
    values: OBJECT[] [] EvalAsList[ParenListExpression](env, compile);
    namespaces: NAMESPACE{[]} {}; {v};
    for each v in values do
        if v in NAMESPACE then throw a TypeError exception end if;
        namespaces [] namespaces [] {v}{}
    end for each;
    ext.openNamespaces ext.openNamespaces namespaces
end proc;

14.3 Pragma

Syntax

Pragma [] use PragmaItems

PragmaItems []
    PragmaItem
    | PragmaItems, PragmaItem

PragmaItem []
    PragmaExpr
    | PragmaExpr ?

PragmaExpr []
    Identifier
    | Identifier (PragmaArgument )
PragmaArgument []
  | true
  | false
  | Number
  | - Number
  | - NegatedMinLong
  | String

Validation

proc Validate[Pragma [] use Pragmaltems] (ctx: CONTEXT)
  Validate[Pragmaltems](ctx)
end proc;

Validate[Pragmaltems] (ctx: CONTEXT) propagates the call to Validate to every nonterminal in the expansion of Pragmaltems.

proc Validate[PragmaItem] (ctx: CONTEXT)
  [PragmaItem []PragmaExpr] do Validate[PragmaExpr](ctx, false);
  [PragmaItem []PragmaExpr ?] do Validate[PragmaExpr](ctx, true)
end proc;

proc Validate[PragmaExpr] (ctx: CONTEXT, optional: BOOLEAN)
  [PragmaExpr [] Identifier] do
    processPragma(ctx, Name[Identifier], undefined, optional);
  [PragmaExpr [] Identifier (PragmaArgument)] do
    arg: OBJECT Value[PragmaArgument];
    processPragma(ctx, 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] = (-2^63)_{long};
Value[PragmaArgument [] String] = Value[String];

proc processPragma(ctx: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN)
  if name = "strict" then
    if value [] {true, undefined} then ctx.strict [] true; return end if;
    if value = false then ctx.strict [] false; return end if
  end if;
  if name = "ecmascript" then
    if value [] {undefined, 4.0} then return end if;
    if value [] {1.0, 2.0, 3.0} then
      An implementation may optionally modify ctx to disable features not available in ECMAScript Edition value other than subsequent pragmas.
      return
    end if;
  end if;
  if not optional then throw a SyntaxError exception end if
end proc;
15 Definitions

15.1 Variable Definition

Syntax

\[
\text{VariableDefinition} = \text{VariableDefinitionKind} \text{ VariableBindingList} \\
\text{VariableDefinitionKind} = \text{var} \mid \text{const} \\
\text{VariableBindingList} = \text{VariableBinding} \mid \text{VariableDefinition}\text{ VariableBindingList}\text{ , VariableBinding} \\
\text{VariableBinding} = \text{TypedIdentifier} \text{ VariableInitialisation} \\
\text{VariableInitialisation} = \{\text{empty}\} \mid = \text{VariableInitialiser} \\
\text{VariableInitialiser} = \text{AssignmentExpression} \mid \text{NonexpressionAttribute} \mid \text{AttributeCombination} \\
\text{TypedIdentifier} = \text{Identifier} \mid \text{Identifier} : \text{TypeExpression} \\
\]

Validation

\[
\text{proc Validate}[\text{VariableDefinition} \text{ VariableDefinitionKind} \text{ VariableBindingList}] (\text{ext: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE}) \\
\quad \text{Validate}[\text{VariableBindingList}](\text{ext, env, attr, Immutable[VariableDefinitionKind], false}) \\
\end{proc};
\]

\[
\text{Immutable[VariableDefinitionKind]} : \text{BOOLEAN}; \\
\quad \text{Immutable[VariableDefinitionKind} = \text{var}] = \text{false}; \\
\quad \text{Immutable[VariableDefinitionKind} = \text{const}] = \text{true}; \\
\]

\[
\text{Validate}[\text{VariableBindingList}] (\text{ext: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE, immutable: BOOLEAN, noInitialiser: BOOLEAN}) \text{ propagates the call to Validate to every nonterminal in the expansion of VariableBindingList.}
\]

\[
\text{CompileEnv[VariableBinding]} : \text{ENVIRONMENT}; \\
\text{CompileVar[VariableBinding]} : \text{VARIABLE} \mid \text{DYNAMICVAR} \mid \text{INSTANCEVARIABLE}; \\
\text{OverriddenVar[VariableBinding]} : \text{INSTANCEVARIABLEOPT}; \\
\text{Multiname[VariableBinding]} : \text{MULTINAME};
\]


Validate[TypedIdentifier](ext, env);
Validate[VariableInitialisation](ext, env);

CompileEnv[VariableBinding] env;
name: STRING [] Name(TypedIdentifier);
if not ext.strict and getRegionalFrame(env) [] PACKAGE [] PARAMETERFRAME and not immutable and attr = none and Plain(TypedIdentifier) then
qname: QUALIFIEDNAME [] public::name;
Multiname[VariableBinding] {qname};
CompileVar[VariableBinding] defineHoistedVar(env, name, undefined)
else
a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
if a.dynamic then
throw an AttributeError exception — a variable definition cannot have the dynamic attribute
end if;
if a.prototype then
throw an AttributeError exception — a variable definition cannot have the prototype attribute
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 an AttributeError exception — non-class-member variables cannot have a static, virtual, or final attribute
end if;
end if;
case memberMod of
{none, static} do
initialiser: INITIALISEROPT [] Initialiser[VariableInitialisation];
if noInitialiser and initialiser ≠ none then
throw a SyntaxError exception — a for-in statement’s variable definition must not have an initialiser
end if;
proc variableSetup(): CLASSOPT
  type: CLASSOPT [] SetupAndEval(TypedIdentifier)(env);
  Setup[VariableInitialisation](0);
return type
end proc;
v: VARIABLE [] new VARIABLE[]value: none, immutable: immutable, setup: variableSetup,
  initialiser: initialiser, initialiserEnv: env[]
multiname: MULTINAME [] defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit,
  readWrite, v);
Multiname[VariableBinding] multiname;
CompileVar[VariableBinding] v;
{virtual, final} do
  note not noInitialiser;
c: CLASS [] env[0];
v: INSTANCEVARIABLE [] new INSTANCExVARIABLE[]final: memberMod = final,
  enumerable: a.enumerable, immutable: immutable[]
OverriddenVar[VariableBinding] defineInstanceMember(c, ext, name, a.namespaces, a.overrideMod,
  a.explicit, v);
CompileVar[VariableBinding] v
end case
end if
end proc;

Validate[VariableInitialisation\textsuperscript{b}] (\textit{ctx}: \texttt{CONTEXT}, \textit{env}: \texttt{ENVIRONMENT}) propagates the call to \texttt{Validate} to every nonterminal in the expansion of VariableInitialisation\textsuperscript{b}.

Validate[VariableInitialiser\textsuperscript{b}] (\textit{ctx}: \texttt{CONTEXT}, \textit{env}: \texttt{ENVIRONMENT}) propagates the call to \texttt{Validate} to every nonterminal in the expansion of VariableInitialiser\textsuperscript{b}.

Name[TypedIdentifier\textsuperscript{b}]: \texttt{STRING};
\begin{verbatim}
Name[TypedIdentifier\textsuperscript{b}][ Identifier ] = Name[Identifier];
Name[TypedIdentifier\textsuperscript{b}][ Identifier : TypeExpression\textsuperscript{b} ] = Name[Identifier];
\end{verbatim}

Plain[TypedIdentifier\textsuperscript{b}]: \texttt{BOOLEAN};
\begin{verbatim}
Plain[TypedIdentifier\textsuperscript{b}][ Identifier ] = true;
Plain[TypedIdentifier\textsuperscript{b}][ Identifier : TypeExpression\textsuperscript{b} ] = false;
\end{verbatim}

\texttt{proc} Validate[TypedIdentifier\textsuperscript{b}] (\textit{ctx}: \texttt{CONTEXT}, \textit{env}: \texttt{ENVIRONMENT})
\begin{verbatim}
[TypedIdentifier\textsuperscript{b}][ Identifier ] do nothing;
[TypedIdentifier\textsuperscript{b}][ Identifier : TypeExpression\textsuperscript{b} ] do
Validate[TypeExpression\textsuperscript{b}](\textit{ctx}, \textit{env})
\end{verbatim}
\texttt{end proc;}

Setup

\texttt{proc} Setup[VariableDefinition\textsuperscript{b}][ VariableDefinitionKind VariableBindingList\textsuperscript{b} ] ()
\begin{verbatim}
Setup[VariableBindingList\textsuperscript{b}]()
\end{verbatim}
\texttt{end proc;}

