

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Terminals

General tokens: Identifier NegatedMinLong Number RegularExpression String VirtualSemicolon

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

Reserved words: as break case catch class const continue default delete do else extends false
finally for function if import in instanceof is namespace new null package private public
return super switch this throw true try typeof use var void while with

Future reserved words: abstract debugger enum export goto implements interface native protected
synchronized throws transient volatile

Non-reserved words: get, set

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: import

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: package

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: exclude

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: include

Data Model

Semantic Exceptions

tuple BREAK
value: OBJECT,
label: LABEL
end tuple;

tuple CONTINUE
value: OBJECT,
label: LABEL
end tuple;

tuple RETURN
value: OBJECT
end tuple;

CONTROLTRANSFER = BREAK ∪ CONTINUE ∪ RETURN;

SEMANTICEXCEPTION = OBJECT ∪ CONTROLTRANSFER;

Extended integers and rationals

tag +zero;

tag -zero;

tag +∞;

tag -∞;

tag NaN;

EXTENDED RATIONAL = (RATIONAL - {0}) ∪ {+zero, -zero, +∞, -∞, NaN};

EXTENDED INTEGER = INTEGER ∪ {+∞, -∞, NaN};

tag syntaxError;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: String to Number Conversion

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: EXTENDED RATIONAL = RATIONAL
∪ {+zero, -zero, +∞, -∞, NaN};

Objects

tag none;

tag ok;

tag reject;

OBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHAR16 ∪ STRING ∪
NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ SIMPLEINSTANCE ∪ METHODCLOSURE ∪ DATE ∪ REGEXP ∪
PACKAGE;

PRIMITIVEOBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHAR16 ∪ STRING;

NONPRIMITIVEOBJECT = NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ SIMPLEINSTANCE ∪ METHODCLOSURE ∪
DATE ∪ REGEXP ∪ PACKAGE;

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

OBJECTOPT = OBJECT ∪ {none};

BOOLEANOPT = BOOLEAN ∪ {none};

INTEGEROPT = INTEGER ∪ {none};

Undefined

tag undefined;

UNDEFINED = {undefined};

Null

tag null;

NULL = {null};

Strings

STRINGOPT = STRING ∪ {none};

Namespaces

record NAMESPACE
 name: STRING
end record;

Qualified Names

tuple QUALIFIEDNAME
 namespace: NAMESPACE,
 id: STRING
end tuple;

The notation *ns::id* is a shorthand for QUALIFIEDNAME(namespace: *ns*, id: *id*).

MULTINAME = QUALIFIEDNAME{};

Attributes

tag static;

```

tag virtual;

tag final;

PROPERTYCATEGORY = {none, static, virtual, final};

OVERRIDEMODIFIER = {none, true, false, undefined};

tuple COMPOUNDATTRIBUTE
  namespaces: NAMESPACE {},
  explicit: BOOLEAN,
  enumerable: BOOLEAN,
  dynamic: BOOLEAN,
  category: PROPERTYCATEGORY,
  overrideMod: OVERRIDEMODIFIER,
  prototype: BOOLEAN,
  unused: BOOLEAN
end tuple;

ATTRIBUTE = BOOLEAN ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE;

ATTRIBUTEOPTNOTFALSE = {none, true} ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE;

```

Classes

```

record CLASS
  localBindings: LOCALBINDING {},
  instanceProperties: INSTANCEPROPERTY {},
  super: CLASSOPT,
  prototype: OBJECTOPT,
  complete: BOOLEAN,
  name: STRING,
  typeofString: STRING,
  privateNamespace: NAMESPACE,
  dynamic: BOOLEAN,
  final: BOOLEAN,
  defaultValue: OBJECTOPT,
  defaultHint: HINT,
  bracketRead: OBJECT × CLASS × OBJECT[] × PHASE → OBJECTOPT,
  bracketWrite: OBJECT × CLASS × OBJECT[] × OBJECT × {run} → {none, ok},
  bracketDelete: OBJECT × CLASS × OBJECT[] × {run} → BOOLEANOPT,
  read: OBJECT × CLASS × MULTINAME × ENVIRONMENTOPT × PHASE → OBJECTOPT,
  write: OBJECT × CLASS × MULTINAME × ENVIRONMENTOPT × BOOLEAN × OBJECT × {run} → {none, ok},
  delete: OBJECT × CLASS × MULTINAME × ENVIRONMENTOPT × {run} → BOOLEANOPT,
  enumerate: OBJECT → OBJECT {},
  call: OBJECT × OBJECT[] × PHASE → OBJECT,
  construct: OBJECT[] × PHASE → OBJECT,
  init: (SIMPLEINSTANCE × OBJECT[] × {run} → ()) ∪ {none},
  is: OBJECT × CLASS → BOOLEAN,
  as: OBJECT × CLASS × BOOLEAN → OBJECT
end record;

CLASSOPT = CLASS ∪ {none};

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: CLASSOPT, ,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: OBJECT × CLASS ×
BOOLEAN → OBJECT

Simple Instances

```
record SIMPLEINSTANCE
  localBindings: LOCALBINDING {},
  archetype: OBJECTOPT,
  sealed: BOOLEAN,
  type: CLASS,
  slots: SLOT {},
  call: (OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none},
  construct: (SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none},
  env: ENVIRONMENTOPT
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Slots

```
record SLOT
  id: INSTANCEVARIABLE,
  value: OBJECTOPT
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: OBJECTOPT,

Uninstantiated Functions

```
record UNINSTANTIATEDFUNCTION
  type: CLASS,
  length: INTEGER,
  call: (OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none},
  construct: (SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none},
  instantiations: SIMPLEINSTANCE {}
end record;
```

Method Closures

```
tuple METHODCLOSURE
  this: OBJECT,
  method: INSTANCEMETHOD,
  slots: SLOT {}
end tuple;
```

Dates

```
record DATE
  localBindings: LOCALBINDING {},
  archetype: OBJECTOPT,
  sealed: BOOLEAN,
  timeValue: INTEGER
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: OBJECTOPT,

Regular Expressions

```
record REGEXP
  localBindings: LOCALBINDING {},
  archetype: OBJECTOPT,
  sealed: BOOLEAN,
  source: STRING,
  lastIndex: INTEGER,
  global: BOOLEAN,
  ignoreCase: BOOLEAN,
  multiline: BOOLEAN
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: OBJECTOPT,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Packages

```
record PACKAGE
  localBindings: LOCALBINDING {},
  archetype: OBJECTOPT,
  name: STRING,
  initialize: (() → ()) ∪ {none, busy},
  sealed: BOOLEAN,
  internalNamespace: NAMESPACE
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: OBJECTOPT,

Objects with Limits

instance must be an instance of one of limit's descendants.

```
tuple LIMITEDINSTANCE
  instance: OBJECT,
  limit: CLASS
end tuple;
```

$OBJOPTIONALLIMIT = OBJECT \cup LIMITEDINSTANCE;$

References

```
tuple LEXICALREFERENCE
  env: ENVIRONMENT,
  variableMultiname: MULTINAME,
  strict: BOOLEAN
end tuple;
```

```
tuple DOTREFERENCE
  base: OBJECT,
  limit: CLASS,
  multiname: MULTINAME
end tuple;
```

```
tuple BRACKETREFERENCE
  base: OBJECT,
  limit: CLASS,
  args: OBJECT[]
end tuple;
```

$REFERENCE = LEXICALREFERENCE \cup DOTREFERENCE \cup BRACKETREFERENCE;$

$OBJORREF = OBJECT \cup REFERENCE;$

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: propertyMultiname: MULTINAME

Modes of expression evaluation

tag compile;

tag run;

$PHASE = \{compile, run\};$

Contexts

```
record CONTEXT
  strict: BOOLEAN,
  openNamespaces: NAMESPACE{}
end record;
```

Labels

```
tag default;

LABEL = STRING ∪ {default};

tuple JUMPTARGETS
  breakTargets: LABEL {},
  continueTargets: LABEL {}
end tuple;
```

Function Support

```
tag normal;

tag get;

tag set;

HANDLING = {normal, get, set};

tag plainFunction;

tag uncheckedFunction;

tag prototypeFunction;

tag instanceFunction;

tag constructorFunction;

STATICFUNCTIONKIND = {plainFunction, uncheckedFunction, prototypeFunction};

FUNCTIONKIND = {plainFunction, uncheckedFunction, prototypeFunction, instanceFunction,
  constructorFunction};
```

Environments

An **ENVIRONMENT** is a list of two or more frames. Each frame corresponds to a scope. More specific frames are listed first—each frame’s scope is directly contained in the following frame’s scope. The last frame is always a **PACKAGE**. A **WITHFRAME** is always preceded by a **LOCALFRAME**, so the first frame is never a **WITHFRAME**.

```
ENVIRONMENT = FRAME[];

ENVIRONMENTOPT = ENVIRONMENT ∪ {none};

FRAME = NonWithFRAME ∪ WithFRAME;

NonWithFRAME = PACKAGE ∪ PARAMETERFRAME ∪ CLASS ∪ LOCALFRAME;
```

```
record PARAMETERFRAME
  localBindings: LOCALBINDING {},
  kind: FUNCTIONKIND,
  handling: HANDLING,
  callsSuperconstructor: BOOLEAN,
  superconstructorCalled: BOOLEAN,
  this: OBJECTOPT,
  parameters: PARAMETER[],
  rest: VARIABLEOPT,
  returnType: CLASS
end record;

PARAMETERFRAMEOPT = PARAMETERFRAME ∪ {none};

tuple PARAMETER
  var: VARIABLE ∪ DYNAMICVAR,
  default: OBJECTOPT
end tuple;

record LOCALFRAME
  localBindings: LOCALBINDING {}
end record;

record WITHFRAME
  value: OBJECTOPT
end record;
```

Properties

```
tag read;

tag write;

tag readWrite;

ACCESS = {read, write};

ACCESSSET = {read, write, readWrite};

tuple LOCALBINDING
  qname: QUALIFIEDNAME,
  accesses: ACCESSSET,
  explicit: BOOLEAN,
  enumerable: BOOLEAN,
  content: SINGLETONPROPERTY
end tuple;

tag forbidden;

SINGLETONPROPERTY = {forbidden} ∪ VARIABLE ∪ DYNAMICVAR ∪ GETTER ∪ SETTER;

SINGLETONPROPERTYOPT = SINGLETONPROPERTY ∪ {none};

VARIABLEVALUE = {none} ∪ OBJECT ∪ UNINSTANTIATEDFUNCTION;

tag busy;
```

INITIALIZER = ENVIRONMENT × PHASE → OBJECT;

INITIALIZEROPT = INITIALIZER ∪ {none};

```
record VARIABLE
  type: CLASS,
  value: VARIABLEVALUE,
  immutable: BOOLEAN,
  setup: (() → CLASSOPT) U {none, busy},
  initializer: INITIALIZER U {none, busy},
  initializerEnv: ENVIRONMENT
end record;

VARIABLEOPT = VARIABLE U {none};

record DYNAMICVAR
  value: OBJECT U UNINSTANTIATEDFUNCTION,
  sealed: BOOLEAN
end record;

record GETTER
  call: ENVIRONMENT × PHASE → OBJECT,
  env: ENVIRONMENTOPT
end record;

record SETTER
  call: OBJECT × ENVIRONMENT × PHASE → (),
  env: ENVIRONMENTOPT
end record;

INSTANCEPROPERTY = INSTANCEVARIABLE U INSTANCEMETHOD U INSTANCEGETTER U INSTANCESSETTER;

INSTANCEPROPERTYOPT = INSTANCEPROPERTY U {none};

record INSTANCEVARIABLE
  multiname: MULTINAME,
  final: BOOLEAN,
  enumerable: BOOLEAN,
  type: CLASS,
  defaultValue: OBJECTOPT,
  immutable: BOOLEAN
end record;

INSTANCEVARIABLEOPT = INSTANCEVARIABLE U {none};

record INSTANCEMETHOD
  multiname: MULTINAME,
  final: BOOLEAN,
  enumerable: BOOLEAN,
  signature: PARAMETERFRAME,
  length: INTEGER,
  call: OBJECT × OBJECT[] × PHASE → OBJECT
end record;

record INSTANCEGETTER
  multiname: MULTINAME,
  final: BOOLEAN,
  enumerable: BOOLEAN,
  signature: PARAMETERFRAME,
  call: OBJECT × PHASE → OBJECT
end record;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: INITIALISER U {none, bus... [2]


```

record INSTANCESETTER
  multiname: MULTINAME,
  final: BOOLEAN,
  enumerable: BOOLEAN,
  signature: PARAMETERFRAME,
  call: OBJECT × OBJECT × PHASE → ()
end record;

PROPERTYOPT = SINGLETONPROPERTY ∪ INSTANCEPROPERTY ∪ {none};

```

Miscellaneous

```

tag hintString;
tag hintNumber;

HINT = {hintString, hintNumber};

HINTOPT = HINT ∪ {none};

tag less;
tag equal;
tag greater;
tag unordered;

ORDER = {less, equal, greater, unordered};

```

Data Operations

Numeric Utilities

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

```

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

```

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

```

proc signedWrap32(i: INTEGER): { $-2^{31}$  ...  $2^{31}-1$ }
  j: INTEGER ← bitwiseAnd(i, 0xFFFFFFFF);
  if  $j \geq 2^{31}$  then j ← j -  $2^{32}$  end if;
  return j
end proc;

```

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

```

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

```

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

```

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

```

truncateToExtendedInteger(*x*) returns *x* converted to an integer by rounding towards zero. If *x* is an infinity or a NaN, the result is **+∞**, **-∞**, or **NaN**, as appropriate.

```

proc truncateToExtendedInteger(x: GENERALNUMBER): EXTENDEDINTEGER
  case x of
    {+∞f32, +∞f64} do return +∞;
    {-∞f32, -∞f64} do return -∞;
    {NaNf32, NaNf64} do return NaN;
    FINITEFLOAT32 do return truncateFiniteFloat32(x);
    FINITEFLOAT64 do return truncateFiniteFloat64(x);
    LONG ∪ ULONG do return x.value
  end case
end proc;

```

truncateToInteger(*x*) returns *x* converted to an integer by rounding towards zero. If *x* is an infinity or a NaN, the result is 0.

```

proc truncateToInteger(x: GENERALNUMBER): INTEGER
  i: EXTENDEDINTEGER ← truncateToExtendedInteger(x);
  case i of
    {+∞, -∞, NaN} do return 0;
    INTEGER do return i
  end case
end proc;

```

checkInteger(*x*) returns *x* converted to an integer if its mathematical value is, in fact, an integer. If *x* is an infinity or a NaN or has a fractional part, the result is **none**.

```

proc checkInteger(x: GENERALNUMBER): INTEGEROPT
  case x of
    {NaNf32, NaNf64, +∞f32, +∞f64, -∞f32, -∞f64} do return none;
    {+zerof32, +zerof64, -zerof32, -zerof64} do return 0;
    LONG ∪ ULONG do return x.value;
    NONZEROFINITEFLOAT32 ∪ NONZEROFINITEFLOAT64 do
      r: RATIONAL ← x.value;
      if r ∉ INTEGER then return none end if;
      return r
    end case
  end case
end proc;

```

integerToLong(*i*) converts *i* to the first of the types **LONG**, **ULONG**, or **Float64** that can contain the value *i*. If necessary, the **Float64** result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

```

proc integerToLong(i: INTEGER): GENERALNUMBER
  if  $-2^{63} \leq i \leq 2^{63} - 1$  then return ilong
  elsif  $2^{63} \leq i \leq 2^{64} - 1$  then return iulong
  else return realToFloat64(i)
  end if
end proc;

```

integerToULong(*i*) converts *i* to the first of the types **ULONG**, **LONG**, or **Float64** that can contain the value *i*. If necessary, the **Float64** result may be rounded or converted to an infinity using the IEEE 754 “round to nearest” mode.

```

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

```

rationalToLong(*q*) converts *q* to one of the types **LONG**, **ULONG**, or **Float64**, whichever one can come the closest to representing the true value of *q*. If several of these types can come equally close to the value of *q*, then one of them is chosen according to the algorithm below.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 0.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *truncateToInteger*(*x*: GENERALNUMBER): INTEGER

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: {+∞_{f32}, +∞_{f64}, -∞_{f32}, -∞_{f64}, NaN_{f32}, NaN_{f64}} do return 0;

```

proc rationalToLong(q: RATIONAL): GENERALNUMBER
  if q ∈ INTEGER then return integerToLong(q)
  elseif  $|q| \leq 2^{53}$  then return realToFloat64(q)
  elseif  $q < -2^{63} - 1/2$  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  $i < 2^{63}$  then return ilong else return iulong end if
  end if
end proc;

```

rationalToLong(*q*) converts *q* to one of the types **ULONG**, **LONG**, or **Float64**, whichever one can come the closest to representing the true value of *q*. If several of these types can come equally close to the value of *q*, then one of them is chosen according to the algorithm below.

```

proc rationalToLong(q: RATIONAL): GENERALNUMBER
  if q ∈ INTEGER then return integerToLong(q)
  elseif  $|q| \leq 2^{53}$  then return realToFloat64(q)
  elseif  $q < -2^{63} - 1/2$  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;

```

toRational(*x*) returns the exact **RATIONAL** value of *x*.

```

proc toRational(x: FINITEGENERALNUMBER): RATIONAL
  case x of
    {+zerof32, +zerof64, -zerof32, -zerof64} do return 0;
    NONZEROFINITEFLOAT32 ∪ NONZEROFINITEFLOAT64 ∪ LONG ∪ ULONG do return x.value
  end case
end proc;

```

toFloat32(*x*) converts *x* to a **Float32**, using the IEEE 754 “round to nearest” mode.

```

proc toFloat32(x: GENERALNUMBER): FLOAT32
  case x of
    LONG ∪ ULONG do return realToFloat32(x.value);
    FLOAT32 do return x;
    {-∞f64} do return -∞f32;
    {-zerof64} do return -zerof32;
    {+zerof64} do return +zerof32;
    {+∞f64} do return +∞f32;
    {NaNf64} do return NaNf32;
    NONZEROFINITEFLOAT64 do return realToFloat32(x.value)
  end case
end proc;

```

toFloat64(*x*) converts *x* to a **Float64**, using the IEEE 754 “round to nearest” mode.

```

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;

```

generalNumberCompare(*x*, *y*) compares *x* with *y* using the IEEE 754 rules and returns **less** if $x < y$, **equal** if $x = y$, **greater** if $x > y$, or **unordered** if either *x* or *y* is a NaN. The comparison is done using the exact values of *x* and *y*, even if they have different types. Positive infinities compare equal to each other and greater than any other non-NaN values. Negative infinities

compare equal to each other and less than any other non-NaN values. Positive and negative zeroes compare equal to each other.

```

proc generalNumberCompare(x: GENERALNUMBER, y: GENERALNUMBER): ORDER
  if x ∈ {NaNf32, NaNf64} or y ∈ {NaNf32, NaNf64} then return unordered
  elseif x ∈ {+∞f32, +∞f64} and y ∈ {+∞f32, +∞f64} then return equal
  elseif x ∈ {-∞f32, -∞f64} and y ∈ {-∞f32, -∞f64} then return equal
  elseif x ∈ {+∞f32, +∞f64} or y ∈ {-∞f32, -∞f64} then return greater
  elseif x ∈ {-∞f32, -∞f64} or y ∈ {+∞f32, +∞f64} then return less
  else
    xr: RATIONAL ← toRational(x);
    yr: RATIONAL ← toRational(y);
    if xr < yr then return less
    elseif xr > yr then return greater
    else return equal
  end if
end if
end proc;

```

Character Utilities

```

proc integerToUTF16(i: {0 ... 0x10FFFF}): STRING
  if 0 ≤ i ≤ 0xFFFF then return [integerToChar16(i)]
  else
    j: {0 ... 0xFFFF} ← i - 0x10000;
    high: CHAR16 ← integerToChar16(0xD800 + bitwiseShift(j, -10));
    low: CHAR16 ← integerToChar16(0xDC00 + bitwiseAnd(j, 0x3FF));
    return [high, low]
  end if
end proc;

```

```

proc charToLowerFull(ch: CHAR16): STRING
  return ch converted to a lower case character using the Unicode full, locale-independent case mapping. A single character may be converted to multiple characters. If ch has no lower case equivalent, then the result is the string [ch].
end proc;

```

```

proc charToLowerLocalized(ch: CHAR16): STRING
  return ch converted to a lower case character using the Unicode full case mapping in the host environment's current locale. A single character may be converted to multiple characters. If ch has no lower case equivalent, then the result is the string [ch].
end proc;

```

```

proc charToUpperFull(ch: CHAR16): STRING
  return ch converted to an upper case character using the Unicode full, locale-independent case mapping. A single character may be converted to multiple characters. If ch has no upper case equivalent, then the result is the string [ch].
end proc;

```

```

proc charToUpperLocalized(ch: CHAR16): STRING
  return ch converted to an upper case character using the Unicode full case mapping in the host environment's current locale. A single character may be converted to multiple characters. If ch has no upper case equivalent, then the result is the string [ch].
end proc;

```

Object Utilities

Object Class Inquiries

objectType(o) returns an **OBJECT** *o*'s most specific type. Although *objectType* is used internally throughout this specification, in order to allow one programmer-visible class to be implemented as an ensemble of implementation-specific classes, no way is provided for a user program to directly obtain the result of calling *objectType* on an object.