Setup[VariableBindingList\textsuperscript{b} ] () propagates the call to \texttt{Setup} to every nonterminal in the expansion of VariableBindingList\textsuperscript{b}.
proc Setup[VariableBinding^2[] TypedIdentifier^2[] VariableInitialisation^3[]] ()
  env: ENVIRONMENT[] CompileEnv[VariableBinding^3[]];
v: VARIABLE[] DYNAMIC_VAR[] INSTANCE_VARIABLE[] CompileVar[VariableBinding^3[]];
case v of
  VARIABLE do
    setupVariable(v);
    if not v.immutable then
      defaultValue: OBJECTOPT[] v.type.defaultValue;
      if defaultValue = none then
        throw an UninitializedError exception — Cannot declare a mutable variable of type Never
      end if;
      v.value[] defaultValue
    end if;
  DYNAMIC_VAR do Setup[VariableInitialisation^3[]];
  INSTANCE_VARIABLE do
    t: CLASSOPT[] SetupAndEval[TypedIdentifier^2[]](env);
    if t = none then
      overriddenVar: INSTANCE_VARIABLEOPT[] OverriddenVar[VariableBinding^3[]];
      if overriddenVar ≠ none then t[].overriddenVar.type
      else t[].Object
    end if;
    v.type[] t;
    Setup[VariableInitialisation^3[]];
    initialiser: INITIALISEROPT[] Initialiser[VariableInitialisation^3[]];
    defaultValue: OBJECTOPT[] none;
    if initialiser ≠ none then defaultValue[] initialiser(env, compile)
  elsif not v.immutable then
    defaultValue[] t.defaultValue;
    if defaultValue = none then
      throw an UninitializedError exception — Cannot declare a mutable instance variable of type Never
    end if;
    v.defaultValue[] defaultValue
  end case
end proc;

Setup[VariableInitialisation^3[]] () propagates the call to Setup to every nonterminal in the expansion of VariableInitialisation^3[].

Setup[VariableInitialiser^3[]] () propagates the call to Setup to every nonterminal in the expansion of VariableInitialiser^3[].

Evaluation

proc Eval[VariableDefinition^3[] VariableDefinitionKind VariableBindingList^3[]]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
  Eval[VariableBindingList^3[]](env);
  return d
end proc;

Eval[VariableBindingList^3[]] (env: ENVIRONMENT) propagates the call to Eval to every nonterminal in the expansion of VariableBindingList^3[].
proc Eval[VariableBinding⁰ [] TypedIdentifier⁰ VariableInitialisation⁰] (env: Environment)

case CompileVar[VariableBinding⁰] of
  VARIABLE do
    innerFrame: NonWithFrame [] env[0];
    members: LocalMember[{} | b.content | b innerFrame.localBindings such that
                      b.qname [] Multiname[VariableBinding⁰]];)
    note The members set consists of exactly one VARIABLE element because innerFrame was constructed with that VARIABLE inside Validate.
    v: VARIABLE [] the one element of members;
    initialiser: Initialiser [] {none, busy} [] v.initialiser;
    case initialiser of
      {none} do nothing;
      {busy} do throw a ReferenceError exception;
    END INITIALISER do
      v.initialiser [] busy;
      value: Object [] initialiser(v.initialiserEnv, run);
      writeVariable(v, value, true)
    end case;
  DYNAMICVAR do
    initialiser: InitialiserOpt [] Initialiser[VariableInitialisation⁰];
    if initialiser ≠ none then
      value: Object [] initialiser(env, run);
      lexicalWrite(env, Multiname[VariableBinding⁰], value, false, run)
    end if;
    INSTANCE VARIABLE do nothing
  end case;
end proc;

proc WriteBinding[VariableBinding⁰ [] TypedIdentifier⁰ VariableInitialisation⁰] (env: Environment, newValue: Object)

case CompileVar[VariableBinding⁰] of
  VARIABLE do
    innerFrame: NonWithFrame [] env[0];
    members: LocalMember[{} | b.content | b innerFrame.localBindings such that
                      b.qname [] Multiname[VariableBinding⁰]];)
    note The members set consists of exactly one VARIABLE element because innerFrame was constructed with that VARIABLE inside Validate.
    v: VARIABLE [] the one element of members;
    writeVariable(v, newValue, false);
  DYNAMICVAR do
    lexicalWrite(env, Multiname[VariableBinding⁰], newValue, false, run)
  end case;
end proc;

Initialiser[VariableInitialisation⁰]: InitialiserOpt;
  Initialiser[VariableInitialisation⁰] «empty» = none;
  Initialiser[VariableInitialisation⁰] = VariableInitialiser⁰ = Eval[VariableInitialiser⁰];

  [VariableInitialiser⁰ [] AssignmentExpression⁰] do
    return readReference(Eval[AssignmentExpression⁰](env, phase), phase);
  [VariableInitialiser⁰ [] NonexpressionAttribute] do
    return Eval[NonexpressionAttribute](env, phase);
[VariableInitialiser\[ AttributeCombination\] do
  return Eval[AttributeCombination]\(\text{env, phase}\)
end proc;

proc SetupAndEval[TypedIdentifier\[ Phase\]] \(\text{env: ENVIRONMENT: CLASSOPT}\)
  [TypedIdentifier\[ Identifier\] do return none;
  [TypedIdentifier\[ Identifier : TypeExpression\] do
    return SetupAndEval[TypeExpression\[ Phase\]]\(\text{env}\)
end proc;

15.2 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.

\[
\text{SimpleVariableDefinition \[ var UntypedVariableBindingList}
\]

\[
\text{UntypedVariableBindingList \[}
\]

\[
\text{UntypedVariableBinding \[ Identifier VariableInitialisation}\allowin
\]

Validation

proc Validate[SimpleVariableDefinition \[ var UntypedVariableBindingList\]] \(\text{ext: CONTEXT, env: ENVIRONMENT}\)
if ext.\; strict or getRegionalFrame\(\text{env}\) \[ PACKAGE \[ PARAMETERFRAME \] then
  throw a SyntaxError exception — a variable may not be defined in a substatement except inside a non-strict function or non-strict top-level code; to fix this error, place the definition inside a block
end if;
Validate[UntypedVariableBindingList]\(\text{ext, env}\)
end proc;

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

proc Validate[UntypedVariableBinding \[ Identifier VariableInitialisation]}\allowin
  Validate[VariableInitialisation]}\allowin]\(\text{ext, env}\);
  defineHoistedVar\(\text{env, Name[Identifier], undefined}\)
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]}\allowin
  Setup[VariableInitialisation]}\allowin]()
end proc;
Evaluation

```plaintext
proc Eval[SimpleVariableDefinition] var UntypedVariableBindingList (env: ENVIRONMENT, d: OBJECT): OBJECT
    Eval[UntypedVariableBindingList](env);
    return d
end proc;

proc Eval[UntypedVariableBindingList] (env: ENVIRONMENT)
    [UntypedVariableBindingList] UntypedVariableBinding do
        Eval[UntypedVariableBinding](env);
    end proc;

test Eval[UntypedVariableBinding] Identifier VariableInitialisation (env: ENVIRONMENT)
    initialiser: INITIALISER OPT Initialiser[VariableInitialisation]
    if initialiser ≠ none then
        value: OBJECT initialiser(env, run);
        qname: QUALIFIED NAME public::(Name[Identifier]);
        lexicalWrite(env, {qname}, value, false, run)
    end if
end proc;

15.3 Function Definition

Syntax

```plaintext
FunctionDefinition function FunctionName FunctionCommon

FunctionName Identifier
    | get [no line break] Identifier
    | set [no line break] Identifier

FunctionCommon (Parameters) Result Block
```

Validation

```plaintext
OverriddenMember[FunctionDefinition]: INSTANCEMEMBEROPT;
```

name: STRING [] Name[FunctionName];

handling: HANDLING [] Handling[FunctionName];

case handling of

{} normal do

kind: STATICFUNCTIONKIND;

if unchecked then kind [] uncheckedFunction
elsif a.prototype then kind [] prototypeFunction
else kind [] plainFunction
end if;

f: SIMPLEINSTANCE [] UNINSTANTIATEDFUNCTION

ValidateStaticFunction[FunctionCommon](cxt, env, kind);

if preinst then instantiateFunction(f, env) end if;

if hoisted then defineHoistedVar(env, name, f)
else

v: VARIABLE [] new VARIABLE[type: Function, value: f, immutable: true, setup: none, initialiser: none]

defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
end if;

{} get, set do

if a.prototype then

throw an AttributeError exception — a getter or setter cannot have the prototype attribute
end if;

note not (unchecked or hoisted);

Validate[FunctionCommon](cxt, env, plainFunction, handling);

boundEnv: ENVIRONMENTOPT [] none;

if preinst then boundEnv [] env end if;

case handling of

{} do

getter: GETTER [] new GETTER[call: EvalStaticGet[FunctionCommon], env: boundEnv]

defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, read, getter);

{} set do

setter: SETTER [] new SETTER[call: EvalStaticSet[FunctionCommon], env: boundEnv]

defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, write, setter)

end case

OverriddenMember[FunctionDefinition] [] none

end case

end case

end proc;
proc ValidateInstance[FunctionDefinition [] function FunctionName FunctionCommon]
   (ext: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDATTRIBUTE, final: BOOLEAN)
if a.prototype then
   throw an AttributeError exception — an instance method cannot have the prototype attribute
end if;
handling: HANDLING [] Handling[FunctionName];
Validate[FunctionCommon](ext, env, instanceFunction, handling);
m: INSTANCEMEMBER;
case handling of
   {normal} do
      m [] new INSTANCEMETHOD[]final: final, enumerable: a.enumerable,
         signature: CompileFrame[FunctionCommon], call: EvalInstanceCall[FunctionCommon][]
   {get} do
      m [] new INSTANCEGETTER[]final: final, enumerable: a.enumerable,
         signature: CompileFrame[FunctionCommon], call: EvalInstanceGet[FunctionCommon][]
   {set} do
      m [] new INSTANCESETTER[]final: final, enumerable: a.enumerable,
         signature: CompileFrame[FunctionCommon], call: EvalInstanceSet[FunctionCommon][]
end case;
OverriddenMember[FunctionDefinition] [] defineInstanceMember(c, ext, Name[FunctionName], a.namespaces, a.overrideMod, a.explicit, m)
end proc;

proc ValidateConstructor[FunctionDefinition [] function FunctionName FunctionCommon]
   (ext: CONTEXT, env: ENVIRONMENT, c: CLASS, a: COMPOUNDATTRIBUTE)
if a.prototype then
   throw an AttributeError exception — a class constructor cannot have the prototype attribute
end if;
if Handling[FunctionName] [] {get, set} then
   throw a SyntaxError exception — a class constructor cannot be a getter or a setter
end if;
Validate[FunctionCommon](ext, env, constructorFunction, normal);
if c.init ≠ none then
   throw a DefinitionError exception — duplicate constructor definition
end if;
c.init [] EvalInstanceInit[FunctionCommon];
OverriddenMember[FunctionDefinition] [] none
end proc;
proc Validate[FunctionDefinition [] function FunctionName FunctionCommon]
  (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTFALSE)
  a: COMPOUNDATTRIBUTE toCompoundAttribute(attr);
  if a.dynamic then
    throw an AttributeError exception — a function cannot have the dynamic attribute
  end if;
  frame: FRAME env[0];
  if frame CLASS then
    note preinst;
    case a.memberMod of
      {static} do
        ValidateStatic[FunctionDefinition](ext, env, preinst, a, false, false);
      {none} do
        if Name[FunctionName] = frame.name then
          ValidateConstructor[FunctionDefinition](ext, env, frame, a)
        else ValidateInstance[FunctionDefinition](ext, env, frame, a, false)
        end if;
      {virtual} do ValidateInstance[FunctionDefinition](ext, env, frame, a, false);
      {final} do ValidateInstance[FunctionDefinition](ext, env, frame, a, true)
    end case
  else
    if a.memberMod ≠ none then
      throw an AttributeError exception — non-class-member functions cannot have a static, virtual, or final attribute
    end if;
    unchecked: BOOLEAN not ext.strict and Handling[FunctionName] = normal and Plain[FunctionCommon];
    hoisted: BOOLEAN unchecked and attr = none and
      (frame PACKAGE or (frame LOCALFRAME and env[1] PARAMETERFRAME));
    ValidateStatic[FunctionDefinition](ext, env, preinst, a, unchecked, hoisted)
  end if;
end proc;

Handling[FunctionName]: HANDLING;
  Handling[FunctionName [] Identifier] = normal;
  Handling[FunctionName [] get [no line break] Identifier] = get;
  Handling[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];

Plain[FunctionCommon [] ( Parameters ) Result Block]: BOOLEAN = Plain[Parameters] and Plain[Result];

CompileEnv[FunctionCommon]: ENVIRONMENT;

CompileFrame[FunctionCommon]: PARAMETERFRAME;
proc Validate[FunctionCommon] (Parameters) Result Block

(cxt: CONTEXT, env: ENVIRONMENT, kind: FUNCTION_KIND, handling: HANDLING)
localCxt: CONTEXT new CONTEXT[]strict: cxt.strict, openNamespaces: cxt.openNamespaces[]
superconstructorCalled: BOOLEAN[]kind ≠ constructorFunction;
compileFrame: PARAMETER_FRAME[]new PARAMETER_FRAME[]localBindings: {}, kind: kind, handling: handling,
callsSuperconstructor: false, superconstructorCalled: superconstructorCalled, this: none, parameters: [],
rest: none[]
compileEnv: ENVIRONMENT[]|compileFrame| ⊙ env;
CompileFrame[FunctionCommon][]compileFrame;
CompileEnv[FunctionCommon][]compileEnv;
if kind = uncheckedFunction then defineHoistedVar(compileEnv, "arguments", undefined)
end if;
Validate[Parameters](localCxt, compileEnv, compileFrame);
Validate[Result](localCxt, compileEnv);
Validate[Block](localCxt, compileEnv, JUMP_TARGETS[]breakTargets: {}, continueTargets: {}[]false)
end proc;

proc ValidateStaticFunction[FunctionCommon] (Parameters) Result Block

(cxt: CONTEXT, env: ENVIRONMENT, kind: STATIC_FUNCTION_KIND): UNINSTANTIATED_FUNCTION
Validate[FunctionCommon](cxt, env, kind, normal);
length: INTEGER[]ParameterCount[Parameters];
case kind of
  {plainFunction} do
    return new UNINSTANTIATED_FUNCTION[]type: Function, buildPrototype: false, length: length,
call: EvalStaticCall[FunctionCommon], construct: none, instantiations: {}[]
  {uncheckedFunction, prototypeFunction} do
    return new UNINSTANTIATED_FUNCTION[]type: PrototypeFunction, buildPrototype: true, length: length,
call: EvalStaticCall[FunctionCommon], construct: EvalPrototypeConstruct[FunctionCommon],
instantiations: {}[]
end case
end proc;
Setup

  overriddenMember: INSTANCEMETHODOPT INSTANCEGETTER INSTANCESETTER do
    SetupOverride[FunctionCommon](overriddenMember.signature);
  end case

  case Handling[FunctionName] of
    { normal } do
      This cannot happen because ValidateInstance already ensured that a function cannot override an instance variable.
    { get } do
      overriddenSignature [ new PARAMETERFRAME()[localBindings: {}, kind: instanceFunction, handling: get, callsSuperconstructor: false, superconstructorCalled: false, this: none, parameters: [ ], rest: none, returnType: overriddenMember.type[]] do
        v: VARIABLE [ new VARIABLE()[type: overriddenMember.type, value: none, immutable: false, setup: none, initialiser: none[]] ] [ PARAMETER[ ] do
            end case
        end overriddenSignature
      end overriddenSignature
    end case
  end case
end proc;

proc Setup[FunctionCommon [ Parameters ] Result Block] ( )
  compileEnv: ENVIRONMENT [ CompileEnv[FunctionCommon] ];
  compileFrame: PARAMETERFRAME [ CompileFrame[FunctionCommon] ];
  Setup[Parameters](compileEnv, compileFrame);
  checkAccessorParameters(compileFrame);
  Setup[Result](compileEnv, compileFrame);
  Setup[Block]()
end proc;

proc SetupOverride[FunctionCommon [ Parameters ] Result Block] ( overriddenSignature: PARAMETERFRAME )
  compileEnv: ENVIRONMENT [ CompileEnv[FunctionCommon] ];
  compileFrame: PARAMETERFRAME [ CompileFrame[FunctionCommon] ];
  SetupOverride[Parameters](compileEnv, compileFrame, overriddenSignature);
  checkAccessorParameters(compileFrame);
  SetupOverride[Result](compileEnv, compileFrame, overriddenSignature);
  Setup[Block]()
end proc;
Evaluation

proc EvalStaticCall[FunctionCommon] ( Parameters ) Result Block

(this: OBJECT; f: SIMPLEx.INSTANCE, args: OBJECT[], phase: PHASE): OBJECT

note The check that phase ≠ compile also ensures that Setup has been called.

if phase = compile then
  throw a ConstantError exception — constant expressions cannot call user-defined functions
end if;

runtimeEnv: ENVIRONMENT f.env;
runtimeThis: OBJECTOPT none;
compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
if compileFrame.kind { uncheckedFunction, prototypeFunction } then
  if this PRIMITIVEOBJECT then runtimeThis getPackageFrame(runtimeEnv)
else runtimeThis this
end if;

runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
assignArguments(runtimeFrame, f, args, phase);
result: OBJECT;
try
Eval[Block](runtimeFrame ⊕ runtimeEnv, undefined); result undefined
catch x: SEMANTICEXCEPTION do
  if x RETURN then result x.value else throw x end if
end try;
coercedResult: OBJECT runtimeFrame.returnType.implicitCoerce(result, false);
return coercedResult
end proc;

proc EvalStaticGet[FunctionCommon] ( Parameters ) Result Block

(runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT

note The check that phase ≠ compile also ensures that Setup has been called.

if phase = compile then
  throw a ConstantError exception — constant expressions cannot call user-defined getters
end if;

compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, none);
assignArguments(runtimeFrame, none, [], phase);
result: OBJECT;
try
Eval[Block](runtimeFrame ⊕ runtimeEnv, undefined);
throw a SyntaxError exception — a getter must return a value and may not return by falling off the end of its code
catch x: SEMANTICEXCEPTION do
  if x RETURN then result x.value else throw x end if
end try;
coercedResult: OBJECT runtimeFrame.returnType.implicitCoerce(result, false);
return coercedResult
end proc;
proc EvalStaticSet[FunctionCommon [] (Parameters) Result Block]
    (newValue: OBJECT, runtimeEnv: ENVIRONMENT, phase: PHASE)

    note The check that phase ≠ compile also ensures that Setup has been called.

    if phase = compile then
        throw a ConstantError exception — constant expressions cannot call setters
    end if;

    compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
    runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, runtimeEnv, none);
    assignArguments(runtimeFrame, none, [newValue], phase);

    try Eval[Block]((runtimeFrame) + runtimeEnv, undefined)
    catch x: SEMANTICEXCEPTION do if x [] RETURN then throw x end if
    end try;

end proc;

proc EvalInstanceCall[FunctionCommon [] (Parameters) Result Block]
    (this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT

    note The check that phase ≠ compile also ensures that Setup has been called.

    if phase = compile then
        throw a ConstantError exception — constant expressions cannot call user-defined functions
    end if;