```

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 float;
    FLOAT64 do return Number;
    CHAR16 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;

```

is(o, c) returns **true** if *o* is an instance of class *c* or one of its subclasses.

```

proc is(o: OBJECT, c: CLASS): BOOLEAN
  return c.is(o, c)
end proc;

```

ordinaryIs(o, c) is the implementation of *is* for a native class unless specified otherwise in the class's definition. Host classes may either also use *ordinaryIs* or define a different procedure to perform this test.

```

proc ordinaryIs(o: OBJECT, c: CLASS): BOOLEAN
  return isAncestor(c, objectType(o))
end proc;

```

Return an ordered list of class *c*'s ancestors, including *c* itself.

```

proc ancestors(c: CLASS): CLASS[]
  s: CLASSOPT ← c.super;
  if s = none then return [c] else return ancestors(s) ⊕ [c] end if
end proc;

```

Return **true** if *c* is *d* or an ancestor of *d*.

```

proc isAncestor(c: CLASS, d: CLASS): BOOLEAN
  if c = d then return true
  else
    s: CLASSOPT ← d.super;
    if s = none then return false end if;
    return isAncestor(c, s)
  end if
end proc;

```

Object to Boolean Conversion

objectToBoolean(o) returns *o* converted to a *Boolean*.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *objectType*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *is* -

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *toBoolean* -

... [4]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

```

proc objectToBoolean(o: OBJECT): BOOLEAN
case o of
  UNDEFINED ∪ NULL do return false;
  BOOLEAN do return o;
  LONG ∪ ULONG do return o.value ≠ 0;
  FLOAT32 do return o ∉ {+zerof32, -zerof32, NaNf32};
  FLOAT64 do return o ∉ {+zerof64, -zerof64, NaNf64};
  STRING do return o ≠ "";
  CHAR16 ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ SIMPLEINSTANCE ∪ METHODCLOSURE ∪ DATE ∪
    REGEXP ∪ PACKAGE do
    return true
end case
end proc;

```

Object to Primitive Conversion

```

proc objectToPrimitive(o: OBJECT, hint: HINTOPT, phase: PHASE): PRIMITIVEOBJECT
if o ∈ PRIMITIVEOBJECT then return o end if;
c: CLASS ← objectType(o);
h: HINT;
if hint ∈ HINT then h ← hint else h ← c.defaultHint end if;
case h of
  {hintString} do
    toStringMethod: OBJECTOPT ← c.read(o, c, {public::"toString"}, none, phase);
    if toStringMethod ≠ none then
      r: OBJECT ← call(o, toStringMethod, [], phase);
      if r ∈ PRIMITIVEOBJECT then return r end if
    end if;
    valueOfMethod: OBJECTOPT ← c.read(o, c, {public::"valueOf"}, none, phase);
    if valueOfMethod ≠ none then
      r: OBJECT ← call(o, valueOfMethod, [], phase);
      if r ∈ PRIMITIVEOBJECT then return r end if
    end if;
  {hintNumber} do
    valueOfMethod: OBJECTOPT ← c.read(o, c, {public::"valueOf"}, none, phase);
    if valueOfMethod ≠ none then
      r: OBJECT ← call(o, valueOfMethod, [], phase);
      if r ∈ PRIMITIVEOBJECT then return r end if
    end if;
    toStringMethod: OBJECTOPT ← c.read(o, c, {public::"toString"}, none, phase);
    if toStringMethod ≠ none then
      r: OBJECT ← call(o, toStringMethod, [], phase);
      if r ∈ PRIMITIVEOBJECT then return r end if
    end if;
  end case;
  throw a TypeError exception — cannot convert this object to a primitive
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toGeneralNumber

Object to Number Conversions

objectToGeneralNumber(o, phase) returns o converted to a *GeneralNumber*. If *phase* is **compile**, only constant conversions are permitted.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toGeneralNumber(o, phase) returns o coerced

```

proc objectToGeneralNumber(o: OBJECT, phase: PHASE): GENERALNUMBER
  a: PRIMITIVEOBJECT;
  if o ∈ PRIMITIVEOBJECT then a ← o
  else a ← objectToPrimitive(o, hintNumber, phase)
  end if;
  case a of
    UNDEFINED do return NaN64;
    NULL ∪ {false} do return +zero64;
    {true} do return 1.064;
    GENERALNUMBER do return a;
    CHAR16 ∪ STRING do return stringToFloat64(toString(a))
  end case
end proc;

```

objectToFloat32(*o*, *phase*) returns *o* converted to a FLOAT32. If *phase* is **compile**, only constant conversions are permitted.

```

proc objectToFloat32(o: OBJECT, phase: PHASE): FLOAT32
  a: PRIMITIVEOBJECT;
  if o ∈ PRIMITIVEOBJECT then a ← o
  else a ← objectToPrimitive(o, hintNumber, phase)
  end if;
  case a of
    UNDEFINED do return NaN32;
    NULL ∪ {false} do return +zero32;
    {true} do return 1.032;
    GENERALNUMBER do return toFloat32(a);
    CHAR16 ∪ STRING do return stringToFloat32(toString(a))
  end case
end proc;

```

objectToFloat64(*o*, *phase*) returns *o* converted to a FLOAT64. If *phase* is **compile**, only constant conversions are permitted.

```

proc objectToFloat64(o: OBJECT, phase: PHASE): FLOAT64
  return toFloat64(objectToGeneralNumber(o, phase))
end proc;

```

objectToImpreciseInteger(*o*, *phase*) returns *o* converted to an EXTENDEDINTEGER. If *o* has a fractional part, it is truncated towards zero. If *o* is a string, then it is converted to a FLOAT64 first, which may cause loss of precision. If *phase* is **compile**, only constant conversions are permitted.

```

proc objectToImpreciseInteger(o: OBJECT, phase: PHASE): EXTENDEDINTEGER
  return truncateToExtendedInteger(objectToGeneralNumber(o, phase))
end proc;

```

objectToPreciseInteger(*o*, *phase*) returns *o* converted to an INTEGER. An error occurs if *o* has a fractional part or is not finite. If *o* is a string, then it is converted exactly. If *phase* is **compile**, only constant conversions are permitted.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toGeneralNumber(*o*: OBJECT, *phase*: PHASE): GENERALNUMBER [5]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: GENERALNUMBER do return ... [6]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toString(*o*, *phase*) returns *o* coerced to a String.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toString(*o*: OBJECT, *phase*: PHASE): STRING [7]

```

proc objectToPreciseInteger(o: OBJECT, phase: PHASE): INTEGER
  a: PRIMITIVEOBJECT;
  if o ∈ PRIMITIVEOBJECT then a ← o
  else a ← objectToPrimitive(o, hintNumber, phase)
  end if;
  case a of
    UNDEFINED ∪ NULL ∪ NULL ∪ {false} do return 0;
    {true} do return 1;
    GENERALNUMBER do
      i: INTEGEROPT ← checkInteger(a);
      if i = none then throw a RangeError exception — a is not finite
      else return i
      end if;
    CHAR16 ∪ STRING do
      i: INTEGEROPT ← stringToInteger(toString(a), 10);
      if i = none then
        throw a TypeError exception — the string a does not contain an integer literal
      else return i
      end if;
  end case
end proc;

proc stringToFloat32(s: STRING): FLOAT32
  q: EXTENDED RATIONAL ∪ {syntaxError} ← the result of parsing s using StringNumericLiteral as the start symbol;
  case q of
    {syntaxError} do return NaNf32;
    RATIONAL do return realToFloat32(q);
    {+zero} do return +zerof32;
    {-zero} do return -zerof32;
    {+∞} do return +∞f32;
    {-∞} do return -∞f32;
    {NaN} do return NaNf32
  end case
end proc;

proc stringToFloat64(s: STRING): FLOAT64
  q: EXTENDED RATIONAL ∪ {syntaxError} ← the result of parsing s using StringNumericLiteral as the start symbol;
  case q of
    {syntaxError} do return NaNf64;
    RATIONAL do return realToFloat64(q);
    {+zero} do return +zerof64;
    {-zero} do return -zerof64;
    {+∞} do return +∞f64;
    {-∞} do return -∞f64;
    {NaN} do return NaNf64
  end case
end proc;

proc stringToInteger(s: STRING, radix: INTEGER): INTEGEROPT
  ???
end proc;

```

Object to String Conversions

objectToString(o, phase) returns *o* converted to a *String*. If *phase* is **compile**, only constant conversions are permitted.


```

proc objectToString(o: OBJECT, phase: PHASE): STRING
  a: PRIMITIVEOBJECT;
  if o ∈ PRIMITIVEOBJECT then a ← o
  else a ← objectToPrimitive(o, hintString, phase)
  end if;
  case a of
    UNDEFINED do return "undefined";
    NULL do return "null";
    {false} do return "false";
    {true} do return "true";
    GENERALNUMBER do return generalNumberToString(a);
    CHAR16 do return [a];
    STRING do return a
  end case
end proc;

```

```

proc toString(o: CHAR16 ∪ STRING): STRING
  case o of
    CHAR16 do return [o];
    STRING do return o
  end case
end proc;

```

```

proc generalNumberToString(x: GENERALNUMBER): STRING
  case x of
    LONG ∪ ULONG do return integerToString(x.value);
    FLOAT32 do return float32ToString(x);
    FLOAT64 do return float64ToString(x)
  end case
end proc;

```

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

```

proc integerToString(i: INTEGER): STRING
  if i < 0 then return ['−'] ⊕ integerToString(−i) end if;
  q: INTEGER ← ⌊i/10⌋;
  r: INTEGER ← i − q×10;
  c: CHAR16 ← integerToChar16(r + char16ToInteger('0'));
  if q = 0 then return [c] else return integerToString(q) ⊕ [c] end if
end proc;

```

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

```

proc integerToStringWithSign(i: INTEGER): STRING
  if i ≥ 0 then return ['+'] ⊕ integerToString(i)
  else return ['−'] ⊕ integerToString(−i)
  end if
end proc;

```

float32ToString(x) converts a **Float32** *x* to a string using fixed-point notation if the absolute value of *x* is between 10^{-6} inclusive and 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 **−zero**₃₂ would become **+zero**₃₂).

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: LONG ∪ ULONG do return integer
ToString(o.value); , ... [8]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: STRING do return o; , ... [9]

```
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 $k \geq 1$, $10^{k-1} \leq s \leq 10^k$, $realToFloat32(s \times 10^{-k}) = x$, 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 $s \times 10^{-k}$ 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  $k \leq n \leq 21$  then return digits ⊕ repeat('0', n - k)
```

```
    elsif  $0 < n \leq 21$  then return digits[0 ... n - 1] ⊕ "." ⊕ digits[n ...]
```

```
    elsif  $-6 < n \leq 0$  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).

```

proc float64ToString(x: FLOAT64): STRING
  case x of
    {NaN64} do return "NaN";
    {+zero64, -zero64} do return "0";
    {+∞64} do return "Infinity";
    {-∞64} do return "-Infinity";
  NONZEROFINITEFLOAT64 do
    r: RATIONAL ← x.value;
    if r < 0 then return "-" ⊕ float64ToString(float64Negate(x))
    else
      Let  $n$ ,  $k$ , and  $s$  be integers such that  $k \geq 1$ ,  $10^{k-1} \leq s \leq 10^k$ ,  $realToFloat64(s \times 10^{-k}) = x$ , and  $k$  is as small as possible.
      note  $k$  is the number of digits in the decimal representation of  $s$ , that  $s$  is not divisible by 10, and that the least significant digit of  $s$  is not necessarily uniquely determined by 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  $s \times 10^{-k}$  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  $k \leq n \leq 21$  then return digits ⊕ repeat('0',  $n - k$ )
      elseif  $0 < n \leq 21$  then return digits[0 ...  $n - 1$ ] ⊕ "." ⊕ digits[ $n$  ...]
      elseif  $-6 < n \leq 0$  then return "0." ⊕ repeat('0',  $-n$ ) ⊕ digits
      else
        mantissa: STRING;
        if  $k = 1$  then mantissa ← digits
        else mantissa ← digits[0 ... 0] ⊕ "." ⊕ digits[1 ...]
        end if;
        return mantissa ⊕ "e" ⊕ integerToStringWithSign( $n - 1$ )
      end if
    end if
  end case
end proc;

```

Object to Qualified Name Conversion

objectToQualifiedName(o, phase) coerces an object o to a qualified name. If $phase$ is **compile**, only constant conversions are permitted.

```

proc objectToQualifiedName(o: OBJECT, phase: PHASE): QUALIFIEDNAME
  return public::(objectToString(o, phase))
end proc;

```

Object to Class Conversion

objectToClass(o) returns o converted to a non-null Class.

```

proc objectToClass(o: OBJECT): CLASS
  if  $o \in \text{CLASS}$  then return  $o$  else throw a TypeError exception end if
end proc;

```

Object to Attribute Conversion

objectToAttribute(o) returns o converted to an attribute.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toQualifiedName

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toQualifiedName(o, phase)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toQualifiedName(o: OBJECT, phase: PHASE): QUALIFIEDNAME ... [10]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: toAttribute ... [11]

```

proc objectToAttribute(o: OBJECT, phase: PHASE): ATTRIBUTE
  if o ∈ ATTRIBUTE then return o
  else
    note If o is not an attribute, try to call it with no arguments.
    a: OBJECT ← call(null, o, [], phase);
    if a ∈ ATTRIBUTE then return a else throw a TypeError exception end if
  end if
end proc;

```

Implicit Coercions

as(*o*, *c*, *silent*) attempts to implicitly coerce *o* to class *c*. If the coercion succeeds, *as* returns the coerced value. If not, then *as* returns **null** if *silent* is **true** and **null** is a member of type *c*; otherwise, *as* throws a *TypeError*.

The coercion always succeeds and returns *o* unchanged if *o* is already a member of class *c*. The value returned from *as* always is a member of class *c*.

```

proc as(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
  return c.as(o, c, silent)
end proc;

```

ordinaryAs(*o*, *c*) is the implementation of *as* for a native class that has *null* as a member, unless specified otherwise in the class's definition. Host classes may define a different procedure to perform this coercion.

```

proc ordinaryAs(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
  if o = null or is(o, c) then return o
  elseif silent then return null
  else throw a TypeError exception
  end if
end proc;

```

Attributes

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

```

proc combineAttributes(a: ATTRIBUTEOPTNOTFALSE, b: ATTRIBUTE): ATTRIBUTE
  if b = false then return false
  elseif a ∈ {none, true} then return b
  elseif b = true then return a
  elseif a ∈ NAMESPACE then
    if a = b then return a
    elseif b ∈ NAMESPACE then
      return COMPOUNDATTRIBUTE{namespaces: {a, b}, explicit: false, enumerable: false, dynamic: false,
        category: none, overrideMod: none, prototype: false, unused: false}
    else return COMPOUNDATTRIBUTE{namespaces: b.namespaces ∪ {a}, other fields from b}
    end if
  elseif 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.category ≠ none and b.category ≠ none and a.category ≠ b.category) 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, category: a.category ≠ none ? a.category : b.category,
        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;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*(*o*, *c*, *silent*)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*(*o*: OBJECT, *c*: CLASS, *silent*: BOOLEAN): OBJECT ... [12]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *ordinaryImplicitCoerceNull*(*o*, *c*) is the implementation of *implicitCoerce*

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: Such host

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: either also use *ordinaryImplicitCoerceNoNull* or

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *ordinaryImplicitCoerceNull*(*o*: OBJECT, *c*: CLASS, *silent*: BOOLEAN): OBJECT

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

```

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

```

Access Utilities

accessesOverlap(accesses1, accesses2) returns **true** if the two **ACCESSSET**s have a nonempty intersection.

```

proc accessesOverlap(accesses1: ACCESSSET, accesses2: ACCESSSET): BOOLEAN
  return accesses1 = accesses2 or accesses1 = readWrite or accesses2 = readWrite
end proc;

```

```

proc archetype(o: OBJECT): OBJECTOPT
  case o of
    UNDEFINED U NULL do return none;
    BOOLEAN do return Boolean.prototype;
    LONG do return long.prototype;
    ULONG do return ulong.prototype;
    FLOAT32 do return float.prototype;
    FLOAT64 do return Number.prototype;
    CHAR16 do return Character.prototype;
    STRING do return String.prototype;
    NAMESPACE do return Namespace.prototype;
    COMPOUNDATTRIBUTE do return Attribute.prototype;
    METHODCLOSURE do return Function.prototype;
    CLASS do return Class.prototype;
    SIMPLEINSTANCE U REGEXP U DATE U PACKAGE do return o.archetype
  end case
end proc;

```

archetypes(o) returns the set of *o*'s *archetypes*, not including *o* itself.

```

proc archetypes(o: OBJECT): OBJECT{}
  a: OBJECTOPT  $\leftarrow$  archetype(o);
  if a = none then return {} end if;
  return {a} U archetypes(a)
end proc;

```

o is an object that is known to have slot *id*. *findSlot(o, id)* returns that slot.

```

proc findSlot(o: OBJECT, id: INSTANCEVARIABLE): SLOT
  note o must be a SIMPLEINSTANCE or a METHODCLOSURE in order to have slots.
  matchingSlots: SLOT{}  $\leftarrow$  {s |  $\forall s \in o.slots$  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.

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *objectSuper*(*o*: **OBJECT**): **OBJECTOPT**
T ... [13]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *objectSupers*(*o*)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: superobjects,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *objectSupers*(*o*: **OBJECT**): **C** ... [14]

```

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 case
end proc;

```

`writeVariable(v, newValue, clearInitializer)` writes the value `newValue` into the mutable or immutable variable `v`. `newValue` is coerced to `v`'s type. If the `clearInitializer` flag is set, then the caller has just evaluated `v`'s initialiser and is supplying its result in `newValue`. In this case `writeVariable` atomically clears `v.initializer` while writing `v.value`. In all other cases the presence of an initialiser or an existing value will prevent an immutable variable's value from being written.

```

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

```

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 ∈ env do
    case frame of
      LOCALFRAME ∪ WITHFRAME do nothing;
      PARAMETERFRAME do return frame;
      PACKAGE ∪ CLASS do return none
    end case
  end for each;
  return none
end proc;

```

`getRegionalEnvironment(env)` returns all frames in `env` up to and including the first regional frame. A regional frame is either any frame other than a 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.