    note Class frames are always preinstantiated, so the run environment is the same as compile environment.

    env: ENVIRONMENT CompileEnv[FunctionCommon []];
    compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
    runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, env, this);
    assignArguments(runtimeFrame, none, args, phase);

    result: OBJECT;

    try Eval[Block]((runtimeFrame) + env, undefined); result [] undefined
    catch x: SEMANTICEXCEPTION do
        if x [] RETURN then result [] x.value else throw x end if
    end try;

    coercedResult: OBJECT runtimeFrame.returnType.implicitCoerce(result, false);

    return coercedResult

end proc;

proc EvalInstanceGet[FunctionCommon [] (Parameters) Result Block] (this: OBJECT, phase: PHASE): OBJECT

    note The check that phase ≠ compile also ensures that Setup has been called.

    if phase = compile then
        throw a ConstantError exception — constant expressions cannot call user-defined getters
    end if;

    note Class frames are always preinstantiated, so the run environment is the same as compile environment.

    env: ENVIRONMENT CompileEnv[FunctionCommon []];
    compileFrame: PARAMETERFRAME CompileFrame[FunctionCommon];
    runtimeFrame: PARAMETERFRAME instantiateParameterFrame(compileFrame, env, this);
    assignArguments(runtimeFrame, none, [], phase);

    result: OBJECT;

    try
        Eval[Block]((runtimeFrame) + env, undefined);
        throw a SyntaxError exception — a getter must return a value and may not return by falling off the end of its code
    catch x: SEMANTICEXCEPTION do
        if x [] RETURN then result [] x.value else throw x end if
    end try;

    coercedResult: OBJECT runtimeFrame.returnType.implicitCoerce(result, false);

    return coercedResult

end proc;
proc **EvalInstanceSet**[FunctionCommon [] (Parameters) Result Block]

(this: Object, newValue: Object, phase: PHASE)

note The check that phase ≠ compile also ensures that Setup has been called.

if phase = compile then
    throw a ConstantError exception — constant expressions cannot call setters
end if;

note Class frames are always preinstantiated, so the run environment is the same as compile environment.

env: ENVIRONMENT = CompileEnv[FunctionCommon];

compileFrame: PARAMETERFRAME = CompileFrame[FunctionCommon];

runtimeFrame: PARAMETERFRAME = instantiateParameterFrame(compileFrame, env, this);

assignArguments(runtimeFrame, none, [newValue], phase);

try Eval[Block](runtimeFrame ⊕ env, undefined)
catch x: SEMANTICEXCEPTION do if x = RETURN then throw x end if
end try;

end proc;

proc **EvalInstanceInit**[FunctionCommon [] (Parameters) Result Block]

(this: SIMPLEINSTANCE, args: Object[], phase: {run})

note Class frames are always preinstantiated, so the run environment is the same as compile environment.

env: ENVIRONMENT = CompileEnv[FunctionCommon];

compileFrame: PARAMETERFRAME = CompileFrame[FunctionCommon];

runtimeFrame: PARAMETERFRAME = instantiateParameterFrame(compileFrame, env, this);

assignArguments(runtimeFrame, none, args, phase);

if not runtimeFrame.callsSuperconstructor then
    c: CLASS = getEnclosingClass(env);
    callInit(this, c.super, [], run);
    runtimeFrame.superconstructorCalled = true
end if;

try Eval[Block](runtimeFrame ⊕ env, undefined)
catch x: SEMANTICEXCEPTION do if x = RETURN then throw x end if
end try;

if not runtimeFrame.superconstructorCalled then
    throw an UninitializedError exception — the superconstructor must be called before returning normally from a constructor
end if;

end proc;
proc EvalPrototypeConstruct[FunctionCommon [] (Parameters) Result Block]
  (f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT

note The check that phase ≠ compile also ensures that Setup has been called.
if phase = compile then
  throw a ConstantError exception — constant expressions cannot call user-defined prototype constructors
end if;
runtimeEnv: ENVIRONMENT [] f.env;
super: OBJECT [] dotRead(f, {public::"prototype"}, phase);
if super œ {null, undefined} then super œ objectPrototype
elsif not Prototype.is(super) then
  throw a TypeError exception — the prototype must have type Prototype
end if;
result: OBJECT [] createSimpleInstance(Prototype, super, none, none, none);
compileFrame: PARAMETERFRAME [] CompileFrame[FunctionCommon];
runtimeFrame: PARAMETERFRAME [] instantiateParameterFrame(compileFrame, runtimeEnv, o);
assignArguments(runtimeFrame, f, args, phase);
result: OBJECT;
try Eval[Block](runtimeFrame) ⊕ runtimeEnv, undefined); result œ undefined
catch x: SEMANTICEXCEPTION do
  if x œ RETURN then result œ x.value else throw x end if
end try;
coercedResult: OBJECT [] runtimeFrame.returnType.implicitCoerce(result, false);
if coercedResult œ PRIMITIVEOBJECT then return o else return coercedResult end if
end proc;

proc checkAccessorParameters(frame: PARAMETERFRAME)
parameters: PARAMETER[] [] frame.parameters;
rest: VARIABLEOPT [] frame.rest;
case frame.handling of
  {normal} do nothing;
  {get} do
    if parameters œ [] or rest œ none then
      throw a SyntaxError exception — a getter cannot take any parameters
    end if;
  {set} do
    if [parameters] œ 1 or rest œ none then
      throw a SyntaxError exception — a setter must take exactly one parameter
    end if;
    if parameters[0].default œ none then
      throw a SyntaxError exception — a setter’s parameter cannot be optional
    end if
  end case
end proc;
proc assignArguments(runtimeFrame: ParameterFrame, f: SimpleInstance[] {none}, args: Object[],
    phase: [run])

This procedure performs a number of checks on the arguments, including checking their count, names, and values. Although this procedure performs these checks in a specific order for expository purposes, an implementation may perform these checks in a different order, which could have the effect of reporting a different error if there are multiple errors. For example, if a function only allows between 2 and 4 arguments, the first of which must be a Number and is passed five arguments the first of which is a String, then the implementation may throw an exception either about the argument count mismatch or about the type coercion error in the first argument.

argumentsObject: ObjectOpt[] {none};
if runtimeFrame.kind = uncheckedFunction then
    argumentsObject [] Array.construct([], phase);
createDynamicProperty(argumentsObject, public:“callee”, false, false, f);
nArgs: INTEGER | args;
if nArgs > arrayLimit then throw a RangeError exception end if;
doWrite(argumentsObject, [arrayPrivate:”length”], nArgsalong: phase)
end if;
restObject: ObjectOpt[] {none};
rest: VARIABLE [none] runtimeFrame.rest;
if rest ≠ none then restObject [] Array.construct([], phase) end if;
parameters: Parameter[] runtimeFrame.parameters;
i: INTEGER | 0;
j: INTEGER | 0;
for each arg [] args do
    if i < [parameters] then
        parameter: Parameter [] parameters[i];
v: DynamicVar [] Variable [] parameter.var;
writeLocalMember(v, arg, phase);
if argumentsObject ≠ none then
    note Create an alias of v as the i-th entry of the arguments object.
    note v DynamicVar;
    qname: QualifiedName toQualifiedName(ulong, phase);
    argumentsObject.localBindings [] argumentsObject.localBindings [] {LocalBinding}[name: qname,
        accesses: readWrite, content: v, explicit: false, enumerable: false]}
end if
elsif restObject ≠ none then
    if j ≥ arrayLimit then throw a RangeError exception end if;
    indexWrite(restObject, j, arg, phase);
    note argumentsObject = none because a function can't have both a rest parameter and an arguments object.
    j [] j + 1
elsif argumentsObject ≠ none then indexWrite(argumentsObject, i, arg, phase)
else
    throw an ArgumentError exception — more arguments than parameters were supplied, and the called function does not have a . . . parameter and is not unchecked.
end if;
i [] i + 1
end for each;
while i < [parameters] do
    parameter: Parameter [] parameters[i];
default: ObjectOpt [] parameter.default;
if default = none then
    if argumentsObject ≠ none then default [] undefined
else
    throw an ArgumentError exception — fewer arguments than parameters were supplied, and the called function does not supply default values for the missing parameters and is not unchecked.
end if
end if;
writeLocalMember(parameter:var, default, phase);

end while

end proc;

Syntax

Parameters []

«empty»

| NonemptyParameters

NonemptyParameters []

ParameterInit

| ParameterInit, NonemptyParameters

| RestParameter

Parameter [] ParameterAttributes TypedIdentifier

ParameterAttributes []

«empty»

| const

ParameterInit []

Parameter

| Parameter = AssignmentExpression

RestParameter []

... . . ParameterAttributes Identifier

Result []

«empty»

| : TypeExpression

Validation

Plain[Parameters]: BOOLEAN;

Plain[Parameters «empty»] = true;

Plain[Parameters NonemptyParameters] = Plain[NonemptyParameters];

ParameterCount[Parameters]: INTEGER;

ParameterCount[Parameters «empty»] = 0;

ParameterCount[Parameters NonemptyParameters] = ParameterCount[NonemptyParameters];

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

Plain[NonemptyParameters]: BOOLEAN;

Plain[NonemptyParameters ParameterInit] = Plain[ParameterInit];

Plain[NonemptyParameters, NonemptyParameters, NonemptyParameters] = Plain[ParameterInit] and Plain[NonemptyParameters];

Plain[NonemptyParameters RestParameter] = false;
ParameterCount[NonemptyParameters]: INTEGER;
ParameterCount[NonemptyParameters ⦃ ParameterInit ⦄ Parameter] = 1;
ParameterCount[NonemptyParameters ⦃ ParameterInit ⦄ NonemptyParameters ⦄ NonemptyParameters ⦄ RestParameter] = 0;

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

Name[Parameter ⦃ ParameterAttributes TypedIdentifier allowIn ⦄]: STRING = Name[TypedIdentifier allowIn];

Plain[Parameter ⦃ ParameterAttributes TypedIdentifier allowIn ⦄]: BOOLEAN
Plain[TypedIdentifier allowIn] and not HasConst[ParameterAttributes];

CompileVar[Parameter]: DYNAMICVAR ⦃ VARIABLE;

| proc Validate[Parameter ⦃ ParameterAttributes TypedIdentifier allowIn ⦄] (ctx: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME ⦃ LOCALFRAME) Validate[TypedIdentifier allowIn](ctx, env); immutable: BOOLEAN ⦃ HasConst[ParameterAttributes]; name: STRING ⦃ Name[TypedIdentifier allowIn]; v: DYNAMICVAR ⦃ VARIABLE;
| if compileFrame ⦃ PARAMETERFRAME and compileFrame.kind = uncheckedFunction then
| note not immutable;
| v ⦃ defineHoistedVar(env, name, undefined)
| else
| v ⦃ new VARIABLE ⦃ value: none, immutable: immutable, setup: none, initialiser: none ⦄
| defineLocalMember(env, name, {public}, none, false, readWrite, v)
| end if;
| CompileVar[Parameter] ⦃ v
end proc;

HasConst[ParameterAttributes]: BOOLEAN;
HasConst[ParameterAttributes ⦃ «empty»] = false;
HasConst[ParameterAttributes ⦃ const] = true;

Plain[ParameterInit]: BOOLEAN;
Plain[ParameterInit ⦃ Parameter ⦄] = Plain[Parameter];
Plain[ParameterInit ⦃ Parameter = AssignmentExpression allowIn] = false;

| proc Validate[ParameterInit] (ctx: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
| [ParameterInit ⦃ Parameter] do Validate[Parameter](ctx, env, compileFrame);
| [ParameterInit ⦃ Parameter = AssignmentExpression allowIn] do
| Validate[Parameter](ctx, env, compileFrame);
| Validate[AssignmentExpression allowIn](ctx, env)
end proc;

| proc Validate[RestParameter] (ctx: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