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: writeVariable(v, newValue, clearInitialiser)

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: clearInitialiser

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: v.initialiser

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: writeVariable(v: VARIABLE, newValue: OBJECT, clearInitialiser: BOOLEAN): OBJECT

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: coercedValue: OBJECT ← implicitCoerce(newValue, v.type, false); - ... [15]

```

proc getRegionalEnvironment(env: ENVIRONMENT): FRAME[]
  i: INTEGER ← 0;
  while env[i] ∈ LOCALFRAME ∪ WITHFRAME do i ← i + 1 end while;
  if env[i] ∈ CLASS then while i ≠ 0 and env[i] ∉ LOCALFRAME do i ← i - 1 end while
  end if;
  return env[0 ... i]
end proc;

```

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

```

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

```

getPackageFrame(*env*) returns the innermost package frame in *env*.

```

proc getPackageFrame(env: ENVIRONMENT): PACKAGE
  i: INTEGER ← 0;
  while env[i] ∉ PACKAGE do i ← i + 1 end while;
  note Every environment ends with a PACKAGE frame, so one will always be found.
  return env[i]
end proc;

```

Property Lookup

findLocalSingletonProperty(*o*, *multiname*, *access*) looks in *o* for a local singleton property with one of the names in *multiname* and access that includes *access*. If there is no such property, *findLocalSingletonProperty* returns **none**. If there is exactly one such property, *findLocalSingletonProperty* returns it. If there is more than one such property, *findLocalSingletonProperty* throws an error.

```

proc findLocalSingletonProperty(o: NONWITHFRAME ∪ SIMPLEINSTANCE ∪ REGEXP ∪ DATE, multiname: MULTINAME,
  access: ACCESS): SINGLETONPROPERTYOPT
  matchingLocalBindings: LOCALBINDING{} ← {b | ∀b ∈ o.localBindings such that
    b.qname ∈ multiname and accessesOverlap(b.accesses, access)};
  note If the same property was found via several different bindings b, then it will appear only once in the set
    matchingProperties.
  matchingProperties: SINGLETONPROPERTY{} ← {b.content | ∀b ∈ matchingLocalBindings};
  if matchingProperties = {} then return none
  elseif |matchingProperties| = 1 then return the one element of matchingProperties
  else
    throw a ReferenceError exception — this access is ambiguous because the bindings it found belong to several
      different local properties
  end if
end proc;

```

instancePropertyAccesses(*m*) returns instance property's ACCESSSET.

```

proc instancePropertyAccesses(m: INSTANCEPROPERTY): ACCESSSET
  case m of
    INSTANCEVARIABLE ∪ INSTANCEMETHOD do return readWrite;
    INSTANCEGETTER do return read;
    INSTANCESETTER do return write
  end case
end proc;

```

findLocalInstanceProperty(*c*, *multiname*, *accesses*) looks in class *c* for a local instance property with one of the names in *multiname* and accesses that have a nonempty intersection with *accesses*. If there is no such property, *findLocalInstanceProperty* returns **none**. If there is exactly one such property, *findLocalInstanceProperty* returns it. If there is more than one such property, *findLocalInstanceProperty* throws an error.

```

proc findLocalInstanceProperty(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
  matches: INSTANCEPROPERTY{} ← {m | ∀m ∈ c.instanceProperties such that m.multiname ∩ multiname ≠ {} and
    accessesOverlap(instancePropertyAccesses(m), accesses)};
  if matches = {} then return none
  elseif |matches| = 1 then return the one element of matches
  else
    throw a ReferenceError exception — this access is ambiguous because it found several different instance properties
    in the same class
  end if
end proc;

```

findArchetypeProperty(*o*, *multiname*, *access*, *flat*) looks in object *o* for any local or inherited property with one of the names in *multiname* and access that includes *access*. If *flat* is **true**, then inherited properties are not considered in the search except when *o* is a class. If it finds no property, *findArchetypeProperty* returns **none**. If it finds one property, *findArchetypeProperty* returns it. If it finds more than one property, *findArchetypeProperty* prefers the more local one in the list of *o*'s *superclasses* or *archetypes*; if two or more properties remain, the singleton one is preferred; if two or more properties still remain, *findArchetypeProperty* throws an error.

Note that *findArchetypeProperty*(*o*, *multiname*, *access*, *flat*) searches *o* itself rather than *o*'s class for properties. *findArchetypeProperty* will not find instance properties unless *o* is a class.

```

proc findArchetypeProperty(o: OBJECT, multiname: MULTINAME, access: ACCESS, flat: BOOLEAN): PROPERTYOPT
  m: PROPERTYOPT;
  case o of
    UNDEFINED ∪ NULL ∪ BOOLEAN ∪ LONG ∪ ULONG ∪ FLOAT32 ∪ FLOAT64 ∪ CHAR16 ∪ STRING ∪
      NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ METHODCLOSURE do
      m ← none;
    SIMPLEINSTANCE ∪ REGEXP ∪ DATE ∪ PACKAGE do
      m ← findLocalSingletonProperty(o, multiname, access);
    CLASS do m ← findClassProperty(o, multiname, access)
  end case;
  if m ≠ none then return m end if;
  if flat then return none end if;
  a: OBJECTOPT ← archetype(o);
  if a = none then return none end if;
  return findArchetypeProperty(a, multiname, access, flat)
end proc;

```

```

proc findClassProperty(c: CLASS, multiname: MULTINAME, access: ACCESS): PROPERTYOPT
  m: PROPERTYOPT ← findLocalSingletonProperty(c, multiname, access);
  if m = none then
    m ← findLocalInstanceProperty(c, multiname, access);
    if m = none then
      super: CLASSOPT ← c.super;
      if super ≠ none then m ← findClassProperty(super, multiname, access) end if
    end if
  end if;
  return m
end proc;

```

findBaseInstanceProperty(*c*, *multiname*, *accesses*) looks in class *c* and its ancestors for an instance property with one of the names in *multiname* and accesses that have a nonempty intersection with *accesses*. If there is no such property, *findBaseInstanceProperty* returns **none**. If there is exactly one such property, *findBaseInstanceProperty* returns it. If there is more than one such property, *findBaseInstanceProperty* prefers the one defined in the least specific class; if two or more properties still remain, *findBaseInstanceProperty* throws an error.

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: superobjects;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: findProperty(*o*: OBJECT, *multiname*: MULTINAME, *access*: ACCESS, *flat*: BOOLEAN):

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: CLASS do [16]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if *flat* and *o* ∉ CLASS then return none end if; [17]


```

proc findBaseInstanceProperty(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
  note Start from the root class (Object) and proceed through more specific classes that are ancestors of c.
  for each s ∈ ancestors(c) do
    m: INSTANCEPROPERTYOPT ← findLocalInstanceProperty(s, multiname, accesses);
    if m ≠ none then return m end if
  end for each;
  return none
end proc;

```

getDerivedInstanceProperty(*c*, *mBase*, *accesses*) returns the most derived instance property whose name includes that of *mBase* and whose accesses that have a nonempty intersection with *accesses*. The caller of *getDerivedInstanceProperty* ensures that such an instance property 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 getDerivedInstanceProperty(c: CLASS, mBase: INSTANCEPROPERTY, accesses: ACCESSSET): INSTANCEPROPERTY
  if some m ∈ c.instanceProperties satisfies mBase.multiname ⊆ m.multiname and
    accessesOverlap(instancePropertyAccesses(m), accesses) then
    return m
  else return getDerivedInstanceProperty(c.super, mBase, accesses)
  end if
end proc;

```

readImplicitThis(*env*) returns the value of implicit **this** to be used to access instance properties within a class's scope without using the **.** operator. An implicit **this** is well-defined only inside instance methods and constructors; *readImplicitThis* throws an error if there is no well-defined implicit **this** value or if an attempt is made to read it before it has been initialised.

```

proc readImplicitThis(env: ENVIRONMENT): OBJECT
  frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
  if frame = none then
    throw a ReferenceError exception — can't access instance properties 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 properties 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 properties 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 properties from within a constructor before the
      superconstructor has been called
  end if;
  return this
end proc;

```

```

proc hasProperty(o: OBJECT, qname: QUALIFIEDNAME, flat: BOOLEAN): BOOLEAN
  c: CLASS ← objectType(o);
  return findBaseInstanceProperty(c, {qname}, read) ≠ none or
    findBaseInstanceProperty(c, {qname}, write) ≠ none or
    findArchetypeProperty(o, {qname}, read, flat) ≠ none or
    findArchetypeProperty(o, {qname}, write, flat) ≠ none
end proc;

```

Reading

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

```

proc readReference( $r$ : OBJORREF,  $phase$ : PHASE): OBJECT
   $result$ : OBJECTOPT;
  case  $r$  of
    OBJECT do  $result \leftarrow r$ ;
    LEXICALREFERENCE do  $result \leftarrow lexicalRead(r.env, r.variableMultiname, phase)$ ;
    DOTREFERENCE do
       $result \leftarrow r.limit.read(r.base, r.limit, r.multiname, none, phase)$ ;
    BRACKETREFERENCE do  $result \leftarrow r.limit.bracketRead(r.base, r.limit, r.args, phase)$ 
  end case;
  if  $result \neq none$  then return  $result$ 
  else
    throw a ReferenceError exception — property not found, and no default value is available
  end if
end proc;

```

$dotRead(o, multiname, phase)$ reads and returns the value of the $multiname$ property of o . $dotRead$ throws an error if the property does not exist and no default value was available for it.

```

proc dotRead( $o$ : OBJECT,  $multiname$ : MULTINAME,  $phase$ : PHASE): OBJECT
   $limit$ : CLASS  $\leftarrow objectType(o)$ ;
   $result$ : OBJECTOPT  $\leftarrow limit.read(o, limit, multiname, none, phase)$ ;
  if  $result = none$  then
    throw a ReferenceError exception — property not found, and no default value is available
  end if;
  return  $result$ 
end proc;

```

$indexRead(o, i, phase)$ returns the value of $o[i]$ or **none** if no such property was found. An error is thrown if i is not a valid array index.

```

proc indexRead( $o$ : OBJECT,  $i$ : INTEGER,  $phase$ : PHASE): OBJECTOPT
  if  $i < 0$  or  $i \geq arrayLimit$  then throw a RangeError exception end if;
   $limit$ : CLASS  $\leftarrow objectType(o)$ ;
  return  $limit.bracketRead(o, limit, [i]_{ulong}, phase)$ 
end proc;

```

$ordinaryBracketRead(o, limit, args, phase)$ evaluates the expression $o[args]$ when o is a native object. Host objects may either also use $ordinaryBracketRead$ or choose a different procedure P to evaluate $o[args]$ by writing P into $objectType(o).bracketRead$.

$limit$ is used to handle the expression $super(o)[args]$, in which case $limit$ is the superclass of the class inside which the **super** expression appears. Otherwise, $limit$ is set to $objectType(o)$.

```

proc ordinaryBracketRead( $o$ : OBJECT,  $limit$ : CLASS,  $args$ : OBJECT[],  $phase$ : PHASE): OBJECTOPT
  if  $|args| \neq 1$  then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
   $qname$ : QUALIFIEDNAME  $\leftarrow objectToQualifiedName(args[0], phase)$ ;
  return  $limit.read(o, limit, \{qname\}, none, phase)$ 
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: $result \leftarrow r.limit.read(r.base, r.limit, r.propertyMultiname, none, phase)$;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: $qname$: **QUALIFIEDNAME** $\leftarrow toQualifiedName(args[0], phase)$;

```
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);
      PARAMETERFRAME ∪ LOCALFRAME do
        m: SINGLETONPROPERTYOPT ← findLocalSingletonProperty(frame, multiname, read);
        if m ≠ none then result ← readSingletonProperty(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 constant
                expressions;
            {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 property found with the name multiname
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

```
proc ordinaryRead(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, phase: PHASE):
```

```
  OBJECTOPT
```

```
  mBase: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(limit, multiname, read);
```

```
  if mBase ≠ none then return readInstanceProperty(o, limit, mBase, phase) end if;
```

```
  if limit ≠ objectType(o) then return none end if;
```

```
  flat: BOOLEAN ← env ≠ none and o ∈ CLASS;
```

```
  m: PROPERTYOPT ← findArchetypeProperty(o, multiname, read, flat);
```

```
  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;
```

```
          {run} do return undefined
```

```
        end case
```

```
      else return none
```

```
      end if;
```

```
  SINGLETONPROPERTY do return readSingletonProperty(m, phase);
```

```
  INSTANCEPROPERTY do
```

```
    if o ∉ CLASS or env = none then
```

```
      throw a ReferenceError exception — cannot read an instance property without supplying an instance
```

```
    end if;
```

```
    this: OBJECT ← readImplicitThis(env);
```

```
    return readInstanceProperty(this, objectType(this), m, phase)
```

```
  end case
```

```
end proc;
```

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

```
proc readInstanceSlot(o: OBJECT, qname: QUALIFIEDNAME, phase: PHASE): OBJECT
```

```
  c: CLASS ← objectType(o);
```

```
  mBase: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(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 readInstanceProperty(o, c, mBase, phase)
```

```
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *m*: PROPERTYOPT ← findProperty(*o*, multina
me, read, false);

```

proc readInstanceProperty(this: OBJECT, c: CLASS, mBase: INSTANCEPROPERTY, phase: PHASE): OBJECT
  m: INSTANCEPROPERTY ← getDerivedInstanceProperty(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 uninitialised const variables from constant
              expressions;
          {run} do
            throw an UninitializedError exception — cannot read a const instance variable before it is initialised
          end case
        end if;
        return v;
      INSTANCEMETHOD do
        slots: SLOT{} ← {new SLOT{id: ivarFunctionLength, value: realToFloat64(m.length)}};
        return METHODCLOSURE{this: this, method: m, slots: slots};
      INSTANCEGETTER do return m.call(this, phase);
      INSTANCESETTER do
        m cannot be an INSTANCESETTER because these are only represented as write-only properties.
      end case
    end case
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: an

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: variable from a constant
expression;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: INSTANCEMETHOD do return METH
ODCLOSURE{this: *this*, method: *m*};

```

proc readSingletonProperty(m: SINGLETONPROPERTY, phase: PHASE): OBJECT
case m of
  {forbidden} do
    throw a ReferenceError exception — cannot access a property defined in a scope outside the current region if
    any block inside the current region shadows it;
  DYNAMICVAR do
    if phase = compile then
      throw a ConstantError exception — constant expressions cannot read mutable variables
    end if;
    value: OBJECT ∪ UNINSTANTIATEDFUNCTION ← m.value;
    note value can be an UNINSTANTIATEDFUNCTION 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: VARIABLEVALUE ← m.value;
    case value of
      OBJECT do return value;
      {none} do
        if not m.immutable then throw an UninitializedError exception end if;
        note Try to run a const variable's initialiser if there is one.
        setupVariable(m);
        initializer: INITIALIZER ∪ {none, busy} ← m.initializer;
        if initializer ∈ {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.initializer ← busy;
        coercedValue: OBJECT;
        try
          newValue: OBJECT ← initializer(m.initializerEnv, compile);
          coercedValue ← writeVariable(m, newValue, true)
        catch x: SEMANTICEXCEPTION do
          note If initialisation failed, restore m.initializer to its original value so it can be tried later.
          m.initializer ← initializer;
          throw x
        end try;
        return coercedValue;
      UNINSTANTIATEDFUNCTION 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: ENVIRONMENTOPT ← m.env;
    if env = none then
      note An uninstantiated getter can only be found when phase = compile.
      throw a ConstantError 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 properties.
  end case
end case
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if not m.immutable then [18]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: INITIALISER ∪ {none, busy} ← m.initialiser; [19]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: m.initialiser ← busy;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: newValue: OBJECT ← initializer(m.initialiser Env, compile);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: m.initialiser

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: m.initialiser ← initializer;

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 constant expression.

```

proc writeReference( $r$ : OBJORREF,  $newValue$ : 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  $r.strict$ ,  $phase$ );
       $result$   $\leftarrow$  ok;
    DOTREFERENCE do
       $result$   $\leftarrow$   $r.limit.write(r.base, r.limit, r.multiname, none, true,  $newValue$ ,  $phase$ );$ 
```

```

    BRACKETREFERENCE do
       $result$   $\leftarrow$   $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  $\leftarrow$  objectType( $o$ );
   $result$ : {none, ok}  $\leftarrow$   $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  $i \geq arrayLimit$  then throw a RangeError exception end if;
   $limit$ : CLASS  $\leftarrow$  objectType( $o$ );
   $result$ : {none, ok}  $\leftarrow$   $limit.bracketWrite(o, limit, [iulong],  $newValue$ ,  $phase$ );$ 
```

```

  if  $result$  = none then
    throw a ReferenceError exception — property not found and could not be created
  end if
end proc;

```

```

proc ordinaryBracketWrite( $o$ : OBJECT,  $limit$ : CLASS,  $args$ : OBJECT[],  $newValue$ : OBJECT,  $phase$ : {run}): {none, ok}
  if  $|args| \neq 1$  then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
   $qname$ : QUALIFIEDNAME  $\leftarrow$  objectToQualifiedName( $args[0]$ ,  $phase$ );
  return  $limit.write(o, limit, \{qname\}, none, true,  $newValue$ ,  $phase$ )$ 
```

```

end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: $r.limit.write(r.base, r.limit,$
 $r.propertyMultiname, **none**, **true**, $newValue$,$
 $phase$);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: $qname$: **QUALIFIEDNAME** $\leftarrow$ toQual
ifiedName($args[0]$, $phase$);

```

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);
      PARAMETERFRAME ∪ LOCALFRAME do
        m: SINGLETONPROPERTYOPT ← findLocalSingletonProperty(frame, multiname, write);
        if m ≠ none then writeSingletonProperty(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 property found with the name multiname and one could not be
    created
end proc;

```



```

proc ordinaryWrite(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT,
  createIfMissing: BOOLEAN, newValue: OBJECT, phase: {run}): {none, ok}
  mBase: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(limit, multiname, write);
  if mBase ≠ none then
    writeInstanceProperty(o, limit, mBase, newValue, phase);
    return ok
  end if;
  if limit ≠ objectType(o) then return none end if;
  m: PROPERTYOPT ← findArchetypeProperty(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 findBaseInstanceProperty(objectType(o), {qname}, read) = none and
          findArchetypeProperty(o, {qname}, read, true) = none then
            createDynamicProperty(o, qname, false, true, newValue);
            return ok
          end if
        end if;
        return none;
      SINGLETONPROPERTY do writeSingletonProperty(m, newValue, phase); return ok;
      INSTANCEPROPERTY do
        if o ∉ CLASS or env = none then
          throw a ReferenceError exception — cannot write an instance property without supplying an instance
        end if;
        this: OBJECT ← readImplicitThis(env);
        writeInstanceProperty(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.

```

proc createDynamicProperty(o: SIMPLEINSTANCE ∪ DATE ∪ REGEXP ∪ PACKAGE, qname: QUALIFIEDNAME,
  sealed: BOOLEAN, enumerable: BOOLEAN, newValue: OBJECT)
  dv: DYNAMICVAR ← new DYNAMICVAR(value: newValue, sealed: sealed);
  o.localBindings ← o.localBindings ∪ {LOCALBINDING(qname: qname, accesses: readWrite, explicit: false,
    enumerable: enumerable, content: dv)};
  end proc;

```

```

proc writeInstanceProperty(this: OBJECT, c: CLASS, mBase: INSTANCEPROPERTY, newValue: OBJECT, phase: {run})
  m: INSTANCEPROPERTY ← getDerivedInstanceProperty(c, mBase, write);
  case m of
    INSTANCEVARIABLE do
      s: SLOT ← findSlot(this, m);
      coercedValue: OBJECT ← as(newValue, m.type, 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 properties.
    INSTANCESETTER do m.call(this, newValue, phase)
  end case
end proc;

proc writeSingletonProperty(m: SINGLETONPROPERTY, newValue: OBJECT, phase: {run})
  case m of
    {forbidden} do
      throw a ReferenceError exception — cannot access a property defined in a scope outside the current region 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 properties.
    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 proc;

```

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 constant expression.

```

proc deleteReference(r: OBJORREF, strict: BOOLEAN, phase: {run}): BOOLEAN
  result: BOOLEANOPT;
  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.multiname, 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;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: coercedValue: OBJECT ← implicitCoerce(newValue, m.type, false);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: result ← r.limit.delete(r.base, r.limit, r.propertyMultiname, none, phase);

```

proc ordinaryBracketDelete(o: OBJECT, limit: CLASS, args: OBJECT[], phase: {run}): BOOLEANOPT
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  qname: QUALIFIEDNAME ← objectToQualifiedName(args[0], phase);
  return limit.delete(o, limit, {qname}, none, phase)
end proc;

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);
      PARAMETERFRAME ∪ LOCALFRAME do
        if findLocalSingletonProperty(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;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *qname*: QUALIFIEDNAME ← *toQualifiedName*(*args*[0], *phase*);

```

proc ordinaryDelete(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, phase: {run}):
  BOOLEANOPT
  if findBaseInstanceProperty(limit, multiname, write) ≠ none then return false end if;
  if limit ≠ objectType(o) then return none end if;
  m: PROPERTYOPT ← findArchetypeProperty(o, multiname, write, true);
  case m of
    {none} do return none;
    {forbidden} do
      throw a ReferenceError exception — cannot access a property defined in a scope outside the current region 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 ∈ o.localBindings such that b.qname ∉ multiname or b.content ≠ m};
        return true
      end if;
    INSTANCEPROPERTY do
      if o ∉ CLASS or env = none then return false end if;
      readImplicitThis(env);
      return false
    end case
  end case
end proc;

```

Enumerating

```

proc ordinaryEnumerate(o: OBJECT): OBJECT{}
  e1: OBJECT{} ← enumerateInstanceProperties(objectType(o));
  e2: OBJECT{} ← enumerateArchetypeProperties(o);
  return e1 ∪ e2
end proc;

proc enumerateInstanceProperties(c: CLASS): OBJECT{}
  e: OBJECT{} ← {};
  for each m ∈ c.instanceProperties do
    if m.enumerable then
      e ← e ∪ {qname.id | ∀ qname ∈ m.multiname such that qname.namespace = public}
    end if
  end for each;
  super: CLASSOPT ← c.super;
  if super = none then return e
  else return e ∪ enumerateInstanceProperties(super)
  end if
end proc;

proc enumerateArchetypeProperties(o: OBJECT): OBJECT{}
  e: OBJECT{} ← {};
  for each a ∈ {o} ∪ archetypes(o) do
    if a ∈ BINDINGOBJECT then e ← e ∪ enumerateSingletonProperties(a) end if
  end for each;
  return e
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: enumerateProperties(*o*: OBJECT): OBJECT{}

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: for each *s* ∈ {*o*} ∪ objectSupers(*o*) ← ... [20]

```

proc enumerateSingletonProperties(o: BINDINGOBJECT): OBJECT{}
  e: OBJECT{} ← {};
  for each b ∈ o.localBindings do
    if b.enumerable and b.qname.namespace = public then e ← e ∪ {b.qname.id} end if
  end for each;
  if o ∈ CLASS then
    super: CLASSOPT ← o.super;
    if super ≠ none then e ← e ∪ enumerateSingletonProperties(super) end if
  end if;
  return e
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Creating Instances

```

proc createSimpleInstance(c: CLASS, archetype: OBJECTOPT,
  call: (OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none},
  construct: (SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT) ∪ {none}, env: ENVIRONMENTOPT):
  SIMPLEINSTANCE
  slots: SLOT{} ← {};
  for each s ∈ ancestors(c) do
    for each m ∈ s.instanceProperties 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;
  return new SIMPLEINSTANCE{localBindings: {}, archetype: archetype, sealed: not c.dynamic, type: c, slots: slots,
    call: call, construct: construct, env: env}
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: OBJECTOPT,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: super,

Adding Local Definitions

```

proc defineSingletonProperty(env: ENVIRONMENT, id: STRING, namespaces: NAMESPACE {},
  overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, accesses: ACCESSSET, m: SINGLETONPROPERTY):
  MULTINAME
  innerFrame: NONWITHFRAME ← 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 property 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 ∉ WITHFRAME 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 a property defined in an outer scope within
      the same region
    end if
  end for each;
  newBindings: LOCALBINDING {} ← {LOCALBINDING(qname: qname, accesses: accesses, explicit: explicit,
    enumerable: true, content: m) | ∀ 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.
  newForbiddenBindings: LOCALBINDING {} ← {LOCALBINDING(qname: qname, accesses: accesses, explicit: true,
    enumerable: true, content: forbidden) | ∀ qname ∈ multiname};
  for each frame ∈ regionalEnv[1 ...] do
    note Since frame ∉ CLASS here, a CLASS frame never gets a forbidden binding.
    if frame ∉ WITHFRAME 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 **PARAMETERFRAME** if there is already a parameter with the same name; a *function* hoisted variable is never hoisted into the **PARAMETERFRAME** 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 − 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 − 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(qname: qname,
      accesses: readWrite, explicit: false, enumerable: true, content: v)};
    return v
  elseif |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: SINGLETONPROPERTY ← 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;

```

Adding Instance Definitions

```

proc searchForOverrides(c: CLASS, multiname: MULTINAME, accesses: ACCESSSET): INSTANCEPROPERTYOPT
  mBase: INSTANCEPROPERTYOPT ← none;
  s: CLASSOPT ← c.super;
  if s ≠ none then
    for each qname ∈ multiname do
      m: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(s, {qname}, accesses);
      if mBase = none then mBase ← m
      elseif 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 defineInstanceProperty(c: CLASS, cxt: CONTEXT, id: STRING, namespaces: NAMESPACE {},
  overrideMod: OVERRIDEMODIFIER, explicit: BOOLEAN, m: INSTANCEPROPERTY): INSTANCEPROPERTYOPT
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 ← instancePropertyAccesses(m);
requestedMultiname: MULTINAME ← {ns::id | ∀ns ∈ namespaces};
openMultiname: MULTINAME ← {ns::id | ∀ns ∈ cxt.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: INSTANCEPROPERTYOPT ← searchForOverrides(c, searchedMultiname, accesses);
mOverridden: INSTANCEPROPERTYOPT ← none;
if mBase ≠ none then
  mOverridden ← getDerivedInstanceProperty(c, mBase, accesses);
  definedMultiname ← mOverridden.multiname;
  if not (requestedMultiname ⊆ definedMultiname) then
    throw a DefinitionError exception — cannot extend the set of a property's namespaces when overriding it
  end if;
  goodKind: BOOLEAN;
  case m of
    INSTANCEVARIABLE do goodKind ← mOverridden ∈ INSTANCEVARIABLE;
    INSTANCEGETTER do
      goodKind ← mOverridden ∈ INSTANCEVARIABLE ∪ INSTANCEGETTER;
    INSTANCESSETTER do
      goodKind ← mOverridden ∈ INSTANCEVARIABLE ∪ INSTANCESSETTER;
    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 property
  end if
end if;
if some m2 ∈ c.instanceProperties satisfies m2.multiname ∩ definedMultiname ≠ {} and
  accessesOverlap(instancePropertyAccesses(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 property 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 property
    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 property
        end if;
        {undefined} do nothing
    end case;
    m.multiname ← definedMultiname;
    c.instanceProperties ← c.instanceProperties ∪ {m};
    return mOverridden
end proc;

```

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 c = PrototypeFunction then
        prototype: OBJECT ← Object.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 instantiateProperty(m: SINGLETONPROPERTY, env: ENVIRONMENT): SINGLETONPROPERTY
  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,
        initializer: m.initializer, initializerEnv: 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 PROPERTYTRANSLATION
  from: SINGLETONPROPERTY,
  to: SINGLETONPROPERTY
end tuple;

proc instantiateLocalFrame(frame: LOCALFRAME, env: ENVIRONMENT): LOCALFRAME
  instantiatedFrame: LOCALFRAME ← new LOCALFRAME{localBindings: {}};
  properties: SINGLETONPROPERTY{} ← {b.content | ∀ b ∈ frame.localBindings};
  propertyTranslations: PROPERTYTRANSLATION{} ← {PROPERTYTRANSLATION{from: m,
    to: instantiateProperty(m, [instantiatedFrame] ⊕ env) | ∀ m ∈ properties};
  proc translateProperty(m: SINGLETONPROPERTY): SINGLETONPROPERTY
    mi: PROPERTYTRANSLATION ← the one element mi ∈ propertyTranslations that satisfies mi.from = m;
    return mi.to
  end proc;
  instantiatedFrame.localBindings ← {LOCALBINDING{content: translateProperty(b.content), other fields from b} |
    ∀ b ∈ frame.localBindings};
  return instantiatedFrame
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: *m*.initialiser, initialiserEnv: *env*};

```

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 properties will contain the set of all SINGLETONPROPERTY records found in the frame.
  properties: SINGLETONPROPERTY {} ← {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 SINGLETONPROPERTY records to properties.
  for each p ∈ frame.parameters do properties ← properties ∪ {p.var} end for each;
  rest: VARIABLEOPT ← frame.rest;
  if rest ≠ none then properties ← properties ∪ {rest} end if;
  propertyTranslations: PROPERTYTRANSLATION {} ← {PROPERTYTRANSLATION{from: m,
    to: instantiateProperty(m, [instantiatedFrame] ⊕ env)} | ∀ m ∈ properties};
  proc translateProperty(m: SINGLETONPROPERTY): SINGLETONPROPERTY
    mi: PROPERTYTRANSLATION ← the one element mi ∈ propertyTranslations that satisfies mi.from = m;
    return mi.to
  end proc;
  instantiatedFrame.localBindings ← {LOCALBINDING{content: translateProperty(b.content), other fields from b} |
    ∀ b ∈ frame.localBindings};
  instantiatedFrame.parameters ← [PARAMETER{var: translateProperty(op.var), default: op.default} |
    ∀ op ∈ frame.parameters];
  if rest = none then instantiatedFrame.rest ← none
  else instantiatedFrame.rest ← translateProperty(rest)
  end if;
  return instantiatedFrame
end proc;

```

Sealing

```

proc sealObject(o: OBJECT)
  if o ∈ SIMPLEINSTANCE ∪ REGEXP ∪ DATE ∪ PACKAGE then o.sealed ← true end if
end proc;

proc sealAllLocalProperties(o: OBJECT)
  if o ∈ BINDINGOBJECT then
    for each b ∈ o.localBindings do
      m: SINGLETONPROPERTY ← b.content;
      if m ∈ DYNAMICVAR then m.sealed ← true end if
    end for each
  end if
end proc;

proc sealLocalProperty(o: OBJECT, qname: QUALIFIEDNAME)
  c: CLASS ← objectType(o);
  if findBaseInstanceProperty(c, {qname}, read) = none and
    findBaseInstanceProperty(c, {qname}, write) = none and o ∈ BINDINGOBJECT then
    matchingProperties: SINGLETONPROPERTY {} ← {b.content | ∀ b ∈ o.localBindings such that b.qname = qname};
    for each m ∈ matchingProperties do
      if m ∈ DYNAMICVAR then m.sealed ← true end if
    end for each
  end if
end proc;

```

Standard Class Utilities

```
proc defaultArg(args: OBJECT[], n: INTEGER, default: OBJECT): OBJECT
  if n < |args| then return args[n] else return default end if
end proc;

proc stdConstBinding(qname: QUALIFIEDNAME, type: CLASS, value: OBJECT): LOCALBINDING
  return LOCALBINDING{qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
    new VARIABLE{type: type, value: value, immutable: true, setup: none, initializer: none}}
end proc;

proc stdExplicitConstBinding(qname: QUALIFIEDNAME, type: CLASS, value: OBJECT): LOCALBINDING
  return LOCALBINDING{qname: qname, accesses: readWrite, explicit: true, enumerable: false, content:
    new VARIABLE{type: type, value: value, immutable: true, setup: none, initializer: none}}
end proc;

proc stdFunction(qname: QUALIFIEDNAME, call: OBJECT × SIMPLEINSTANCE × OBJECT[] × PHASE → OBJECT,
  length: INTEGER): LOCALBINDING
  slots: SLOT{} ← {new SLOT{id: ivarFunctionLength, value: realToFloat64(length)}};
  f: SIMPLEINSTANCE ← new SIMPLEINSTANCE{localBindings: {}, archetype: FunctionPrototype, sealed: true,
    type: Function, slots: slots, call: call, construct: none, env: none};
  return LOCALBINDING{qname: qname, accesses: readWrite, explicit: false, enumerable: false, content:
    new VARIABLE{type: Function, value: f, immutable: true, setup: none, initializer: none}}
end proc;
```

Expressions

Syntax

β ∈ {allowIn, noIn}

Terminal Actions

Name[Identifier]: STRING;
Value[Number]: GENERALNUMBER;
Value[String]: STRING;
Body[RegularExpression]: STRING;
Flags[RegularExpression]: STRING;

Identifiers

Syntax

Identifier ⇒
 Identifier
 | get
 | set

Semantics

Name[Identifier]: STRING;
Name[Identifier ⇒ Identifier] = Name[Identifier];
Name[Identifier ⇒ get] = “get”;
Name[Identifier ⇒ set] = “set”;

Qualified Identifiers

Syntax

```

Qualifier ⇒
  Identifier
  | ReservedNamespace

SimpleQualifiedIdentifier ⇒
  Identifier
  | Qualifier :: Identifier

ExpressionQualifiedIdentifier ⇒ ParenExpression :: Identifier

QualifiedIdentifier ⇒
  SimpleQualifiedIdentifier
  | ExpressionQualifiedIdentifier

```

Validation

```
OpenNamespaces[Qualifier]: NAMESPACE{};
```

```

proc Validate[Qualifier] (cxt: CONTEXT, env: ENVIRONMENT)
  [Qualifier ⇒ Identifier] do OpenNamespaces[Qualifier] ← cxt.openNamespaces;
  [Qualifier ⇒ ReservedNamespace] do Validate[ReservedNamespace](cxt, env)
end proc;

```

```
OpenNamespaces[SimpleQualifiedIdentifier]: NAMESPACE{};
```

```

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

```

```

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

```

Validate[*QualifiedIdentifier*] (*cxt*: *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*.

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *Name*[*Identifier* ⇒ *exclude*] = “
exclude”;

... [22]

Evaluation

```

proc Eval[Qualifier] (env: ENVIRONMENT, phase: PHASE): NAMESPACE
  [Qualifier  $\Rightarrow$  Identifier] do
    multiname: MULTINAME  $\leftarrow$  {ns::(Name[Identifier]) |  $\forall ns \in$  OpenNamespaces[Qualifier]};
    a: OBJECT  $\leftarrow$  lexicalRead(env, multiname, phase);
    if a  $\notin$  NAMESPACE then
      throw a TypeError exception — the qualifier must be a namespace
    end if;
    return a;
  [Qualifier  $\Rightarrow$  ReservedNamespace] do return Eval[ReservedNamespace](env, phase)
end proc;

proc Eval[SimpleQualifiedIdentifier] (env: ENVIRONMENT, phase: PHASE): MULTINAME
  [SimpleQualifiedIdentifier  $\Rightarrow$  Identifier] do
    return {ns::(Name[Identifier]) |  $\forall ns \in$  OpenNamespaces[SimpleQualifiedIdentifier]};
  [SimpleQualifiedIdentifier  $\Rightarrow$  Qualifier :: Identifier] do
    q: NAMESPACE  $\leftarrow$  Eval[Qualifier](env, phase);
    return {q::(Name[Identifier])}
  end proc;

proc Eval[ExpressionQualifiedIdentifier  $\Rightarrow$  ParenExpression :: Identifier]
  (env: ENVIRONMENT, phase: PHASE): MULTINAME
  q: OBJECT  $\leftarrow$  readReference(Eval[ParenExpression](env, phase), phase);
  if q  $\notin$  NAMESPACE then throw a TypeError exception — the qualifier must be a namespace
  end if;
  return {q::(Name[Identifier])}
end proc;

proc Eval[QualifiedIdentifier] (env: ENVIRONMENT, phase: PHASE): MULTINAME
  [QualifiedIdentifier  $\Rightarrow$  SimpleQualifiedIdentifier] do
    return Eval[SimpleQualifiedIdentifier](env, phase);
  [QualifiedIdentifier  $\Rightarrow$  ExpressionQualifiedIdentifier] do
    return Eval[ExpressionQualifiedIdentifier](env, phase)
  end proc;

```

Primary Expressions

Syntax

```

PrimaryExpression  $\Rightarrow$ 
  null
  | true
  | false
  | Number
  | String
  | this
  | RegularExpression
  | ReservedNamespace
  | ParenListExpression
  | ArrayLiteral
  | ObjectLiteral
  | FunctionExpression

```

```

ReservedNamespace ⇒
  public
| private

ParenExpression ⇒ ( AssignmentExpressionallowin )

ParenListExpression ⇒
  ParenExpression
| ( ListExpressionallowin , AssignmentExpressionallowin )

```

Validation

```

proc Validate[PrimaryExpression] (cxt: CONTEXT, env: ENVIRONMENT)
  [PrimaryExpression ⇒ null] do nothing;
  [PrimaryExpression ⇒ true] do nothing;
  [PrimaryExpression ⇒ false] do nothing;
  [PrimaryExpression ⇒ Number] do nothing;
  [PrimaryExpression ⇒ String] do nothing;
  [PrimaryExpression ⇒ this] do
    frame: PARAMETERFRAMEOPT ← getEnclosingParameterFrame(env);
    if frame = none then
      if cxt.strict then
        throw a SyntaxError exception — this can be used outside a function only in non-strict mode
      end if
    elseif frame.kind = plainFunction then
      throw a SyntaxError exception — this function does not define this
    end if;
  [PrimaryExpression ⇒ RegularExpression] do nothing;
  [PrimaryExpression ⇒ ReservedNamespace] do Validate[ReservedNamespace](cxt, env);
  [PrimaryExpression ⇒ ParenListExpression] do
    Validate[ParenListExpression](cxt, env);
  [PrimaryExpression ⇒ ArrayLiteral] do Validate[ArrayLiteral](cxt, env);
  [PrimaryExpression ⇒ ObjectLiteral] do Validate[ObjectLiteral](cxt, env);
  [PrimaryExpression ⇒ FunctionExpression] do Validate[FunctionExpression](cxt, env)
end proc;

```

```

proc Validate[ReservedNamespace] (cxt: CONTEXT, env: ENVIRONMENT)
  [ReservedNamespace ⇒ public] do nothing;
  [ReservedNamespace ⇒ private] do
    if getEnclosingClass(env) = none then
      throw a SyntaxError exception — private is meaningful only inside a class
    end if
  end if;
end proc;

```

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

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

Setup

```

proc Setup[PrimaryExpression] ()
  [PrimaryExpression ⇒ null] do nothing;

```

```

[PrimaryExpression ⇒ true] do nothing;
[PrimaryExpression ⇒ false] do nothing;
[PrimaryExpression ⇒ Number] do nothing;
[PrimaryExpression ⇒ String] do nothing;
[PrimaryExpression ⇒ this] do nothing;
[PrimaryExpression ⇒ RegularExpression] do nothing;
[PrimaryExpression ⇒ ReservedNamespace] do nothing;
[PrimaryExpression ⇒ ParenListExpression] do Setup[ParenListExpression]();
[PrimaryExpression ⇒ ArrayLiteral] do Setup[ArrayLiteral]();
[PrimaryExpression ⇒ ObjectLiteral] do Setup[ObjectLiteral]();
[PrimaryExpression ⇒ FunctionExpression] do Setup[FunctionExpression]()
end proc;

```

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 ⇒ 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
  return Body[RegularExpression] ⊕ “#” ⊕ Flags[RegularExpression];
[PrimaryExpression ⇒ ReservedNamespace] do
  return Eval[ReservedNamespace](env, phase);
[PrimaryExpression ⇒ ParenListExpression] do
  return Eval[ParenListExpression](env, phase);
[PrimaryExpression ⇒ ArrayLiteral] do return Eval[ArrayLiteral](env, phase);
[PrimaryExpression ⇒ ObjectLiteral] do return Eval[ObjectLiteral](env, phase);

```



```

    [PrimaryExpression ⇒ FunctionExpression] do
        return Eval[FunctionExpression](env, phase)
    end proc;

proc Eval[ReservedNamespace] (env: ENVIRONMENT, phase: PHASE): NAMESPACE
    [ReservedNamespace ⇒ public] do return public;
    [ReservedNamespace ⇒ private] do
        c: CLASSOPT ← getEnclosingClass(env);
        note Validate already ensured that c ≠ none.
        return c.privateNamespace
    end proc;

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

proc Eval[ParenListExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
    [ParenListExpression ⇒ ParenExpression] do return Eval[ParenExpression](env, phase);
    [ParenListExpression ⇒ ( ListExpressionallowin , AssignmentExpressionallowin )] do
        readReference(Eval[ListExpressionallowin](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;

```

Function Expressions

Syntax

```

FunctionExpression ⇒
    function FunctionCommon
    | function Identifier FunctionCommon

```

Validation

```

F[FunctionExpression]: UNINSTANTIATEDFUNCTION;

proc Validate[FunctionExpression] (cxt: CONTEXT, env: ENVIRONMENT)
    [FunctionExpression ⇒ function FunctionCommon] do
        kind: STATICFUNCTIONKIND ← plainFunction;
        if not cxt.strict and Plain[FunctionCommon] then kind ← uncheckedFunction
        end if;
        F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](cxt, env, kind);
    end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: busy;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: none;