| [RestParameter ⦃ . . . ] do
| note compileFrame.kind ≠ uncheckedFunction;
| v: VARIABLE ⦃ new VARIABLE ⦃ type: Array, value: none, immutable: true, setup: none, initialiser: none ⦄
| compileFrame.rest ⦃ v
end proc;
[RestParameter [] ... ParameterAttributes Identifier] do
  note compileFrame.kind ≠ uncheckedFunction;
  v: VARIABLE [] new VARIABLE[] type: Array, value: none, immutable: HasConst[ParameterAttributes],
  setup: none, initialiser: none[]
  compileFrame.rest [] v;
  name: STRING [] Name[Identifier];
  defineLocalMember(env, name, {public}, none, false, readWrite, v)
end proc;

Plain[Result]: BOOLEAN;
Plain[Result] «empty» = true;
Plain[Result] : TypeExpression"lower" = false;

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

Setup

Setup[Parameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME) propagates the call to Setup to every nonterminal in the expansion of Parameters.

proc SetupOverride[Parameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenSignature: PARAMETERFRAME)
[Parameters [] «empty»] do
  if overriddenSignature.parameters ≠ [] or overriddenSignature.rest ≠ none then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
[Parameters NonemptyParameters] do
  SetupOverride[NonemptyParameters](compileEnv, compileFrame, overriddenSignature, overriddenSignature.parameters)
end proc;

proc Setup[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[NonemptyParameters [] ParameterInit] do
  Setup[ParameterInit](compileEnv, compileFrame);
[NonemptyParameters0 [] ParameterInit, NonemptyParameters] do
  Setup[ParameterInit](compileEnv, compileFrame);
  Setup[NonemptyParameters](compileEnv, compileFrame);
[NonemptyParameters [] RestParameter] do nothing
end proc;

proc SetupOverride[NonemptyParameters] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenSignature: PARAMETERFRAME, overriddenParameters: PARAMETER[])
[NonemptyParameters [] ParameterInit] do
  if overriddenParameters = [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]);
  if |overriddenParameters| ≠ 1 or overriddenSignature.rest ≠ none then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
[NonemptyParameters0 [] ParameterInit, NonemptyParameters1] do
  if overriddenParameters = [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  SetupOverride[ParameterInit](compileEnv, compileFrame, overriddenParameters[0]);
  SetupOverride[NonemptyParameters1](compileEnv, compileFrame, overriddenSignature,
    overriddenParameters[1 ...]);
[NonemptyParameters [] RestParameter] do
  if overriddenParameters ≠ [] then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  overriddenRest: VARIABLE [] {none} overriddenSignature.rest;
  if overriddenRest = none or overriddenRest.type ≠ Array then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if
end proc;

proc Setup[Parameter [] ParameterAttributes TypedIdentifier] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME [] LOCALFRAME, default: OBJECTOPT)
if compileFrame [] PARAMETERFRAME and default = none and
  (some p2 [] compileFrame.parameters satisfies p2.default ≠ none) then
  throw a SyntaxError exception — a required parameter cannot follow an optional one
end if;
v: DYNAMICVAR [] VARIABLE [] CompileVar[Parameter];
case v of
  DYNAMICVAR do nothing;
  VARIABLE do
    type: CLASSOPT [] SetupAndEval[TypedIdentifier](compileEnv);
    if type = none then type [] Object end if;
  end case;
if compileFrame [] PARAMETERFRAME then
  p: PARAMETER [] PARAMETER[] var: v, default: default[]
  compileFrame.parameters [] compileFrame.parameters ⊕ [p]
end if
end proc;

newDefault: OBJECTOPT [] default;
if newDefault = none then newDefault [] overriddenParameter.default end if;
if default = none and (some p2 [] compileFrame.parameters satisfies p2.default ≠ none) then
  throw a SyntaxError exception — a required parameter cannot follow an optional one
end if;
v: DYNAMICVAR [] VARIABLE [] CompileVar[Parameter];
note v [] DYNAMICVAR;
  type: CLASSOPT [] SetupAndEval[TypedIdentifier](compileEnv);
  if type = none then type [] Object end if;
  if type ≠ overriddenParameter.var.type then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  v.type [] type;
  p: PARAMETER [] PARAMETER[] var: v, default: newDefault[]
  compileFrame.parameters [] compileFrame.parameters ⊕ [p]
end proc;
proc Setup[ParameterInit] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[ParameterInit [] Parameter] do Setup[Parameter](compileEnv, compileFrame, none);
[ParameterInit [] Parameter = AssignmentExpressionallowin] do
  Setup[AssignmentExpressionallowin]()
  default: OBJECT [] readReference(Eval[AssignmentExpressionallowin](compileEnv, compile), compile);
  Setup[Parameter](compileEnv, compileFrame, default)
end proc;

proc SetupOverride[ParameterInit] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenParameter: PARAMETER)
[ParameterInit [] Parameter] do SetupOverride[Parameter](compileEnv, compileFrame, none, overriddenParameter);
[ParameterInit [] Parameter = AssignmentExpressionallowin] do
  Setup[AssignmentExpressionallowin]()
  default: OBJECT [] readReference(Eval[AssignmentExpressionallowin](compileEnv, compile), compile);
  SetupOverride[Parameter](compileEnv, compileFrame, default, overriddenParameter)
end proc;

proc Setup[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME)
[Result [] «empty»] do
  defaultReturnType: CLASS [] Object;
  if cannotReturnValue(compileFrame) then defaultReturnType [] Void end if;
  compileFrame.returnType [] defaultReturnType;
[Result [] : TypeExpressionallowin] do
  if cannotReturnValue(compileFrame) then
    throw a SyntaxError exception — a setter or constructor cannot define a return type
  end if;
  compileFrame.returnType [] SetupAndEval[TypeExpressionallowin](compileEnv)
end proc;

proc SetupOverride[Result] (compileEnv: ENVIRONMENT, compileFrame: PARAMETERFRAME, overriddenSignature: PARAMETERFRAME)
[Result [] «empty»] do compileFrame.returnType [] overriddenSignature.returnType;
[Result [] : TypeExpressionallowin] do
  t: CLASS [] SetupAndEval[TypeExpressionallowin](compileEnv);
  if overriddenSignature.returnType ≠ t then
    throw a DefinitionError exception — mismatch with the overridden method’s signature
  end if;
  compileFrame.returnType [] t
end proc;

15.4 Class Definition

Syntax

ClassDefinition [] class Identifier Inheritance Block

Inheritance []
  «empty»
  | extends TypeExpressionallowin

Validation

Class[ClassDefinition]: CLASS;
proc Validate[ClassDefinition [] class Identifier Inheritance Block]
  (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOptNotFalse)
  if not preinst then
    throw a SyntaxError exception — a class may be defined only in a preinstantiated scope
  end if;
super: CLASS [] Validate[Inheritance](ext, env);
  if not super.complete then
    throw a ConstantError exception — cannot override a class before its definition has been compiled
  end if;
  if super.final then throw a DefinitionError exception — can’t override a final class
  end if;
a: COMPOUNDATTRIBUTE [] toCompoundAttribute(attr);
  if a.prototype then
    throw an AttributeError exception — a class definition cannot have the prototype attribute
  end if;
final: BOOLEAN;
case a.memberMod of
  {none} do final [] false;
  {static} do
    if env[0] [] CLASS then
      throw an AttributeError exception — non-class-member definitions cannot have a static attribute
    end if;
final [] false;
  {final} do final [] true;
  {virtual} do
    throw an AttributeError exception — a class definition cannot have the virtual attribute
  end case;
privateNamespace: NAMESPACE [] new NAMESPACE[name: “private”][]
dynamic: BOOLEAN [] a.dynamic or super.dynamic;
proc cls(o: OBJECT): BOOLEAN
  return isAncestor(c, objectType(o))
end proc;
c.is [] cls;
proc cImplicitCoerce(o: OBJECT, silent: BOOLEAN): OBJECT
  if $o = null or c.is(o) then return $o
  elsif silent then return null
  else throw a TypeError exception
end if;
end proc;
cImplicitCoerce [] cImplicitCoerce;
proc cCall(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  if not c.complete then
    throw a ConstantError exception — cannot coerce to a class before its definition has been compiled
  end if;
  if $args[0] ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  return cImplicitCoerce(args[0], false)
end proc;
c.call [] cCall;
proc cConstruct(args: OBJECT[], phase: PHASE): OBJECT
if not c.complete then
    throw a ConstantError exception — cannot construct an instance of a class before its definition has been compiled
end if;
if phase = compile then
    throw a ConstantError exception — a class constructor call is not a constant expression because it evaluates to a new object each time it is evaluated
end if;
this: SIMPLEINSTANCE [] createSimpleInstance(c, c.prototype, none, none, none);
callInit(this, c, args, phase);
return this end proc;
c.construct [] cConstruct;
Class[ClassDefinition] [] c;
v: VARIABLE [] new VARIABLE[]type: Class, value: c, immutable: true, setup: none, initialiser: none[]
defineLocalMember(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v);
ValidateUsingFrame[Block](cxt, env, JUMPTARGETS[breakTargets: {}, continueTargets: {}]; preinst, c);
if c.init = none then c.init [] super.init end if;
c.complete [] true end proc;

[Inheritance] extends TypeExpression allowed do
    return SetupAndEval[TypeExpression allowed](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;

proc callInit(this: SIMPLEINSTANCE, c: CLASSOPT, args: OBJECT[], phase: {run})
    init: (SIMPLEINSTANCE [] OBJECT[] () [run] [] () {} none) [] none;
    if c ≠ none then init [] c.init end if;
    if init ≠ none then init(this, args, phase)
else
    if args ≠ [] then
        throw an ArgumentError exception — the default constructor does not take any arguments
    end if
end if
end proc;
15.5 Namespace Definition

Syntax

NamespaceDefinition :: namespace Identifier

Validation

proc Validate[ NamespaceDefinition :: namespace Identifier ]
  (ext: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOptNotFalse)
  if not preinst then
    throw a SyntaxError exception — a namespace may be defined only in a preinstantiated scope
  end if;
  a: COMPOUNDATTRIBUTE :: toCompoundAttribute(attr);
  if a.dynamic then
    throw an AttributeError exception — a namespace definition cannot have the dynamic attribute
  end if;
  if a.prototype then
    throw an AttributeError exception — a namespace definition cannot have the prototype attribute
  end if;
  case a.memberMod of
    { none } do nothing;
    { static } do
      if env[0] == CLASS then
        throw an AttributeError exception — non-class-member definitions cannot have a static attribute
      end if;
    end case;
  name: STRING :: Name[Identifier];
  ns: NAMESPACE :: new NAMESPACE[ name: name[] ]
  v: VARIABLE :: new VARIABLE[type: Namespace, value: ns, immutable: true, setup: none, initialiser: none[]]
  defineLocalMember(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)
end proc;

16 Programs

Syntax

Program :: Directives

Evaluation

EvalProgram[ Program :: Directives]: OBJECT
begin
  ext: CONTEXT :: new CONTEXT[strict: false, openNamespaces: {public[]}]
  Validate[Directives](ext, initialEnvironment, JUMPTARGETS[breakTargets: {}, continueTargets: {}] true, none);
  Setup[Directives]();
  return Eval[Directives](initialEnvironment, undefined)
end;
17 Predefined Identifiers

18 Built-in Classes

```ecma
proc makeBuiltInClass(name: STRING, super: CLASSOPT, prototype: OBJECTOPT, typeofString: STRING, dynamic: BOOLEAN, allowNull: BOOLEAN, final: BOOLEAN, defaultValue: OBJECTOPT,
  bracketRead: OBJECT [CLASS [OBJECT] [PHASE] OBJECTOPT, 
  bracketWrite: OBJECT [CLASS [OBJECT] [CLASS] [OBJECT] [PHASE] OBJECTOPT, 
  bracketDelete: OBJECT [CLASS [OBJECT] [PHASE] OBJECTOPT, 
  read: OBJECT [CLASS [MULTINAME [ENVIRONMENTOPT [PHASE] OBJECTOPT, 
  write: OBJECT [CLASS [MULTINAME [ENVIRONMENTOPT [OBJECTOPT [PHASE] OBJECTOPT, 
  delete: OBJECT [CLASS [MULTINAME [ENVIRONMENTOPT [PHASE] OBJECTOPT, 
  enumerate: OBJECT [PHASE] OBJECTOPT, 
  class]
  return c
end proc;

proc call(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  ???
  end proc;

proc construct(args: OBJECT[], phase: PHASE): OBJECT
  ???
  end proc;

privateNamespace: NAMESPACE [] new NAMESPACE[comment: "private"]
c: CLASS [] new CLASS[localBindings: {}, super: super, instanceMembers: {}, complete: true, name: name, 
  prototype: prototype, typeofString: typeofString, privateNamespace: privateNamespace, dynamic: dynamic, 
  final: final, defaultValue: defaultValue, bracketRead: bracketRead, bracketWrite: bracketWrite, 
  bracketDelete: bracketDelete, read: read, write: write, delete: delete, enumerate: enumerate, call: call, 
  construct: construct, init: none[]]
proc is(o: OBJECT): BOOLEAN
  return isAncestor(c, objectType(o))
end proc;

c.is[] is;
proc implicitCoerce(o: OBJECT, silent: BOOLEAN): OBJECT
  if c.is(o) or (o = null and allowNull) then return o 
  elseif silent and allowNull then return null 
  else throw a TypeError exception 
end if
end proc;

c.implicitCoerce[] implicitCoerce;
return c
end proc;