```
[FunctionExpression => function Identifier FunctionCommon] do
  v: VARIABLE ← new VARIABLE⟨type: Function, value: none, immutable: true, setup: none, initializer: busy⟩;
  b: LOCALBINDING ← LOCALBINDING⟨qname: public::⟨Name[Identifier]⟩, accesses: readWrite, explicit: false,
    enumerable: true, content: v⟩;
  compileFrame: LOCALFRAME ← new LOCALFRAME⟨localBindings: {b}⟩;
  kind: STATICFUNCTIONKIND ← plainFunction;
  if not ctx.strict and Plain[FunctionCommon] then kind ← uncheckedFunction
  end if;
  F[FunctionExpression] ← ValidateStaticFunction[FunctionCommon](ctx, [compileFrame] ⊕ env, kind)
end proc;
```

Setup

```
proc Setup[FunctionExpression] ()
  [FunctionExpression => function FunctionCommon] do Setup[FunctionCommon]();
  [FunctionExpression => function Identifier FunctionCommon] do Setup[FunctionCommon]()
end proc;
```

Evaluation

```
proc Eval[FunctionExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [FunctionExpression => function 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;
    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, initializer: none⟩;
    b: LOCALBINDING ← LOCALBINDING⟨qname: public::⟨Name[Identifier]⟩, accesses: readWrite, explicit: false,
      enumerable: true, content: v⟩;
    runtimeFrame: LOCALFRAME ← new LOCALFRAME⟨localBindings: {b}⟩;
    f: SIMPLEINSTANCE ← instantiateFunction(F[FunctionExpression], [runtimeFrame] ⊕ env);
    v.value ← f;
    return f
  end proc;
```

Object Literals

Syntax

```
ObjectLiteral => { FieldList }

FieldList =>
  «empty»
  | NonemptyFieldList

NonemptyFieldList =>
  LiteralField
  | LiteralField , NonemptyFieldList
```

LiteralField $\Rightarrow$ *FieldName* : *AssignmentExpression*^{allowin}

FieldName $\Rightarrow$
 QualifiedIdentifier
 | **String**
 | **Number**
 | *ParenExpression*

Validation

```
proc Validate[ObjectLiteral  $\Rightarrow$  { FieldList }](cxt: CONTEXT, env: ENVIRONMENT)
  Validate[FieldList](cxt, env)
end proc;
```

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

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

```
proc Validate[LiteralField  $\Rightarrow$  FieldName : AssignmentExpressionallowin](cxt: CONTEXT, env: ENVIRONMENT)
  Validate[FieldName](cxt, env);
  Validate[AssignmentExpressionallowin](cxt, env)
end proc;
```

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

Setup

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

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

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

```
proc Setup[LiteralField  $\Rightarrow$  FieldName : AssignmentExpressionallowin]()
  Setup[FieldName]();
  Setup[AssignmentExpressionallowin]()
end proc;
```

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

Evaluation

```
proc Eval[ObjectLiteral  $\Rightarrow$  { FieldList }](env: ENVIRONMENT, phase: PHASE): OBJORREF
  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  $\leftarrow$  Object.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})
  multiname: MULTINAME ← Eval[FieldName](env, phase);
  value: OBJECT ← readReference(Eval[AssignmentExpressionallowIn](env, phase), phase);
  dotWrite(o, multiname, value, phase)
end proc;

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

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: [FieldName ⇒ String] do return {toQualifi
edName(Value[String], *phase*)}; .

... [23]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: return {toQualifiedName(*a*, *phase*)}

Array Literals

Syntax

ArrayLiteral ⇒ [*ElementList*]

ElementList ⇒
 «empty»
 | *LiteralElement*
 | , *ElementList*
 | *LiteralElement* , *ElementList*

LiteralElement ⇒ *AssignmentExpression*^{allowIn}

Validation

```
proc Validate[ArrayLiteral ⇒ [ ElementList ]] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[ElementList](cxt, env)
end proc;
```

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

```
proc Validate[LiteralElement ⇒ AssignmentExpressionallowIn] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[AssignmentExpressionallowIn](cxt, 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  $\Rightarrow$  AssignmentExpressionallowIn] ()
  Setup[AssignmentExpressionallowIn]()
end proc;

```

Evaluation

```

proc Eval[ArrayLiteral  $\Rightarrow$  [ 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  $\leftarrow$  Array.construct([], phase);
  length: INTEGER  $\leftarrow$  Eval[ElementList](env, 0, o, phase);
  if length > arrayLimit then throw a RangeError exception end if;
  dotWrite(o, {arrayPrivate::"length"}, lengthulong, phase);
  return o
end proc;

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

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

```

Super Expressions

Syntax

```

SuperExpression  $\Rightarrow$ 
  super
  | super ParenExpression

```

Validation

```

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

```

Setup

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

Evaluation

```

proc Eval[SuperExpression] (env: ENVIRONMENT, phase: PHASE): OBJOPTIONALLIMIT
  [SuperExpression ⇒ super] do
    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 ← as(o, limit, false);
  if coerced = null then return null end if;
  return LIMITEDINSTANCE⟨instance: coerced, limit: limit⟩
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *coerced*: OBJECT ← *implicitCoerce*(*o*, *limit*, **false**);

Postfix Expressions

Syntax

```

PostfixExpression ⇒
  AttributeExpression
| FullPostfixExpression
| ShortNewExpression

AttributeExpression ⇒
  SimpleQualifiedIdentifier
| AttributeExpression PropertyOperator
| AttributeExpression Arguments

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

FullNewExpression ⇒ new FullNewSubexpression Arguments

FullNewSubexpression ⇒
  PrimaryExpression
| QualifiedIdentifier
| FullNewExpression
| FullNewSubexpression PropertyOperator
| SuperExpression PropertyOperator

ShortNewExpression ⇒ new ShortNewSubexpression

ShortNewSubexpression ⇒
  FullNewSubexpression
| ShortNewExpression

```

Validation

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

Strict[*AttributeExpression*]: BOOLEAN;

```

proc Validate[AttributeExpression] (cxt: CONTEXT, env: ENVIRONMENT)
  [AttributeExpression  $\Rightarrow$  SimpleQualifiedIdentifier] do
    Validate[SimpleQualifiedIdentifier](cxt, env);
    Strict[AttributeExpression]  $\leftarrow$  cxt.strict;
  [AttributeExpression0  $\Rightarrow$  AttributeExpression1 PropertyOperator] do
    Validate[AttributeExpression1](cxt, env);
    Validate[PropertyOperator](cxt, env);
  [AttributeExpression0  $\Rightarrow$  AttributeExpression1 Arguments] do
    Validate[AttributeExpression1](cxt, env);
    Validate[Arguments](cxt, env)
end proc;

```

Strict[*FullPostfixExpression*]: **BOOLEAN**;

```

proc Validate[FullPostfixExpression] (cxt: CONTEXT, env: ENVIRONMENT)
  [FullPostfixExpression  $\Rightarrow$  PrimaryExpression] do
    Validate[PrimaryExpression](cxt, env);
  [FullPostfixExpression  $\Rightarrow$  ExpressionQualifiedIdentifier] do
    Validate[ExpressionQualifiedIdentifier](cxt, env);
    Strict[FullPostfixExpression]  $\leftarrow$  cxt.strict;
  [FullPostfixExpression  $\Rightarrow$  FullNewExpression] do
    Validate[FullNewExpression](cxt, env);
  [FullPostfixExpression0  $\Rightarrow$  FullPostfixExpression1 PropertyOperator] do
    Validate[FullPostfixExpression1](cxt, env);
    Validate[PropertyOperator](cxt, env);
  [FullPostfixExpression  $\Rightarrow$  SuperExpression PropertyOperator] do
    Validate[SuperExpression](cxt, env);
    Validate[PropertyOperator](cxt, env);
  [FullPostfixExpression0  $\Rightarrow$  FullPostfixExpression1 Arguments] do
    Validate[FullPostfixExpression1](cxt, env);
    Validate[Arguments](cxt, env);
  [FullPostfixExpression  $\Rightarrow$  PostfixExpression [no line break] ++] do
    Validate[PostfixExpression](cxt, env);
  [FullPostfixExpression  $\Rightarrow$  PostfixExpression [no line break] --] do
    Validate[PostfixExpression](cxt, env)
end proc;

```

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

Strict[*FullNewSubexpression*]: **BOOLEAN**;

```

proc Validate[FullNewSubexpression] (cxt: CONTEXT, env: ENVIRONMENT)
  [FullNewSubexpression  $\Rightarrow$  PrimaryExpression] do Validate[PrimaryExpression](cxt, env);
  [FullNewSubexpression  $\Rightarrow$  QualifiedIdentifier] do
    Validate[QualifiedIdentifier](cxt, env);
    Strict[FullNewSubexpression]  $\leftarrow$  cxt.strict;
  [FullNewSubexpression  $\Rightarrow$  FullNewExpression] do Validate[FullNewExpression](cxt, env);

```



```

[FullNewSubexpression0 ⇒ FullNewSubexpression1 PropertyOperator] do
  Validate[FullNewSubexpression1](cxt, env);
  Validate[PropertyOperator](cxt, env);
[FullNewSubexpression ⇒ SuperExpression PropertyOperator] do
  Validate[SuperExpression](cxt, env);
  Validate[PropertyOperator](cxt, env)
end proc;

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

Validate[ShortNewSubexpression] (cxt: 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

```

proc Eval[PostfixExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [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;

proc Eval[AttributeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [AttributeExpression ⇒ SimpleQualifiedIdentifier] do
    m: MULTINAME ← Eval[SimpleQualifiedIdentifier](env, phase);
    return LEXICALREFERENCE{env: env, variableMultiname: m, strict: Strict[AttributeExpression]};
  [AttributeExpression0 ⇒ AttributeExpression1 PropertyOperator] do
    a: OBJECT ← readReference(Eval[AttributeExpression1](env, phase), phase);
    return Eval[PropertyOperator](env, a, phase);

```

```

[AttributeExpression0 ⇒ AttributeExpression1 Arguments] do
  r: OBJORREF ← Eval[AttributeExpression1](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: env, variableMultiname: m, strict: Strict[FullPostfixExpression]};
  [FullPostfixExpression ⇒ FullNewExpression] do
    return Eval[FullNewExpression](env, phase);
  [FullPostfixExpression0 ⇒ FullPostfixExpression1 PropertyOperator] do
    a: OBJECT ← readReference(Eval[FullPostfixExpression1](env, phase), phase);
    return Eval[PropertyOperator](env, a, phase);
  [FullPostfixExpression ⇒ SuperExpression PropertyOperator] do
    a: OBJOPTIONALLIMIT ← Eval[SuperExpression](env, phase);
    return Eval[PropertyOperator](env, a, phase);
  [FullPostfixExpression0 ⇒ FullPostfixExpression1 Arguments] do
    r: OBJORREF ← Eval[FullPostfixExpression1](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 constant expressions
    end if;
    r: OBJORREF ← Eval[PostfixExpression](env, phase);
    a: OBJECT ← readReference(r, phase);
    b: OBJECT ← plus(a, phase);
    c: OBJECT ← add(b, 1.0164, phase);
    writeReference(r, c, phase);
    return b;
  end

```

```

[FullPostfixExpression  $\Rightarrow$  PostfixExpression [no line break] --] do
  if phase = compile then
    throw a ConstantError exception — -- cannot be used in constant expressions
  end if;
  r: OBJORREF  $\leftarrow$  Eval[PostfixExpression](env, phase);
  a: OBJECT  $\leftarrow$  readReference(r, phase);
  b: OBJECT  $\leftarrow$  plus(a, phase);
  c: OBJECT  $\leftarrow$  subtract(b, 1.064, phase);
  writeReference(r, c, phase);
  return b
end proc;

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

proc Eval[FullNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [FullNewSubexpression  $\Rightarrow$  PrimaryExpression] do
    return Eval[PrimaryExpression](env, phase);
  [FullNewSubexpression  $\Rightarrow$  QualifiedIdentifier] do
    m: MULTINAME  $\leftarrow$  Eval[QualifiedIdentifier](env, phase);
    return LEXICALREFERENCE(env: env, variableMultiname: m, strict: Strict[FullNewSubexpression]);
  [FullNewSubexpression  $\Rightarrow$  FullNewExpression] do
    return Eval[FullNewExpression](env, phase);
  [FullNewSubexpression0  $\Rightarrow$  FullNewSubexpression1 PropertyOperator] do
    a: OBJECT  $\leftarrow$  readReference(Eval[FullNewSubexpression1](env, phase), phase);
    return Eval[PropertyOperator](env, a, phase);
  [FullNewSubexpression  $\Rightarrow$  SuperExpression PropertyOperator] do
    a: OBJOPTIONALLIMIT  $\leftarrow$  Eval[SuperExpression](env, phase);
    return Eval[PropertyOperator](env, a, phase)
  end proc;

proc Eval[ShortNewExpression  $\Rightarrow$  new ShortNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  f: OBJECT  $\leftarrow$  readReference(Eval[ShortNewSubexpression](env, phase), phase);
  return construct(f, [], phase)
end proc;

proc Eval[ShortNewSubexpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ShortNewSubexpression  $\Rightarrow$  FullNewSubexpression] do
    return Eval[FullNewSubexpression](env, phase);
  [ShortNewSubexpression  $\Rightarrow$  ShortNewExpression] do
    return Eval[ShortNewExpression](env, phase)
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

```

proc call(this: OBJECT, a: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  case a of
    UNDEFINED ∪ NULL ∪ BOOLEAN ∪ GENERALNUMBER ∪ CHAR16 ∪ STRING ∪ NAMESPACE ∪
      COMPOUNDATTRIBUTE ∪ DATE ∪ REGEXP ∪ PACKAGE do
      throw a TypeError exception;
    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);
    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 ∪ CHAR16 ∪ STRING ∪ NAMESPACE ∪
      COMPOUNDATTRIBUTE ∪ METHODCLOSURE ∪ DATE ∪ REGEXP ∪ PACKAGE do
      throw a TypeError exception;
    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;

```

Property Operators

Syntax

PropertyOperator ⇒
 . *QualifiedIdentifier*
 | *Brackets*

Brackets ⇒
 []
 | [*ListExpression*^{allowIn}]
 | [*ExpressionsWithRest*]

Arguments ⇒
 ()
 | *ParenListExpression*
 | (*ExpressionsWithRest*)

ExpressionsWithRest ⇒
RestExpression
 | *ListExpression*^{allowIn} , *RestExpression*

RestExpression ⇒ ... *AssignmentExpression*^{allowIn}

Validation

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

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

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

proc **Eval**[*PropertyOperator*] (*env*: **ENVIRONMENT**, *base*: **OBJOPTIONALLIMIT**, *phase*: **PHASE**): **OBJORREF**

[*PropertyOperator* $\Rightarrow$. *QualifiedIdentifier*] **do**

m: **MULTINAME** $\leftarrow$ **Eval**[*QualifiedIdentifier*](*env*, *phase*);

case *base* **of**

OBJECT **do**

return **DOTREFERENCE**(*base*: *base*, *limit*: *objectType*(*base*), *multiname*: *m*);

LIMITEDINSTANCE **do**

return **DOTREFERENCE**(*base*: *base*.*instance*, *limit*: *base*.*limit*, *multiname*: *m*)

end case;

[*PropertyOperator* $\Rightarrow$ *Brackets*] **do**

args: **OBJECT**[] $\leftarrow$ **Eval**[*Brackets*](*env*, *phase*);

case *base* **of**

OBJECT **do**

return **BRACKETREFERENCE**(*base*: *base*, *limit*: *objectType*(*base*), *args*: *args*);

LIMITEDINSTANCE **do**

return **BRACKETREFERENCE**(*base*: *base*.*instance*, *limit*: *base*.*limit*, *args*: *args*)

end case

end proc;

proc **Eval**[*Brackets*] (*env*: **ENVIRONMENT**, *phase*: **PHASE**): **OBJECT**[]

[*Brackets* $\Rightarrow$ [*1*]] **do** **return** [];

[*Brackets* $\Rightarrow$ [*ListExpression*^{allowin}]] **do**

return **EvalAsList**[*ListExpression*^{allowin}](*env*, *phase*);

[*Brackets* $\Rightarrow$ [*ExpressionsWithRest*]] **do** **return** **Eval**[*ExpressionsWithRest*](*env*, *phase*)

end proc;

proc **Eval**[*Arguments*] (*env*: **ENVIRONMENT**, *phase*: **PHASE**): **OBJECT**[]

[*Arguments* $\Rightarrow$ ()] **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 is(a, Array) then throw a TypeError exception — the ... operand must be an Array
end if;
length: ULONG ← readInstanceSlot(a, arrayPrivate::"length", phase);
i: INTEGER ← 0;
args: OBJECT[] ← [];
while i ≠ length.value do
  arg: OBJECTOPT ← indexRead(a, i, phase);
  if arg = none then
    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 ← args ⊕ [arg];
  i ← i + 1
end while;
return args
end proc;

```

Unary Operators

Syntax

```

UnaryExpression ⇒
  PostfixExpression
| delete PostfixExpression
| void UnaryExpression
| typeof UnaryExpression
| ++ PostfixExpression
| -- PostfixExpression
| + UnaryExpression
| - UnaryExpression
| - NegatedMinLong
| ~ UnaryExpression
| ! UnaryExpression

```

Validation

```

Strict[UnaryExpression]: BOOLEAN;

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

```

Setup

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

Evaluation

```

proc Eval[UnaryExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [UnaryExpression  $\Rightarrow$  PostfixExpression] do return Eval[PostfixExpression](env, phase);
  [UnaryExpression  $\Rightarrow$  delete PostfixExpression] do
    if phase = compile then
      throw a ConstantError exception — delete cannot be used in constant expressions
    end if;
    r: OBJORREF  $\leftarrow$  Eval[PostfixExpression](env, phase);
    return deleteReference(r, Strict[UnaryExpression], phase);
  [UnaryExpression0  $\Rightarrow$  void UnaryExpression1] do
    readReference(Eval[UnaryExpression1](env, phase), phase);
    return undefined;
  [UnaryExpression0  $\Rightarrow$  typeof UnaryExpression1] do
    a: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression1](env, phase), phase);
    c: CLASS  $\leftarrow$  objectType(a);
    return c.typeofString;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

```
[UnaryExpression  $\Rightarrow$  ++ PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — ++ cannot be used in constant expressions
  end if;
  r: OBJORREF  $\leftarrow$  Eval[PostfixExpression](env, phase);
  a: OBJECT  $\leftarrow$  readReference(r, phase);
  b: OBJECT  $\leftarrow$  plus(a, phase);
  c: OBJECT  $\leftarrow$  add(b, 1.0164, phase);
  writeReference(r, c, phase);
  return c;

[UnaryExpression  $\Rightarrow$  -- PostfixExpression] do
  if phase = compile then
    throw a ConstantError exception — -- cannot be used in constant expressions
  end if;
  r: OBJORREF  $\leftarrow$  Eval[PostfixExpression](env, phase);
  a: OBJECT  $\leftarrow$  readReference(r, phase);
  b: OBJECT  $\leftarrow$  plus(a, phase);
  c: OBJECT  $\leftarrow$  subtract(b, 1.0164, phase);
  writeReference(r, c, phase);
  return c;

[UnaryExpression0  $\Rightarrow$  + UnaryExpression1] do
  a: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression1](env, phase), phase);
  return plus(a, phase);

[UnaryExpression0  $\Rightarrow$  - UnaryExpression1] do
  a: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression1](env, phase), phase);
  return minus(a, phase);

[UnaryExpression  $\Rightarrow$  - NegatedMinLong] do return (-263)long;

[UnaryExpression0  $\Rightarrow$  ~ UnaryExpression1] do
  a: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression1](env, phase), phase);
  return bitNot(a, phase);

[UnaryExpression0  $\Rightarrow$  ! UnaryExpression1] do
  a: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression1](env, phase), phase);
  return logicalNot(a, phase)
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

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

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

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return toGeneralNumber(a, phase)

minus(a, phase) returns the value of the unary expression -a. If phase is **compile**, only constant operations are permitted.

```
proc minus(a: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER  $\leftarrow$  objectToGeneralNumber(a, phase);
  return generalNumberNegate(x)
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: x: GENERALNUMBER $\leftarrow$ toGeneralNumber(a, phase);