proc makeSimpleBuiltInClass(name: STRING, super: CLASS, typeofString: STRING, dynamic: BOOLEAN, 
  allowNull: BOOLEAN, final: BOOLEAN, defaultValue: OBJECTOPT): CLASS
return makeBuiltInClass(name, super, super.prototype, typeofString, dynamic, allowNull, final, defaultValue, 
  super.bracketRead, super.bracketWrite, super.bracketDelete, super.read, super.write, super.delete, 
  super.enumerate)
end proc;
```
proc makeBuiltInIntegerClass(name: String, low: Integer, high: Integer): Class
  proc call(this: Object, args: Object[], phase: Phase): Object
    ????
  end proc;
  proc construct(args: Object[], phase: Phase): Object
    ????
  end proc;
  proc is(o: Object): Boolean
    if o [] Float64 then
      case o of
        {NaN64, +∞f64, –∞f64} do return false;
        [+zero64, –zero64] do return true;
        NONZEROFINITEFLOAT64 do
          r: RATIONAL [] o.value;
          return r [] INTEGER and low ≤ r ≤ high
        end case
      else return false
    end if
  end proc;
  proc implicitCoerce(o: Object, silent: Boolean): Object
    if o = undefined then return +zero64
    elsif o [] GENERALNUMBER then
      i: INTEGEROPT [] checkInteger(o);
      if i ≠ none and low ≤ i ≤ high then
        note –zero32, +zero32, and –zero64 are all coerced to +zero64.
        return realToF64(i)
      end if
    end if;
    throw a TypeError exception
  end proc;
privateNamespace: Namespace [] new Namespace[]name: “private”[]
return new Class[]localBindings: {}, super: Number, instanceMembers: {}, complete: true, name: name,
  prototype: Number.prototype, typeofString: “number”, privateNamespace: privateNamespace,
  dynamic: false, final: true, defaultValue: +zero64, bracketRead: Number.bracketRead,
  bracketWrite: Number.bracketWrite, bracketDelete: Number.bracketDelete, read: Number.read,
  write: Number.write, delete: Number.delete, enumerate: Number.enumerate, call: call, construct: construct,
  init: none, is: is, implicitCoerce: implicitCoerce[]
end proc;

Object: Class = makeBuiltInClass(“Object”, none, none, “object”, false, true, false, undefined,
  defaultBracketRead, defaultBracketWrite, defaultBracketDelete, defaultReadProperty, defaultWriteProperty,
  defaultDeleteProperty, defaultEnumerate);

Never: Class = makeSimpleBuiltInClass(“Never”, Object, “”, false, false, true, none);

Void: Class = makeSimpleBuiltInClass(“Void”, Object, “undefined”, false, false, true, undefined);

Null: Class = makeSimpleBuiltInClass(“Null”, Object, “object”, false, true, true, null);

Boolean: Class = makeSimpleBuiltInClass(“Boolean”, Object, “boolean”, false, false, true, false);

GeneralNumber: Class
  = makeSimpleBuiltInClass(“GeneralNumber”, Object, “object”, false, false, false, NaN64);

long: Class = makeSimpleBuiltInClass(“long”, GeneralNumber, “long”, false, false, true, 0_long);

ulong: Class = makeSimpleBuiltInClass(“ulong”, GeneralNumber, “ulong”, false, false, true, 0_ulong);
float: `CLASS = makeSimpleBuiltInClass("float", GeneralNumber, "float", false, false, true, NaN);`

`Number: CLASS = makeSimpleBuiltInClass("Number", GeneralNumber, "number", false, false, true, NaN);`

`sbyte: CLASS = makeBuiltInIntegerClass("sbyte", -128, 127);`

`byte: CLASS = makeBuiltInIntegerClass("byte", 0, 255);`

`short: CLASS = makeBuiltInIntegerClass("short", -32768, 32767);`

`ushort: CLASS = makeBuiltInIntegerClass("ushort", 0, 65535);`

`int: CLASS = makeBuiltInIntegerClass("int", -2147483648, 2147483647);`

`uint: CLASS = makeBuiltInIntegerClass("uint", 0, 4294967295);`

`Character: CLASS = makeSimpleBuiltInClass("Character", Object, "character", false, false, true, ‘\0’);`

`String: CLASS = makeSimpleBuiltInClass("String", Object, "string", false, true, true, null);`

`Array: CLASS = makeBuiltInClass("Array", Object, arrayPrototype, "object", true, true, true, null, defaultBracketsRead, defaultBracketsWrite, defaultBracketsDelete, defaultReadProperty, arrayWriteProperty, defaultDeleteProperty, defaultEnumerate);`

`Namespace: CLASS = makeSimpleBuiltInClass("Namespace", Object, "namespace", false, true, true, null);`

`Attribute: CLASS = makeSimpleBuiltInClass("Attribute", Object, "object", false, true, true, null);`

`Date: CLASS = makeSimpleBuiltInClass("Date", Object, "object", true, true, true, null);`

`RegExp: CLASS = makeSimpleBuiltInClass("RegExp", Object, "object", true, true, true, null);`

`Class: CLASS = makeSimpleBuiltInClass("Class", Object, "function", false, true, true, null);`

`PrototypeFunction: CLASS = makeSimpleBuiltInClass("Function", Object, "function", false, true, true, null);`

`Prototype: CLASS = makeSimpleBuiltInClass("Object", Object, "object", true, true, true, null);`

`Package: CLASS = makeSimpleBuiltInClass("Package", Object, "object", true, true, true, null);`

`Error: CLASS = makeSimpleBuiltInClass("Error", Object, "object", true, true, false, null);`

`ArgumentError: CLASS = makeSimpleBuiltInClass("ArgumentError", Error, "object", true, true, false, null);`

`AttributeError: CLASS = makeSimpleBuiltInClass("AttributeError", Error, "object", true, true, false, null);`

`ConstantError: CLASS = makeSimpleBuiltInClass("ConstantError", Error, "object", true, true, false, null);`

`DefinitionError: CLASS = makeSimpleBuiltInClass("DefinitionError", Error, "object", true, true, false, null);`

`EvalError: CLASS = makeSimpleBuiltInClass("EvalError", Error, "object", true, true, false, null);`

`RangeError: CLASS = makeSimpleBuiltInClass("RangeError", Error, "object", true, true, false, null);`

`ReferenceError: CLASS = makeSimpleBuiltInClass("ReferenceError", Error, "object", true, true, false, null);`

`SyntaxError: CLASS = makeSimpleBuiltInClass("SyntaxError", Error, "object", true, true, false, null);`

`TypeError: CLASS = makeSimpleBuiltInClass("TypeError", Error, "object", true, true, false, null);`
UninitializedError: CLASS
  = makeSimpleBuiltInClass("UninitializedError", Error, "object", true, true, false, null);

URIError: CLASS = makeSimpleBuiltInClass("URIError", Error, "object", true, true, false, null);

objectPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE()

arrayPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE()

arrayLimit: INTEGER = 2^64 – 1;

arrayPrivate: NAMESPACE = new NAMESPACE()
  name: "private"

proc arrayWriteProperty(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT,
  createIfMissing: BOOLEAN, newValue: OBJECT, phase: {run}): {none, ok}
result: {none, ok} []
  defaultWriteProperty(o, limit, multiname, env, createIfMissing, newValue, phase);
if result = ok and |multiname| = 1 then
  qname: QUALIFIEDNAME [] the one element of multiname;
if qname.namespace = public then
  name: STRING [] qname.id;
i: INTEGER [] truncateToInteger(toGeneralNumber(name, phase));
if name = integerToString(i) and 0 ≤ i < arrayLimit then
  length: ULONG [] readInstanceProperty(o, arrayPrivate::"length", phase);
if i ≥ length.value then
  length [] (i + 1)ulong;
dotWrite(o, {arrayPrivate::"length"}, length, phase)
end if
end if
end if;
return result
end proc;

proc constructError(e: CLASS): OBJECT
  return e.construct([], run)
end proc;
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

```javascript
public: NAMESPACE = new NAMESPACE(name: "public")

internal: NAMESPACE = new NAMESPACE(name: "internal")

globalObject: PACKAGE = new PACKAGE(localBindings: {}, super: objectPrototype, sealed: false, internalNamespace: internal)

initialEnvironment: ENVIRONMENT = [globalObject, new SYSTEMFRAME(localBindings: {})]
```

22 Errors

23 Optional Packages

23.1 Machine Types

23.2 Internationalisation

A Index

A.1 Nonterminals