```
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  $\leftarrow$  objectToGeneralNumber(a, phase);
  case x of
    LONG do i:  $\{-2^{63} \dots 2^{63} - 1\} \leftarrow x.value$ ; return bitwiseXor(i, -1)long;
    ULONG do
      i:  $\{0 \dots 2^{64} - 1\} \leftarrow x.value$ ;
      return bitwiseXor(i, 0xFFFFFFFFFFFFFFFF)ulong;
    FLOAT32  $\cup$  FLOAT64 do
      i:  $\{-2^{31} \dots 2^{31} - 1\} \leftarrow signedWrap32(truncateToInteger(x))$ ;
      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 constant operations are permitted.

```

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

```

Multiplicative Operators

Syntax

```

MultiplicativeExpression  $\Rightarrow$ 
  UnaryExpression
  | MultiplicativeExpression * UnaryExpression
  | MultiplicativeExpression / UnaryExpression
  | MultiplicativeExpression % UnaryExpression

```

Validation

Validate[*MultiplicativeExpression*] (*cxt*: 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

```

proc Eval[MultiplicativeExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [MultiplicativeExpression  $\Rightarrow$  UnaryExpression] do
    return Eval[UnaryExpression](env, phase);
  [MultiplicativeExpression0  $\Rightarrow$  MultiplicativeExpression1 * UnaryExpression] do
    a: OBJECT  $\leftarrow$  readReference(Eval[MultiplicativeExpression1](env, phase), phase);
    b: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression](env, phase), phase);
    return multiply(a, b, phase);
  [MultiplicativeExpression0  $\Rightarrow$  MultiplicativeExpression1 / UnaryExpression] do
    a: OBJECT  $\leftarrow$  readReference(Eval[MultiplicativeExpression1](env, phase), phase);
    b: OBJECT  $\leftarrow$  readReference(Eval[UnaryExpression](env, phase), phase);
    return divide(a, b, phase);

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *x*: GENERALNUMBER $\leftarrow$ *toGeneralNumber*(*a*, *phase*);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: **return not** *toBoolean*(*a*, *phase*)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

```

[MultiplicativeExpression0 ⇒ MultiplicativeExpression1 * UnaryExpression] do
  a: OBJECT ← readReference(Eval[MultiplicativeExpression1](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 ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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 float64Multiply(toFloat64(x), toFloat64(y))
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *x*: GENERALNUMBER ← toGeneralNumber(*a*, *phase*); - ... [24]

```

proc divide(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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;
      if x ∈ ULONG or y ∈ ULONG then return rationalToULong(q)
      else return rationalToLong(q)
    end if
  end if;
  return float64Divide(toFloat64(x), toFloat64(y))
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *x*: GENERALNUMBER ← toGeneralNumber(*a*, *phase*); - ... [25]

```

proc remainder(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(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 ? ⌊q⌋ : ⌈q⌉;
      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;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *x*: GENERALNUMBER ← toGeneralNumber(*a*, *phase*); - ... [26]

Additive Operators

Syntax

AdditiveExpression $\Rightarrow$
MultiplicativeExpression
 | *AdditiveExpression* + *MultiplicativeExpression*
 | *AdditiveExpression* - *MultiplicativeExpression*

Validation

Validate[*AdditiveExpression*] (*cxt*: **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  $\Rightarrow$  MultiplicativeExpression] do
    return Eval[MultiplicativeExpression](env, phase);
  [AdditiveExpression0  $\Rightarrow$  AdditiveExpression1 + MultiplicativeExpression] do
    a: OBJECT  $\leftarrow$  readReference(Eval[AdditiveExpression1](env, phase), phase);
    b: OBJECT  $\leftarrow$  readReference(Eval[MultiplicativeExpression](env, phase), phase);
    return add(a, b, phase);
  [AdditiveExpression0  $\Rightarrow$  AdditiveExpression1 - MultiplicativeExpression] do
    a: OBJECT  $\leftarrow$  readReference(Eval[AdditiveExpression1](env, phase), phase);
    b: OBJECT  $\leftarrow$  readReference(Eval[MultiplicativeExpression](env, phase), phase);
    return subtract(a, b, phase)
end proc;

```

```

proc add(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  ap: PRIMITIVEOBJECT  $\leftarrow$  objectToPrimitive(a, none, phase);
  bp: PRIMITIVEOBJECT  $\leftarrow$  objectToPrimitive(b, none, phase);
  if ap  $\in$  CHAR16  $\cup$  STRING or bp  $\in$  CHAR16  $\cup$  STRING then
    return objectToString(ap, phase)  $\oplus$  objectToString(bp, phase)
  end if;
  x: GENERALNUMBER  $\leftarrow$  objectToGeneralNumber(ap, phase);
  y: GENERALNUMBER  $\leftarrow$  objectToGeneralNumber(bp, phase);
  if x  $\in$  LONG  $\cup$  ULONG or y  $\in$  LONG  $\cup$  ULONG then
    i: INTEGEROPT  $\leftarrow$  checkInteger(x);
    j: INTEGEROPT  $\leftarrow$  checkInteger(y);
    if i  $\neq$  none and j  $\neq$  none then
      k: INTEGER  $\leftarrow$  i + j;
      if x  $\in$  ULONG or y  $\in$  ULONG then return integerToULong(k)
      else return integerToLong(k)
    end if
  end if
  end if;
  return float64Add(toFloat64(x), toFloat64(y))
end proc;

```

```
proc subtract(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(b, phase);
  if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
    i: INTEGEROPT ← checkInteger(x);
    j: INTEGEROPT ← checkInteger(y);
    if i ≠ none and j ≠ none then
      k: INTEGER ← i − j;
      if x ∈ ULONG or y ∈ ULONG then return integerToULong(k)
      else return integerToLong(k)
    end if
  end if
end if;
return float64Subtract(toFloat64(x), toFloat64(y))
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *x*: GENERALNUMBER ← toGeneralNumber(*a*,
phase); - ... [29]

Bitwise Shift Operators

Syntax

ShiftExpression ⇒
 AdditiveExpression
| *ShiftExpression* << *AdditiveExpression*
| *ShiftExpression* >> *AdditiveExpression*
| *ShiftExpression* >>> *AdditiveExpression*

Validation

Validate[*ShiftExpression*] (*ctx*: 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 ← objectToGeneralNumber(a, phase);
  count: INTEGER ← truncateToInteger(objectToGeneralNumber(b, phase));
  case x of
    FLOAT32 ∪ FLOAT64 do
      i: { $-2^{31} \dots 2^{31} - 1$ } ← signedWrap32(truncateToInteger(x));
      count ← bitwiseAnd(count, 0x1F);
      i ← signedWrap32(bitwiseShift(i, count));
      return realToFloat64(i);
    LONG do
      count ← bitwiseAnd(count, 0x3F);
      i: { $-2^{63} \dots 2^{63} - 1$ } ← signedWrap64(bitwiseShift(x.value, count));
      return  $i_{\text{long}}$ ;
    ULONG do
      count ← bitwiseAnd(count, 0x3F);
      i: { $0 \dots 2^{64} - 1$ } ← unsignedWrap64(bitwiseShift(x.value, count));
      return  $i_{\text{ulong}}$ ;
  end case
end proc;

proc shiftRight(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  count: INTEGER ← truncateToInteger(objectToGeneralNumber(b, phase));
  case x of
    FLOAT32 ∪ FLOAT64 do
      i: { $-2^{31} \dots 2^{31} - 1$ } ← signedWrap32(truncateToInteger(x));
      count ← bitwiseAnd(count, 0x1F);
      i ← bitwiseShift(i, -count);
      return realToFloat64(i);
    LONG do
      count ← bitwiseAnd(count, 0x3F);
      i: { $-2^{63} \dots 2^{63} - 1$ } ← bitwiseShift(x.value, -count);
      return  $i_{\text{long}}$ ;
    ULONG do
      count ← bitwiseAnd(count, 0x3F);
      i: { $-2^{63} \dots 2^{63} - 1$ } ← bitwiseShift(signedWrap64(x.value), -count);
      return (unsignedWrap64(i))ulong;
  end case
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: x: GENERALNUMBER ← toGeneralNumber(a, phase); - ... [30]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: x: GENERALNUMBER ← toGeneralNumber(a, phase); - ... [31]

```

proc shiftRightUnsigned(a: OBJECT, b: OBJECT, phase: PHASE): OBJECT
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  count: INTEGER ← truncateToInteger(objectToGeneralNumber(b, phase));
  case x of
    FLOAT32 ∪ FLOAT64 do
      i: {0 ... 232 - 1} ← unsignedWrap32(truncateToInteger(x));
      count ← bitwiseAnd(count, 0x1F);
      i ← bitwiseShift(i, -count);
      return realToFloat64(i);
    LONG do
      count ← bitwiseAnd(count, 0x3F);
      i: {0 ... 264 - 1} ← bitwiseShift(unsignedWrap64(x.value), -count);
      return (signedWrap64(i))long;
    ULONG do
      count ← bitwiseAnd(count, 0x3F);
      i: {0 ... 264 - 1} ← bitwiseShift(x.value, -count);
      return iulong
  end case
end proc;

```

Relational Operators

Syntax

RelationalExpression^{allowin} ⇒

- ShiftExpression*
- | *RelationalExpression*^{allowin} < *ShiftExpression*
- | *RelationalExpression*^{allowin} > *ShiftExpression*
- | *RelationalExpression*^{allowin} <= *ShiftExpression*
- | *RelationalExpression*^{allowin} >= *ShiftExpression*
- | *RelationalExpression*^{allowin} **is** *ShiftExpression*
- | *RelationalExpression*^{allowin} **as** *ShiftExpression*
- | *RelationalExpression*^{allowin} **in** *ShiftExpression*
- | *RelationalExpression*^{allowin} **instanceof** *ShiftExpression*

RelationalExpression^{noin} ⇒

- ShiftExpression*
- | *RelationalExpression*^{noin} < *ShiftExpression*
- | *RelationalExpression*^{noin} > *ShiftExpression*
- | *RelationalExpression*^{noin} <= *ShiftExpression*
- | *RelationalExpression*^{noin} >= *ShiftExpression*
- | *RelationalExpression*^{noin} **is** *ShiftExpression*
- | *RelationalExpression*^{noin} **as** *ShiftExpression*
- | *RelationalExpression*^{noin} **instanceof** *ShiftExpression*

Validation

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

Setup

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

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *x*: GENERALNUMBER ← *toGeneralNumber*(*a*, *phase*); - ... [32]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Evaluation

```

proc Eval[RelationalExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [RelationalExpressionn ⇒ ShiftExpression] do
    return Eval[ShiftExpression](env, phase);
  [RelationalExpression0 ⇒ RelationalExpression1 < ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    return isLess(a, b, phase);
  [RelationalExpression0 ⇒ RelationalExpression1 > ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    return isLess(b, a, phase);
  [RelationalExpression0 ⇒ RelationalExpression1 <= ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    return isLessOrEqual(a, b, phase);
  [RelationalExpression0 ⇒ RelationalExpression1 >= ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    return isLessOrEqual(b, a, phase);
  [RelationalExpression0 ⇒ RelationalExpression1 is ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    c: CLASS ← objectToClass(b);
    return is(a, c);
  [RelationalExpression0 ⇒ RelationalExpression1 as ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    c: CLASS ← objectToClass(b);
    return as(a, c, true);
  [RelationalExpressionallowin0 ⇒ RelationalExpressionallowin1 in ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpressionallowin1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    qname: QUALIFIEDNAME ← objectToQualifiedname(a, phase);
    return hasProperty(b, qname, false);
  [RelationalExpression0 ⇒ RelationalExpression1 instanceof ShiftExpression] do
    a: OBJECT ← readReference(Eval[RelationalExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[ShiftExpression](env, phase), phase);
    if b ∈ CLASS then return is(a, b)
    elseif is(b, PrototypeFunction) then
      prototype: OBJECT ← dotRead(b, {public::"prototype"}, phase);
      return prototype ∈ archetypes(a)
    else throw a TypeError exception
    end if
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: c: CLASS ← toClass(b);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: c: CLASS ← toClass(b); - ... [33]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: qname: QUALIFIEDNAME ← toQualifiedname(a, phase); - ... [34]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return prototype ∈ objectSupers(a)

```

proc isLess(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  ap: PRIMITIVEOBJECT ← objectToPrimitive(a, hintNumber, phase);
  bp: PRIMITIVEOBJECT ← objectToPrimitive(b, hintNumber, phase);
  if ap ∈ CHAR16 ∪ STRING and bp ∈ CHAR16 ∪ STRING then
    return toString(ap) < toString(bp)
  end if;
  return generalNumberCompare(objectToGeneralNumber(ap, phase), objectToGeneralNumber(bp, phase)) = less
end proc;

proc isLessOrEqual(a: OBJECT, b: OBJECT, phase: PHASE): BOOLEAN
  ap: PRIMITIVEOBJECT ← objectToPrimitive(a, hintNumber, phase);
  bp: PRIMITIVEOBJECT ← objectToPrimitive(b, hintNumber, phase);
  if ap ∈ CHAR16 ∪ STRING and bp ∈ CHAR16 ∪ STRING then
    return toString(ap) ≤ toString(bp)
  end if;
  return generalNumberCompare(objectToGeneralNumber(ap, phase),
    objectToGeneralNumber(bp, phase)) ∈ {less, equal}
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: ap: PRIMITIVEOBJECT ← toPrimitive(a, null, phase); - ... [35]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: return toString(ap, phase) < toString(bp, phase)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: return generalNumberCompare(toGeneralNumber(ap, phase), toGeneralNumber(bp, phase)) =

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: ap: PRIMITIVEOBJECT ← toPrimitive(a, null, phase); - ... [36]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: return toString(ap, phase) ≤ toString(bp, phase) - ... [37]

Equality Operators

Syntax

*EqualityExpression*ⁿ ⇒
*RelationalExpression*ⁿ
 | *EqualityExpression*ⁿ == *RelationalExpression*ⁿ
 | *EqualityExpression*ⁿ != *RelationalExpression*ⁿ
 | *EqualityExpression*ⁿ === *RelationalExpression*ⁿ
 | *EqualityExpression*ⁿ !== *RelationalExpression*ⁿ

Validation

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

Setup

Setup[*EqualityExpression*ⁿ] () propagates the call to **Setup** to every nonterminal in the expansion of *EqualityExpression*ⁿ.

Evaluation

```

proc Eval[EqualityExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [EqualityExpressionn ⇒ RelationalExpressionn] do
    return Eval[RelationalExpressionn](env, phase);
  [EqualityExpressionn0 ⇒ EqualityExpressionn1 == RelationalExpressionn] do
    a: OBJECT ← readReference(Eval[EqualityExpressionn1](env, phase), phase);
    b: OBJECT ← readReference(Eval[RelationalExpressionn](env, phase), phase);
    return isEqual(a, b, phase);
  [EqualityExpressionn0 ⇒ EqualityExpressionn1 != RelationalExpressionn] do
    a: OBJECT ← readReference(Eval[EqualityExpressionn1](env, phase), phase);
    b: OBJECT ← readReference(Eval[RelationalExpressionn](env, phase), phase);
    return not isEqual(a, b, phase);

```



```

[EqualityExpression0 ⇒ EqualityExpression1 == RelationalExpression] do
  a: OBJECT ← readReference(Eval[EqualityExpression1](env, phase), phase);
  b: OBJECT ← readReference(Eval[RelationalExpression](env, phase), phase);
  return isStrictlyEqual(a, b, phase);
[EqualityExpression0 ⇒ EqualityExpression1 != RelationalExpression] do
  a: OBJECT ← readReference(Eval[EqualityExpression1](env, phase), phase);
  b: OBJECT ← readReference(Eval[RelationalExpression](env, phase), phase);
  return not isStrictlyEqual(a, b, phase)
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(objectToGeneralNumber(a, phase), b, phase)
      end if;
    GENERALNUMBER do
      bp: PRIMITIVEOBJECT ← objectToPrimitive(b, none, phase);
      case bp of
        UNDEFINED ∪ NULL do return false;
        BOOLEAN ∪ GENERALNUMBER ∪ CHAR16 ∪ STRING do
          return generalNumberCompare(a, objectToGeneralNumber(bp, phase)) = equal
        end case;
      CHAR16 ∪ STRING do
        bp: PRIMITIVEOBJECT ← objectToPrimitive(b, none, phase);
        case bp of
          UNDEFINED ∪ NULL do return false;
          BOOLEAN ∪ GENERALNUMBER do
            return generalNumberCompare(objectToGeneralNumber(a, phase),
              objectToGeneralNumber(bp, phase)) = equal;
          CHAR16 ∪ STRING do return toString(a) = toString(bp)
          end case;
        end case;
      NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ SIMPLEINSTANCE ∪ DATE ∪ REGEXP ∪
      PACKAGE do
        case b of
          UNDEFINED ∪ NULL do return false;
          NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ SIMPLEINSTANCE ∪ DATE ∪
          REGEXP ∪ PACKAGE do
            return isStrictlyEqual(a, b, phase);
          BOOLEAN ∪ GENERALNUMBER ∪ CHAR16 ∪ STRING do
            ap: PRIMITIVEOBJECT ← objectToPrimitive(a, none, phase);
            return isEqual(ap, b, phase)
          end case
        end case
      end case
  end case;

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;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: else return isEqual(toGeneralNumber(a, phase), b, phase)

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: bp: PRIMITIVEOBJECT ← toPrimitive(e(b, null, phase);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return generalNumberCompare(a, toGeneralNumber(bp, phase)) = equal

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: bp: PRIMITIVEOBJECT ← toPrimitive(e(b, null, phase);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return generalNumberCompare(toGeneralNumber(a, phase), toGeneralNumber(bp, phase)) = equal ... [38]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: ap: PRIMITIVEOBJECT ← toPrimitive(e(a, null, phase);

Binary Bitwise Operators

Syntax

$\text{BitwiseAndExpression}^n \Rightarrow$
 $\text{EqualityExpression}^n$
 $| \text{BitwiseAndExpression}^n \ \& \ \text{EqualityExpression}^n$

$\text{BitwiseXorExpression}^n \Rightarrow$
 $\text{BitwiseAndExpression}^n$
 $| \text{BitwiseXorExpression}^n \ \wedge \ \text{BitwiseAndExpression}^n$

$\text{BitwiseOrExpression}^n \Rightarrow$
 $\text{BitwiseXorExpression}^n$
 $| \text{BitwiseOrExpression}^n \ | \ \text{BitwiseXorExpression}^n$

Validation

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

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

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

Setup

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

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

$\text{Setup}[\text{BitwiseOrExpression}^n]$ () propagates the call to **Setup** to every nonterminal in the expansion of $\text{BitwiseOrExpression}^n$.

Evaluation

```

proc Eval[BitwiseAndExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [BitwiseAndExpressionn ⇒ EqualityExpressionn] do
    return Eval[EqualityExpressionn](env, phase);
  [BitwiseAndExpressionn0 ⇒ BitwiseAndExpressionn1 & EqualityExpressionn] do
    a: OBJECT ← readReference(Eval[BitwiseAndExpressionn1](env, phase), phase);
    b: OBJECT ← readReference(Eval[EqualityExpressionn](env, phase), phase);
    return bitAnd(a, b, phase)
end proc;

proc Eval[BitwiseXorExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [BitwiseXorExpressionn ⇒ BitwiseAndExpressionn] do
    return Eval[BitwiseAndExpressionn](env, phase);
  
```

```

[BitwiseXorExpression0 ⇒ BitwiseXorExpression1 ∧ BitwiseAndExpression1] do
  a: OBJECT ← readReference(Eval[BitwiseXorExpression1](env, phase), phase);
  b: OBJECT ← readReference(Eval[BitwiseAndExpression1](env, phase), phase);
  return bitXor(a, b, phase)
end proc;

```

```

proc Eval[BitwiseOrExpression1] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[BitwiseOrExpression0 ⇒ BitwiseXorExpression1] do
  return Eval[BitwiseXorExpression1](env, phase);
[BitwiseOrExpression0 ⇒ BitwiseOrExpression1 | BitwiseXorExpression1] do
  a: OBJECT ← readReference(Eval[BitwiseOrExpression1](env, phase), phase);
  b: OBJECT ← readReference(Eval[BitwiseXorExpression1](env, phase), phase);
  return bitOr(a, b, phase)
end proc;

```