```
AdditiveExpression 100
AnnotatableDirective 139
Arguments 93
ArrayLiteral 85
ASCIIDigit 33
AssignmentExpression 112
Attribute 142
AttributeCombination 142
AttributeExpression 88
Attributes 142
BitwiseAndExpression 107
BitwiseOrExpression 108
BitwiseXorExpression 107
Block 121
BlockCommentCharacters 29
Brackets 93
BreakStatement 134
CaseElement 124
CaseElements 124
CaseElementsPrefix 124
CaseLabel 124
CatchClause 136
CatchClauses 136
CatchClausesOpt 136
ClassDefinition 167
CompoundAssignment 112
ConditionalExpression 111
ContinueStatement 133
ContinuingIdentifierCharacter 30
ContinuingIdentifierCharacterOrEscape 30
ControlEscape 35
DecimalDigits 33
DecimalIntegerLiteral 33
DecimalLiteral 32
Directive 139
Directives 139
DirectivesPrefix 139
DivisionPunctuator 32
DoStatement 127
ElementList 86
EmptyStatement 119
EndOfInput 27
EqualityExpression 105
ExpressionQualifiedIdentifier 78
ExpressionStatement 120
ExpressionsWithRest 93
FieldList 84
FieldName 84
ForInBinding 129
ForInitialiser 129
ForStatement 128
Fraction 33
FullNewExpression 88
FullNewSubexpression 89
FullPostfixExpression 88
FunctionCommon 152
FunctionDefinition 152
FunctionExpression 83
FunctionName 152
HexDigit 33
HexEscape 35
HexIntegerLiteral 33
Identifier 78
IdentifierName 30
IdentifierOrKeyword 29
IdentityEscape 35
IfStatement 123
Inheritance 167
InitialIdentifierCharacter 30
InitialIdentifierCharacterOrEscape 30
InputElement 27
IntegerLiteral 32
LabeledStatement 122
LetterE 32
```
LetterF 32
LetterL 32
LetterU 32
LetterX 33
LineBreak 28
LineBreaks 28
LineComment 29
LineCommentCharacters 29
LineTerminator 28
ListExpression 115
LiteralElement 86
LiteralField 84
LiteralStringChar 35
LogicalAndExpression 110
LogicalAssignment 112
LogicalOrExpression 110
LogicalXorExpression 110
Mantissa 32
MemberOperator 93
MultiLineBlockComment 29
MultiLineBlockCommentCharacters 29
MultiplicativeExpression 98
NamespaceDefinition 170
NextInputElement 27
NonAssignmentExpression 111
NonemptyFieldList 84
NonemptyParameters 163
NonexpressionAttribute 142
NonTerminator 29
NonTerminatorOrAsteriskOrSlash 29
NonTerminatorOrSlash 29
NonZeroDecimalDigits 33
NonZeroDigit 33
NullEscape 30
NullEscapes 30
NumericLiteral 32
ObjectLiteral 84
OptionalExpression 129
OrdinaryRegExpChar 36
Parameter 163
ParameterAttributes 163
ParameterInit 163
Parameters 163
ParenExpression 80
ParenListExpression 80
PostfixExpression 88
Pragma 144
PragmaArgument 145
PragmaExpr 144
PragmaItem 144
PragmaItems 144
PreSlashCharacters 29
PrimaryExpression 80
Program 170
Punctuator 32
QualifiedIdentifier 78
Qualifier 78
RegExpBody 36
RegExpChar 36
RegExpChars 36
RegExpFlags 36
RegExpLiteral 36
RelationalExpression 103
RestExpression 163
Result 163
ReturnStatement 135
Semicolon 116
ShiftExpression 101
ShortNewExpression 89
ShortNewSubexpression 89
SignedInteger 33
SimpleQualifiedIdentifier 78
SimpleVariableDefinition 151
SingleLineBlockComment 29
Statement 116
StringChar 35
StringChars 35
StringEscape 35
StringLiteral 35
Substatement 116
Substatements 116
SubstatementsPrefix 116
SuperExpression 87
SuperStatement 120
SwitchStatement 124
ThrowStatement 135
TryStatement 136
TypedIdentifier 146
TypeExpression 115
UnaryExpression 96
UnicodeAlphanumeric 30
UnicodeCharacter 29
UnicodeInitialAlphabetic 30
UntypedVariableBinding 151
UntypedVariableBindingList 151
UseDirective 144
VariableBinding 146
VariableBindingList 146
VariableDefinition 146
VariableDefinitionKind 146
VariableInitialisation 146
VariableInitialiser 146
WhileStatement 128
WhiteSpace 28
WhiteSpaceCharacter 28
WithStatement 133
ZeroEscape 35

A.2 Tags

→ 11, 12
+ 11, 12
+zero 11, 12
andEq 112
busy 48
compile 44
constructorFunction 45
default 44
decimal 52
false 4, 38
final 39
forbidden 48
get 46
greater 52
instanceFunction 45
less 52
NaN 11, 12
none 38, 39, 41, 46, 47, 48, 49
normal 46
null 38
orEq 112
plainFunction 45
prototypeFunction 45
read 47
readWrite 47
run 44
set 46
static 39
true 4, 38
uncheckedFunction 45
undefined 38
unordered 52
virtual 39
write 47
xorEq 112
–zero 11, 12

A.3 Semantic Domains

ACCESS 47
ACCESSSET 47
ATTRIBUTE 39
A.4 Globals

accessesOverlap 57
cannotReturnValue 135
callInit 169
class 39
classOpt 41
compoundAttribute 39
context 44
continue 45
controlTransfer 45
date 42
denormalisedFloat32Values 11
denormalisedFloat64Values 13
dotReference 44
dynamicVar 48
evironment 46
evironmentOpt 46
finiteFloat32 11
finiteFloat64 12
finiteGeneralNumber 10
float32 11
float64 12
frame 46
functionKind 45
generalNumber 10
getter 48
handling 46
initialiser 48
initialiserOpt 48
inputElement 26
instanceGetter 49
instanceMember 49
instanceMemberOpt 49
instanceMethod 49
instanceSetter 50
instanceVariable 49
instanceVariableOpt 49
integer 6
integerOpt 38
jumpTargets 44
label 44
lexicalReference 43
limitedInstance 43
localBinding 47
localFrame 47
localMember 48
localMemberOpt 48
long 10
memberModifier 39
memberTranslation 76
methodClosure 42
 multiname 39
namespace 38
nonWithFrame 46
nonzeroFiniteFloat32 11
nonzeroFiniteFloat64 12
normalisedFloat32Values 11
normalisedFloat64Values 12
null 38
object 38
objectOpt 38
objOptionalLimit 43
objOrRef 43
order 52
overrideModifier 39
package 43
parameter 47
parameterFrame 46
parameterFrameOpt 47
phase 44
qualifiedObject 38
rational 6
real 6
reference 43
regexp 42
return 45
semanticException 45
setter 49
simpleInstance 41
slot 41
staticFunctionKind 45
string 8, 38
stringOpt 38
switchGuard 124
switchKey 124
systemFrame 46
token 26
ulong 10
undefined 38
uninstantiatedFunction 41
variable 48
variableOpt 48
variableValue 48
withFrame 47

defineLocalMember 72
definitionError 173
deleteReference 69
divide 99
dotRead 62
dotWrite 66
enumerateCommonMembers 71
enumerateInstanceMembers 71
error 173
evaluateError 173
findBaseInstanceMember 60
findCommonMember 60
findLocalInstanceMember 60
findLocalMember 59
findSlot 58
float 173
float32Negate 12
float32ToFloat64 13
float32ToString 55
float64Abs 14
float64Add 14
float64Divide 15
float64Multiply 15
float64Negate 14  
float64Remainder 15  
float64Subtract 14  
float64ToString 56  
Function 173  
GeneralNumber 172  
generalNumberCompare 52  
generalNumberNegate 98  
getDerivedInstanceMember 61  
getEnclosingClass 58  
getEnclosingParameterFrame 59  
getPackageFrame 59  
getRegionalEnvironment 59  
globalObject 176  
indexRead 62  
indexWrite 66  
initialEnvironment 176  
instantiateMemberAccesses 60  
instantiateFunction 75  
instantiateLocalFrame 76  
instantiateMember 76  
instantiateParameterFrame 77  
int 173  
integerToLong 51  
integerToString 54  
integerToStringWithSign 54  
integerToULong 51  
internal 176  
isEqual 107  
isLess 105  
isLessOrEqual 105  
isStrictlyEqual 107  
lexicalDelete 70  
lexicalRead 62  
lexicalWrite 67  
logicalNot 98  
long 172  
lookUpInstanceMember 61  
makeBuiltInClass 171  
makeBuiltInIntegerClass 172  
makeLimitedInstance 88  
makeSimpleBuiltInClass 171  
minus 97  
multiply 99  
Namespace 173  
Never 172  
Null 172  
Number 173  
Object 172  
objectPrototype 174  
objectSupers 58  
objectType 53  
Package 173  
plus 97  
processPragma 145  
Prototype 173  
PrototypeFunction 173  
public 176  
RangeError 173  
rationalToLong 51  
rationalToULong 52  
readImplicitThis 61  
readInstanceMember 64  
readInstanceProperty 63  
readLocalMember 65  
readReference 61  
realToFloat32 11  
realToFloat64 13  
ReferenceError 173  
RegExp 173  
remainder 100  
sbyte 173  
searchForOverrides 73  
setupVariable 58  
shiftLeft 102  
shiftRight 103  
shiftRightUnsigned 103  
short 173  
signedWrap32 50  
signedWrap64 51  
String 173  
subtract 101  
SyntaxError 173  
toBoolean 53  
toClass 56  
toCompoundAttribute 57  
toFloat64 52  
toGeneralNumber 53  
toPrimitive 56  
toQualifiedName 56  
toRational 52  
toString 54  
truncateFiniteFloat32 12  
truncateFiniteFloat64 13  
truncateToInteger 51  
TypeError 173  
uint 173  
ulong 172  
UninitializedError 174  
unsignedWrap32 50  
unsignedWrap64 50  
URIError 174  
ushort 173  
Void 172  
writeInstanceMember 68  
writeLocalMember 69  
writeReference 66  
writeVariable 58