```

proc bitAnd(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
y: GENERALNUMBER ← objectToGeneralNumber(b, phase);
if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
  i: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(x));
  j: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(y));
  k: {−263 ... 263 − 1} ← bitwiseAnd(i, j);
  if x ∈ ULONG or y ∈ ULONG then return (unsignedWrap64(k))ulong
  else return klong
end if
else
  i: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(x));
  j: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(y));
  return realToFloat64(bitwiseAnd(i, j))
end if
end proc;

```

```

proc bitXor(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
y: GENERALNUMBER ← objectToGeneralNumber(b, phase);
if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
  i: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(x));
  j: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(y));
  k: {−263 ... 263 − 1} ← bitwiseXor(i, j);
  if x ∈ ULONG or y ∈ ULONG then return (unsignedWrap64(k))ulong
  else return klong
end if
else
  i: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(x));
  j: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(y));
  return realToFloat64(bitwiseXor(i, j))
end if
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: x: GENERALNUMBER ← toGeneralN
umber(a, phase); - ... [39]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: x: GENERALNUMBER ← toGeneralN
umber(a, phase); - ... [40]

```
proc bitOr(a: OBJECT, b: OBJECT, phase: PHASE): GENERALNUMBER
  x: GENERALNUMBER ← objectToGeneralNumber(a, phase);
  y: GENERALNUMBER ← objectToGeneralNumber(b, phase);
  if x ∈ LONG ∪ ULONG or y ∈ LONG ∪ ULONG then
    i: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(x));
    j: {−263 ... 263 − 1} ← signedWrap64(truncateToInteger(y));
    k: {−263 ... 263 − 1} ← bitwiseOr(i, j);
    if x ∈ ULONG or y ∈ ULONG then return (unsignedWrap64(k))ulong
    else return klong
  end if
else
  i: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(x));
  j: {−231 ... 231 − 1} ← signedWrap32(truncateToInteger(y));
  return realToFloat64(bitwiseOr(i, j))
end if
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *x*: GENERALNUMBER ← toGeneralNumber(*a*,
phase); - ... [41]

Binary Logical Operators

Syntax

```
LogicalAndExpressionr ⇒  
  BitwiseOrExpressionr  
  | LogicalAndExpressionr && BitwiseOrExpressionr  
  
LogicalXorExpressionr ⇒  
  LogicalAndExpressionr  
  | LogicalXorExpressionr ^^ LogicalAndExpressionr  
  
LogicalOrExpressionr ⇒  
  LogicalXorExpressionr  
  | LogicalOrExpressionr || LogicalXorExpressionr
```

Validation

- Validate[LogicalAndExpression^r] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalAndExpression^r.
- Validate[LogicalXorExpression^r] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalXorExpression^r.
- Validate[LogicalOrExpression^r] (*cxt*: CONTEXT, *env*: ENVIRONMENT) propagates the call to Validate to every nonterminal in the expansion of LogicalOrExpression^r.

Setup

- Setup[LogicalAndExpression^r] () propagates the call to Setup to every nonterminal in the expansion of LogicalAndExpression^r.
- Setup[LogicalXorExpression^r] () propagates the call to Setup to every nonterminal in the expansion of LogicalXorExpression^r.
- Setup[LogicalOrExpression^r] () propagates the call to Setup to every nonterminal in the expansion of LogicalOrExpression^r.

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Evaluation

```

proc Eval[LogicalAndExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [LogicalAndExpressionn ⇒ BitwiseOrExpressionn] do
    return Eval[BitwiseOrExpressionn](env, phase);
  [LogicalAndExpression0 ⇒ LogicalAndExpression1 && BitwiseOrExpression1] do
    a: OBJECT ← readReference(Eval[LogicalAndExpression1](env, phase), phase);
    if objectToBoolean(a) then
      return readReference(Eval[BitwiseOrExpression1](env, phase), phase)
    else return a
    end if
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if toBoolean(a, phase) then

```

proc Eval[LogicalXorExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [LogicalXorExpressionn ⇒ LogicalAndExpressionn] do
    return Eval[LogicalAndExpressionn](env, phase);
  [LogicalXorExpression0 ⇒ LogicalXorExpression1 ^^ LogicalAndExpression1] do
    a: OBJECT ← readReference(Eval[LogicalXorExpression1](env, phase), phase);
    b: OBJECT ← readReference(Eval[LogicalAndExpression1](env, phase), phase);
    ba: BOOLEAN ← objectToBoolean(a);
    bb: BOOLEAN ← objectToBoolean(b);
    return ba xor bb
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: ba: BOOLEAN ← toBoolean(a, phase);
... [42]

```

proc Eval[LogicalOrExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [LogicalOrExpressionn ⇒ LogicalXorExpressionn] do
    return Eval[LogicalXorExpressionn](env, phase);
  [LogicalOrExpression0 ⇒ LogicalOrExpression1 || LogicalXorExpression1] do
    a: OBJECT ← readReference(Eval[LogicalOrExpression1](env, phase), phase);
    if objectToBoolean(a) then return a
    else return readReference(Eval[LogicalXorExpression1](env, phase), phase)
    end if
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if toBoolean(a, phase) then return a

Conditional Operator

Syntax

```

ConditionalExpressionn ⇒
  LogicalOrExpressionn
| LogicalOrExpressionn ? AssignmentExpressionn : AssignmentExpressionn

NonAssignmentExpressionn ⇒
  LogicalOrExpressionn
| LogicalOrExpressionn ? NonAssignmentExpressionn : NonAssignmentExpressionn

```

Validation

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

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

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

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

```
proc Eval[ConditionalExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ConditionalExpression  $\Rightarrow$  LogicalOrExpression] do
    return Eval[LogicalOrExpression](env, phase);
  [ConditionalExpression  $\Rightarrow$  LogicalOrExpression ? AssignmentExpression1 : AssignmentExpression2] do
    a: OBJECT  $\leftarrow$  readReference(Eval[LogicalOrExpression](env, phase), phase);
    if objectToBoolean(a) then
      return readReference(Eval[AssignmentExpression1](env, phase), phase)
    else return readReference(Eval[AssignmentExpression2](env, phase), phase)
    end if
  end proc;

proc Eval[NonAssignmentExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [NonAssignmentExpression  $\Rightarrow$  LogicalOrExpression] do
    return Eval[LogicalOrExpression](env, phase);
  [NonAssignmentExpression0  $\Rightarrow$  LogicalOrExpression ? NonAssignmentExpression1 : NonAssignmentExpression2]
  do
    a: OBJECT  $\leftarrow$  readReference(Eval[LogicalOrExpression](env, phase), phase);
    if objectToBoolean(a) then
      return readReference(Eval[NonAssignmentExpression1](env, phase), phase)
    else return readReference(Eval[NonAssignmentExpression2](env, phase), phase)
    end if
  end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if toBoolean(a, phase) then

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if toBoolean(a, phase) then

Assignment Operators

Syntax

AssignmentExpression $\Rightarrow$
ConditionalExpression
| *PostfixExpression* = *AssignmentExpression*
| *PostfixExpression* CompoundAssignment *AssignmentExpression*
| *PostfixExpression* LogicalAssignment *AssignmentExpression*

CompoundAssignment $\Rightarrow$

```

*=
| /=
| %=
| +=
| -=
| <<=
| >>=
| >>>=
| &=
| ^=
| |=

```

LogicalAssignment $\Rightarrow$

```

&&=
| ^=
| |=

```

Semantics**tag andEq;****tag xorEq;****tag orEq;****Validation**

```

proc Validate[AssignmentExpressionn] (cxt: CONTEXT, env: ENVIRONMENT)
  [AssignmentExpressionn  $\Rightarrow$  ConditionalExpressionn] do
    Validate[ConditionalExpressionn](cxt, env);
  [AssignmentExpression0  $\Rightarrow$  PostfixExpression = AssignmentExpression1] do
    Validate[PostfixExpression](cxt, env);
    Validate[AssignmentExpression1](cxt, env);
  [AssignmentExpression0  $\Rightarrow$  PostfixExpression CompoundAssignment AssignmentExpression1] do
    Validate[PostfixExpression](cxt, env);
    Validate[AssignmentExpression1](cxt, env);
  [AssignmentExpression0  $\Rightarrow$  PostfixExpression LogicalAssignment AssignmentExpression1] do
    Validate[PostfixExpression](cxt, env);
    Validate[AssignmentExpression1](cxt, env)
end proc;

```

Setup

```

proc Setup[AssignmentExpressionn] ()
  [AssignmentExpressionn  $\Rightarrow$  ConditionalExpressionn] do Setup[ConditionalExpressionn]();
  [AssignmentExpression0  $\Rightarrow$  PostfixExpression = AssignmentExpression1] do
    Setup[PostfixExpression]();
    Setup[AssignmentExpression1]();
  [AssignmentExpression0  $\Rightarrow$  PostfixExpression CompoundAssignment AssignmentExpression1] do
    Setup[PostfixExpression]();
    Setup[AssignmentExpression1]();

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

```
[AssignmentExpression0 ⇒ PostfixExpression LogicalAssignment AssignmentExpression1] do
  Setup[PostfixExpression]();
  Setup[AssignmentExpression1]()
end proc;
```

Evaluation

```
proc Eval[AssignmentExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
[AssignmentExpression ⇒ ConditionalExpression] do
  return Eval[ConditionalExpression](env, phase);
[AssignmentExpression0 ⇒ PostfixExpression = AssignmentExpression1] do
  if phase = compile then
    throw a ConstantError exception — assignment cannot be used in constant expressions
  end if;
  ra: OBJORREF ← Eval[PostfixExpression](env, phase);
  b: OBJECT ← readReference(Eval[AssignmentExpression1](env, phase), phase);
  writeReference(ra, b, phase);
  return b;
[AssignmentExpression0 ⇒ PostfixExpression CompoundAssignment AssignmentExpression1] do
  if phase = compile then
    throw a ConstantError exception — assignment cannot be used in constant expressions
  end if;
  rLeft: OBJORREF ← Eval[PostfixExpression](env, phase);
  oLeft: OBJECT ← readReference(rLeft, phase);
  oRight: OBJECT ← readReference(Eval[AssignmentExpression1](env, phase), phase);
  result: OBJECT ← Op[CompoundAssignment](oLeft, oRight, phase);
  writeReference(rLeft, result, phase);
  return result;
[AssignmentExpression0 ⇒ PostfixExpression LogicalAssignment AssignmentExpression1] do
  if phase = compile then
    throw a ConstantError exception — assignment cannot be used in constant expressions
  end if;
  rLeft: OBJORREF ← Eval[PostfixExpression](env, phase);
  oLeft: OBJECT ← readReference(rLeft, phase);
  bLeft: BOOLEAN ← objectToBoolean(oLeft);
  result: OBJECT ← oLeft;
  case Operator[LogicalAssignment] of
    {andEq} do
      if bLeft then
        result ← readReference(Eval[AssignmentExpression1](env, phase), phase)
      end if;
    {xorEq} do
      bRight: BOOLEAN ← objectToBoolean(readReference(Eval[AssignmentExpression1](env, phase), phase));
      result ← bLeft xor bRight;
    {orEq} do
      if not bLeft then
        result ← readReference(Eval[AssignmentExpression1](env, phase), phase)
      end if;
    end case;
  writeReference(rLeft, result, phase);
  return result
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: a

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: bLeft: BOOLEAN ← toBoolean(oLeft, phase);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: toBoolean(readReference(Eval[AssignmentExpression₁](env, phase), phase), phase);


```

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;

```

Comma Expressions

Syntax

```

ListExpressionn ⇒
  AssignmentExpressionn
| ListExpressionn , AssignmentExpressionn

```

Validation

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

Setup

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

Evaluation

```

proc Eval[ListExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [ListExpressionn ⇒ AssignmentExpressionn] do
    return Eval[AssignmentExpressionn](env, phase);
  [ListExpressionn ⇒ ListExpressionn1 , AssignmentExpressionn] do
    readReference(Eval[ListExpressionn1](env, phase), phase);
    return readReference(Eval[AssignmentExpressionn](env, phase), phase)
end proc;

proc EvalAsList[ListExpressionn] (env: ENVIRONMENT, phase: PHASE): OBJECT[]
  [ListExpressionn ⇒ AssignmentExpressionn] do
    elt: OBJECT ← readReference(Eval[AssignmentExpressionn](env, phase), phase);
    return [elt];

```

```
[ListExpression0 ⇒ ListExpression1 , AssignmentExpression] do
  elts: OBJECT[] ← EvalAsList[ListExpression1](env, phase);
  elt: OBJECT ← readReference(Eval[AssignmentExpression](env, phase), phase);
  return elts ⊕ [elt]
end proc;
```

Type Expressions

Syntax

TypeExpression^{*} ⇒ NonAssignmentExpression^{*}

Validation

```
proc Validate[TypeExpression* ⇒ NonAssignmentExpression*] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[NonAssignmentExpression*](cxt, env)
end proc;
```

Setup and Evaluation

```
proc SetupAndEval[TypeExpression* ⇒ NonAssignmentExpression*] (env: ENVIRONMENT): CLASS
  Setup[NonAssignmentExpression*]();
  o: OBJECT ← readReference(Eval[NonAssignmentExpression*](env, compile), compile);
  return objectToClass(o)
end proc;
```

Statements

Syntax

ω ∈ {abbrev, noShortIf, full}

Statement⁻ ⇒

- ExpressionStatement Semicolon-
- SuperStatement Semicolon-
- Block
- LabeledStatement-
- IfStatement-
- SwitchStatement
- DoStatement Semicolon-
- WhileStatement-
- ForStatement-
- WithStatement-
- ContinueStatement Semicolon-
- BreakStatement Semicolon-
- ReturnStatement Semicolon-
- ThrowStatement Semicolon-
- TryStatement

Substatement $\Rightarrow$
 | *EmptyStatement*
 | *Statement*
 | *SimpleVariableDefinition Semicolon*
 | *Attributes* [no line break] { *Substatements* }

Substatements $\Rightarrow$
 | *«empty»*
 | *SubstatementsPrefix Substatement*^{abbrev}

SubstatementsPrefix $\Rightarrow$
 | *«empty»*
 | *SubstatementsPrefix Substatement*^{full}

Semicolon^{abbrev} $\Rightarrow$
 | *;*
 | *VirtualSemicolon*
 | *«empty»*

Semicolon^{noShortIf} $\Rightarrow$
 | *;*
 | *VirtualSemicolon*
 | *«empty»*

Semicolon^{full} $\Rightarrow$
 | *;*
 | *VirtualSemicolon*

Validation

```

proc Validate[Statement-] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL {}, jt: JUMPTARGETS, preinst: BOOLEAN)
  [Statement-  $\Rightarrow$  ExpressionStatement Semicolon-] do
    Validate[ExpressionStatement](cxt, env);
  [Statement-  $\Rightarrow$  SuperStatement Semicolon-] do Validate[SuperStatement](cxt, env);
  [Statement-  $\Rightarrow$  Block] do Validate[Block](cxt, env, jt, preinst);
  [Statement-  $\Rightarrow$  LabeledStatement-] do Validate[LabeledStatement-](cxt, env, sl, jt);
  [Statement-  $\Rightarrow$  IfStatement-] do Validate[IfStatement-](cxt, env, jt);
  [Statement-  $\Rightarrow$  SwitchStatement] do Validate[SwitchStatement](cxt, env, jt);
  [Statement-  $\Rightarrow$  DoStatement Semicolon-] do Validate[DoStatement](cxt, env, sl, jt);
  [Statement-  $\Rightarrow$  WhileStatement-] do Validate[WhileStatement-](cxt, env, sl, jt);
  [Statement-  $\Rightarrow$  ForStatement-] do Validate[ForStatement-](cxt, env, sl, jt);
  [Statement-  $\Rightarrow$  WithStatement-] do Validate[WithStatement-](cxt, env, jt);
  [Statement-  $\Rightarrow$  ContinueStatement Semicolon-] do Validate[ContinueStatement](jt);
  [Statement-  $\Rightarrow$  BreakStatement Semicolon-] do Validate[BreakStatement](jt);
  [Statement-  $\Rightarrow$  ReturnStatement Semicolon-] do Validate[ReturnStatement](cxt, env);
  [Statement-  $\Rightarrow$  ThrowStatement Semicolon-] do Validate[ThrowStatement](cxt, env);
  [Statement-  $\Rightarrow$  TryStatement] do Validate[TryStatement](cxt, env, jt)
end proc;

```

Enabled[*Substatement*-]: BOOLEAN;

```

proc Validate[Substatement-] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL {}, jt: JUMPTARGETS)
  [Substatement-  $\Rightarrow$  EmptyStatement] do nothing;

```

```

[Substatement- ⇒ Statement-] do Validate[Statement-](cxt, env, sl, jt, false);
[Substatement- ⇒ SimpleVariableDefinition Semicolon-] do
  Validate[SimpleVariableDefinition](cxt, env);
[Substatement- ⇒ Attributes [no line break] { Substatements } ] do
  Validate[Attributes](cxt, env);
  Setup[Attributes]();
  attr: ATTRIBUTE ← Eval[Attributes](env, compile);
  if attr ∉ BOOLEAN then
    throw a TypeError 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](cxt, env, jt) end if
end proc;

proc Validate[Substatements] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [Substatements ⇒ «empty»] do nothing;
  [Substatements ⇒ SubstatementsPrefix Substatementabbrev] do
    Validate[SubstatementsPrefix](cxt, env, jt);
    Validate[Substatementabbrev](cxt, env, {}, jt)
  end proc;

proc Validate[SubstatementsPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [SubstatementsPrefix ⇒ «empty»] do nothing;
  [SubstatementsPrefix0 ⇒ SubstatementsPrefix1 Substatementfull] do
    Validate[SubstatementsPrefix1](cxt, env, jt);
    Validate[Substatementfull](cxt, env, {}, jt)
  end proc;

```

Setup

Setup[Statement-] () propagates the call to **Setup** to every nonterminal in the expansion of *Statement-*.

```

proc Setup[Substatement-] ()
  [Substatement- ⇒ EmptyStatement] do nothing;
  [Substatement- ⇒ Statement-] do Setup[Statement-]();
  [Substatement- ⇒ SimpleVariableDefinition Semicolon-] do
    Setup[SimpleVariableDefinition]();
  [Substatement- ⇒ Attributes [no line break] { Substatements } ] do
    if Enabled[Substatement-] 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-] ()
  [Semicolon- ⇒ ; ] do nothing;
  [Semicolon- ⇒ VirtualSemicolon] do nothing;
  [Semicolonabbrev ⇒ «empty»] do nothing;
  [SemicolonnoShortIf ⇒ «empty»] do nothing
end proc;

```

Evaluation

```

proc Eval[Statement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Statement  $\Rightarrow$  ExpressionStatement Semicolon-] do
    return Eval[ExpressionStatement](env);
  [Statement  $\Rightarrow$  SuperStatement Semicolon-] do return Eval[SuperStatement](env);
  [Statement  $\Rightarrow$  Block] do return Eval[Block](env, d);
  [Statement  $\Rightarrow$  LabeledStatement-] do return Eval[LabeledStatement-](env, d);
  [Statement  $\Rightarrow$  IfStatement-] do return Eval[IfStatement-](env, d);
  [Statement  $\Rightarrow$  SwitchStatement] do return Eval[SwitchStatement](env, d);
  [Statement  $\Rightarrow$  DoStatement Semicolon-] do return Eval[DoStatement](env, d);
  [Statement  $\Rightarrow$  WhileStatement-] do return Eval[WhileStatement-](env, d);
  [Statement  $\Rightarrow$  ForStatement-] do return Eval[ForStatement-](env, d);
  [Statement  $\Rightarrow$  WithStatement-] do return Eval[WithStatement-](env, d);
  [Statement  $\Rightarrow$  ContinueStatement Semicolon-] do
    return Eval[ContinueStatement](env, d);
  [Statement  $\Rightarrow$  BreakStatement Semicolon-] do return Eval[BreakStatement](env, d);
  [Statement  $\Rightarrow$  ReturnStatement Semicolon-] do return Eval[ReturnStatement](env);
  [Statement  $\Rightarrow$  ThrowStatement Semicolon-] do return Eval[ThrowStatement](env);
  [Statement  $\Rightarrow$  TryStatement] do return Eval[TryStatement](env, d)
end proc;

```

```

proc Eval[Substatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Substatement  $\Rightarrow$  EmptyStatement] do return d;
  [Substatement  $\Rightarrow$  Statement-] do return Eval[Statement-](env, d);
  [Substatement  $\Rightarrow$  SimpleVariableDefinition Semicolon-] do
    return Eval[SimpleVariableDefinition](env, d);
  [Substatement  $\Rightarrow$  Attributes [no line break] { Substatements } ] do
    if Enabled[Substatement-] then return Eval[Substatements](env, d)
    else return d
    end if
end proc;

```

```

proc Eval[Substatements] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Substatements  $\Rightarrow$  «empty»] do return d;
  [Substatements  $\Rightarrow$  SubstatementsPrefix Substatementabbrev] do
    o: OBJECT  $\leftarrow$  Eval[SubstatementsPrefix](env, d);
    return Eval[Substatementabbrev](env, o)
end proc;

```

```

proc Eval[SubstatementsPrefix] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [SubstatementsPrefix  $\Rightarrow$  «empty»] do return d;
  [SubstatementsPrefix0  $\Rightarrow$  SubstatementsPrefix1 Substatementfull] do
    o: OBJECT  $\leftarrow$  Eval[SubstatementsPrefix1](env, d);
    return Eval[Substatementfull](env, o)
end proc;

```

Empty Statement

Syntax

EmptyStatement $\Rightarrow$;

Expression Statement

Syntax

ExpressionStatement $\Rightarrow$ [lookahead $\notin$ {**function**, {}}] *ListExpression*^{allowIn}

Validation

```
proc Validate[ExpressionStatement  $\Rightarrow$  [lookahead $\notin$ {function, {}}] ListExpressionallowIn]
  (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpressionallowIn](cxt, env)
end proc;
```

Setup

```
proc Setup[ExpressionStatement  $\Rightarrow$  [lookahead $\notin$ {function, {}}] ListExpressionallowIn] ()
  Setup[ListExpressionallowIn]()
end proc;
```

Evaluation

```
proc Eval[ExpressionStatement  $\Rightarrow$  [lookahead $\notin$ {function, {}}] ListExpressionallowIn] (env: ENVIRONMENT): OBJECT
  return readReference(Eval[ListExpressionallowIn](env, run), run)
end proc;
```

Super Statement

Syntax

SuperStatement $\Rightarrow$ **super** *Arguments*

Validation

```
proc Validate[SuperStatement  $\Rightarrow$  super Arguments] (cxt: CONTEXT, env: ENVIRONMENT)
  frame: PARAMETERFRAMEOPT  $\leftarrow$  getEnclosingParameterFrame(env);
  if frame = none or frame.kind  $\neq$  constructorFunction then
    throw a SyntaxError exception — a super statement is meaningful only inside a constructor
  end if;
  Validate[Arguments](cxt, env);
  frame.callsSuperconstructor  $\leftarrow$  true
end proc;
```

Setup

```
proc Setup[SuperStatement  $\Rightarrow$  super Arguments] ()
  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;

```

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

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

```

Labeled Statements

Syntax

LabeledStatement $\Rightarrow$ Identifier : Substatement

Validation

```

proc Validate[LabeledStatement  $\Rightarrow$  Identifier : Substatement]
  (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL {}, jt: JUMPTARGETS)
  name: STRING  $\leftarrow$  Name[Identifier];
  if name  $\in$  jt.breakTargets then
    throw a SyntaxError exception — nesting labeled statements with the same label is not permitted
  end if;
  jt2: JUMPTARGETS  $\leftarrow$  JUMPTARGETS {breakTargets: jt.breakTargets  $\cup$  {name},
    continueTargets: jt.continueTargets};
  Validate[Substatement](cxt, env, sl  $\cup$  {name}, jt2)
end proc;

```

Setup

```

proc Setup[LabeledStatement  $\Rightarrow$  Identifier : Substatement] ()
  Setup[Substatement]()
end proc;

```

Evaluation

```

proc Eval[LabeledStatement  $\Rightarrow$  Identifier : Substatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  try return Eval[Substatement](env, d)
  catch x: SEMANTICEXCEPTION do
    if x  $\in$  BREAK and x.label = Name[Identifier] then return x.value
    else throw x
    end if
  end try
end proc;

```


If Statement

Syntax

$\text{IfStatement}^{\text{abbrev}} \Rightarrow$
 $\text{if ParenListExpression Substatement}^{\text{abbrev}}$
 $| \text{if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{abbrev}}$
 $\text{IfStatement}^{\text{full}} \Rightarrow$
 $\text{if ParenListExpression Substatement}^{\text{full}}$
 $| \text{if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{full}}$
 $\text{IfStatement}^{\text{noShortIf}} \Rightarrow \text{if ParenListExpression Substatement}^{\text{noShortIf}} \text{ else Substatement}^{\text{noShortIf}}$

Validation

```

proc Validate[IfStatement-] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [IfStatementabbrev ⇒ if ParenListExpression Substatementabbrev] do
    Validate[ParenListExpression](cxt, env);
    Validate[Substatementabbrev](cxt, env, {}, jt);
  [IfStatementfull ⇒ if ParenListExpression Substatementfull] do
    Validate[ParenListExpression](cxt, env);
    Validate[Substatementfull](cxt, env, {}, jt);
  [IfStatement- ⇒ if ParenListExpression SubstatementnoShortIf1 else Substatement2] do
    Validate[ParenListExpression](cxt, env);
    Validate[SubstatementnoShortIf1](cxt, env, {}, jt);
    Validate[Substatement2](cxt, env, {}, jt)
end proc;

```

Setup

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

Evaluation

```

proc Eval[IfStatement-] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [IfStatementabbrev ⇒ if ParenListExpression Substatementabbrev] do
    o: OBJECT ← readReference(Eval[ParenListExpression](env, run), run);
    if objectToBoolean(o) then return Eval[Substatementabbrev](env, d)
    else return d
    end if;
  [IfStatementfull ⇒ if ParenListExpression Substatementfull] do
    o: OBJECT ← readReference(Eval[ParenListExpression](env, run), run);
    if objectToBoolean(o) then return Eval[Substatementfull](env, d)
    else return d
    end if;
  [IfStatement- ⇒ if ParenListExpression SubstatementnoShortIf1 else Substatement2] do
    o: OBJECT ← readReference(Eval[ParenListExpression](env, run), run);
    if objectToBoolean(o) then return Eval[SubstatementnoShortIf1](env, d)
    else return Eval[Substatement2](env, d)
    end if
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: if toBoolean(o, run) then return Eval[Substatement^{abbrev}](env, d)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: if toBoolean(o, run) then return Eval[Substatement^{full}](env, d)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: if toBoolean(o, run) then return Eval[Substatement^{noShortIf}₁](env, d)

Switch Statement

Semantics

```
tuple SWITCHKEY
  key: OBJECT
end tuple;
```

$\text{SWITCHGUARD} = \text{SWITCHKEY} \cup \{\text{default}\} \cup \text{OBJECT};$

Syntax

SwitchStatement $\Rightarrow$ **switch** *ParenListExpression* { *CaseElements* }

CaseElements $\Rightarrow$
 «empty»
 | *CaseLabel*
 | *CaseLabel CaseElementsPrefix CaseElement*^{abbrev}

CaseElementsPrefix $\Rightarrow$
 «empty»
 | *CaseElementsPrefix CaseElement*^{full}

CaseElement $\Rightarrow$
Directive·
 | *CaseLabel*

CaseLabel $\Rightarrow$
case *ListExpression*^{allowIn} :
 | **default** :

Validation

CompileFrame[*SwitchStatement*]: **LOCALFRAME**;

```
proc Validate[SwitchStatement  $\Rightarrow$  switch ParenListExpression { CaseElements }]  

  (cxt: 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](cxt, env);  

  jt2: JUMPTARGETS  $\leftarrow$  JUMPTARGETS{breakTargets: jt.breakTargets  $\cup$  {default},  

    continueTargets: jt.continueTargets};  

  compileFrame: LOCALFRAME  $\leftarrow$  new LOCALFRAME{localBindings: {}};  

  CompileFrame[SwitchStatement]  $\leftarrow$  compileFrame;  

  localCxt: CONTEXT  $\leftarrow$  new CONTEXT{strict: cxt.strict, openNamespaces: cxt.openNamespaces};  

  Validate[CaseElements](localCxt, [compileFrame]  $\oplus$  env, jt2)  

end proc;
```

NDefaults[*CaseElements*]: **INTEGER**;
 NDefaults[*CaseElements* $\Rightarrow$ «empty»] = 0;
 NDefaults[*CaseElements* $\Rightarrow$ *CaseLabel*] = NDefaults[*CaseLabel*];
 NDefaults[*CaseElements* $\Rightarrow$ *CaseLabel CaseElementsPrefix CaseElement*^{abbrev}]
 = NDefaults[*CaseLabel*] + NDefaults[*CaseElementsPrefix*] + NDefaults[*CaseElement*^{abbrev}];

Validate[*CaseElements*] (*cxt*: **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 CaseElementfull]
  = NDefaults[CaseElementsPrefix1] + NDefaults[CaseElementfull];

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

NDefaults[CaseElement-]: INTEGER;
NDefaults[CaseElement- ⇒ Directive-] = 0;
NDefaults[CaseElement- ⇒ CaseLabel] = NDefaults[CaseLabel];

proc Validate[CaseElement-] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [CaseElement- ⇒ Directive-] do Validate[Directive-](cxt, env, jt, false, none);
  [CaseElement- ⇒ CaseLabel] do Validate[CaseLabel](cxt, env, jt)
end proc;

NDefaults[CaseLabel]: INTEGER;
NDefaults[CaseLabel ⇒ case ListExpressionallowin :] = 0;
NDefaults[CaseLabel ⇒ default :] = 1;

proc Validate[CaseLabel] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [CaseLabel ⇒ case ListExpressionallowin :] do
    Validate[ListExpressionallowin](cxt, 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⁻] () propagates the call to **Setup** to every nonterminal in the expansion of *CaseElement⁻*.

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

Evaluation

```

proc Eval[SwitchStatement  $\Rightarrow$  switch ParenListExpression { CaseElements }]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
  key: OBJECT  $\leftarrow$  readReference(Eval[ParenListExpression](env, run), run);
  compileFrame: LOCALFRAME  $\leftarrow$  CompileFrame[SwitchStatement];
  runtimeFrame: LOCALFRAME  $\leftarrow$  instantiateLocalFrame(compileFrame, env);
  runtimeEnv: ENVIRONMENT  $\leftarrow$  [runtimeFrame]  $\oplus$  env;
  result: SWITCHGUARD  $\leftarrow$  Eval[CaseElements](runtimeEnv, SWITCHKEY{key: key}, d);
  if result  $\in$  OBJECT then return result end if;
  note result = SWITCHKEY{key: key};
  result  $\leftarrow$  Eval[CaseElements](runtimeEnv, default, d);
  if result  $\in$  OBJECT then return result end if;
  note result = default;
  return d
end proc;

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

proc Eval[CaseElementsPrefix] (env: ENVIRONMENT, guard: SWITCHGUARD, d: OBJECT): SWITCHGUARD
  [CaseElementsPrefix  $\Rightarrow$  «empty»] do return guard;
  [CaseElementsPrefix0  $\Rightarrow$  CaseElementsPrefix1 CaseElementfull] do
    guard2: SWITCHGUARD  $\leftarrow$  Eval[CaseElementsPrefix1](env, guard, d);
    return Eval[CaseElementfull](env, guard2, d)
  end proc;

proc Eval[CaseElement-] (env: ENVIRONMENT, guard: SWITCHGUARD, d: OBJECT): SWITCHGUARD
  [CaseElement-  $\Rightarrow$  Directive-] do
    case guard of
      SWITCHKEY  $\cup$  {default} do return guard;
      OBJECT do return Eval[Directive-](env, guard)
    end case;
  [CaseElement-  $\Rightarrow$  CaseLabel] do return Eval[CaseLabel](env, guard, d)
  end proc;

proc Eval[CaseLabel] (env: ENVIRONMENT, guard: SWITCHGUARD, d: OBJECT): SWITCHGUARD
  [CaseLabel  $\Rightarrow$  case ListExpressionallowin : ] do
    case guard of
      {default}  $\cup$  OBJECT do return guard;
      SWITCHKEY do
        label: OBJECT  $\leftarrow$  readReference(Eval[ListExpressionallowin](env, run), run);
        if isStrictlyEqual(guard.key, label, run) then return d
        else return guard
        end if
      end case;
  end proc;

```

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

Do-While Statement

Syntax

DoStatement ⇒ **do** *Substatement*^{abbrev} **while** *ParenListExpression*

Validation

```
Labels[DoStatement]: LABEL{};

proc Validate[DoStatement ⇒ do Substatementabbrev while ParenListExpression]
  (cxt: 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[Substatementabbrev](cxt, env, {}, jt2);
  Validate[ParenListExpression](cxt, env)
end proc;
```

Setup

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

Evaluation

```
proc Eval[DoStatement ⇒ do Substatementabbrev while ParenListExpression]
  (env: ENVIRONMENT, d: OBJECT): OBJECT
  try
    d1: OBJECT ← d;
    while true do
      try d1 ← Eval[Substatementabbrev](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 objectToBoolean(o) 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;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

While Statement

Syntax

WhileStatement $\Rightarrow$ **while** *ParenListExpression Substatement*

Validation

```
Labels[WhileStatement]: LABEL{};

proc Validate[WhileStatement  $\Rightarrow$  while ParenListExpression Substatement]
  (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL{}, jt: JUMPTARGETS)
  continueLabels: LABEL{}  $\leftarrow$  sl  $\cup$  {default};
  Labels[WhileStatement]  $\leftarrow$  continueLabels;
  jt2: JUMPTARGETS  $\leftarrow$  JUMPTARGETS{breakTargets: jt.breakTargets  $\cup$  {default},
    continueTargets: jt.continueTargets  $\cup$  continueLabels};
  Validate[ParenListExpression](cxt, env);
  Validate[Substatement](cxt, env, {}, jt2)
end proc;
```

Setup

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

Evaluation

```
proc Eval[WhileStatement  $\Rightarrow$  while ParenListExpression Substatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  try
    dl: OBJECT  $\leftarrow$  d;
    while objectToBoolean(readReference(Eval[ParenListExpression](env, run), run)) do
      try dl  $\leftarrow$  Eval[Substatement](env, dl)
      catch x: SEMANTICEXCEPTION do
        if x  $\in$  CONTINUE and x.label  $\in$  Labels[WhileStatement] then
          dl  $\leftarrow$  x.value
        else throw x
        end if
      end try
    end while;
    return dl
  catch x: SEMANTICEXCEPTION do
    if x  $\in$  BREAK and x.label = default then return x.value else throw x end if
  end try
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: while toBoolean(readReference(Eval[ParenListExpression](env, run), run), run) do

For Statements

Syntax

ForStatement $\Rightarrow$
~~for~~ (*ForInitializer* ; *OptionalExpression* ; *OptionalExpression*) *Substatement*
| *for* (*ForInBinding* **in** *ListExpression*^{allowin}) *Substatement*

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: **for** (*ForInitialiser* ; *OptionalExpression* ; *OptionalExpression*) *Substatement*

ForInitializer ⇒

```

«empty»
| ListExpressionnoln
| VariableDefinitionnoln
| Attributes [no line break] VariableDefinitionnoln

```

ForInBinding ⇒

```

PostfixExpression
| VariableDefinitionKind VariableBindingnoln
| Attributes [no line break] VariableDefinitionKind VariableBindingnoln

```

OptionalExpression ⇒

```

ListExpressionallowin
| «empty»

```

Validation

Labels[ForStatement-]: LABEL {};

CompileLocalFrame[ForStatement-]: LOCALFRAME;

proc Validate[ForStatement-] (cxt: CONTEXT, env: ENVIRONMENT, sl: LABEL {}, jt: JUMPTARGETS)

[ForStatement- ⇒ for (ForInitializer ; OptionalExpression₁ ; OptionalExpression₂) Substatement-] do

```

  continueLabels: LABEL {} ← sl ∪ {default};
  Labels[ForStatement-] ← continueLabels;
  jt2: JUMPTARGETS ← JUMPTARGETS{breakTargets: jt.breakTargets ∪ {default},
    continueTargets: jt.continueTargets ∪ continueLabels};
  compileLocalFrame: LOCALFRAME ← new LOCALFRAME{localBindings: {}};
  CompileLocalFrame[ForStatement-] ← compileLocalFrame;
  compileEnv: ENVIRONMENT ← [compileLocalFrame] ⊕ env;
  Validate[ForInitializer](cxt, compileEnv);
  Validate[OptionalExpression1](cxt, compileEnv);
  Validate[OptionalExpression2](cxt, compileEnv);
  Validate[Substatement-](cxt, compileEnv, {}, jt2);

```

[ForStatement- ⇒ for (ForInBinding in ListExpression^{allowin}) Substatement-] do

```

  continueLabels: LABEL {} ← sl ∪ {default};
  Labels[ForStatement-] ← continueLabels;
  jt2: JUMPTARGETS ← JUMPTARGETS{breakTargets: jt.breakTargets ∪ {default},
    continueTargets: jt.continueTargets ∪ continueLabels};
  Validate[ListExpressionallowin](cxt, env);
  compileLocalFrame: LOCALFRAME ← new LOCALFRAME{localBindings: {}};
  CompileLocalFrame[ForStatement-] ← compileLocalFrame;
  compileEnv: ENVIRONMENT ← [compileLocalFrame] ⊕ env;
  Validate[ForInBinding](cxt, compileEnv);
  Validate[Substatement-](cxt, compileEnv, {}, jt2)

```

end proc;

Enabled[ForInitializer]: BOOLEAN;

proc Validate[ForInitializer] (cxt: CONTEXT, env: ENVIRONMENT)

[ForInitializer ⇒ «empty»] do nothing;[ForInitializer ⇒ ListExpression^{noln}] do Validate[ListExpression^{noln}](cxt, env);[ForInitializer ⇒ VariableDefinition^{noln}] doValidate[VariableDefinition^{noln}](cxt, env, none);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: ForInitialiser ⇒

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: [ForStatement- ⇒ for (ForInitialiser ; OptionalExpression₁ ; OptionalExpression₂) Substatement-] do

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: Validate[ForInitialiser](cxt, compileEnv);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: Enabled[ForInitialiser]: B ... [43]

$[ForInitializer \Rightarrow Attributes \text{ [no line break] } VariableDefinition^{noin}] \text{ do}$
 Validate[Attributes](cxt, env);
 Setup[Attributes]();
 attr: ATTRIBUTE $\leftarrow$ Eval[Attributes](env, compile);
 $Enabled[ForInitializer] \leftarrow attr \neq \text{false};$
 if attr $\neq$ false then Validate[VariableDefinition^{noin}](cxt, env, attr) end if
end proc;

proc Validate[ForInBinding] (cxt: CONTEXT, env: ENVIRONMENT)
 [ForInBinding $\Rightarrow$ PostfixExpression] do Validate[PostfixExpression](cxt, env);
 [ForInBinding $\Rightarrow$ VariableDefinitionKind VariableBinding^{noin}] do
 Validate[VariableBinding^{noin}](cxt, env, none, Immutable[VariableDefinitionKind], true);
 [ForInBinding $\Rightarrow$ Attributes [no line break] VariableDefinitionKind VariableBinding^{noin}] do
 Validate[Attributes](cxt, env);
 Setup[Attributes]();
 attr: ATTRIBUTE $\leftarrow$ 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^{noin}](cxt, env, attr, Immutable[VariableDefinitionKind], true)
 end proc;

Validate[OptionalExpression] (cxt: 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[ForInitializer] ()
 $[ForInitializer \Rightarrow \text{«empty»}] \text{ do nothing;}$
 $[ForInitializer \Rightarrow ListExpression^{noin}] \text{ do Setup[ListExpression}^{noin}]();$
 $[ForInitializer \Rightarrow VariableDefinition^{noin}] \text{ do Setup[VariableDefinition}^{noin}]();$
 $[ForInitializer \Rightarrow Attributes \text{ [no line break] } VariableDefinition^{noin}] \text{ do}$
if Enabled[ForInitializer] then Setup[VariableDefinition^{noin}]() end if
end proc;

proc Setup[ForInBinding] ()
 [ForInBinding $\Rightarrow$ PostfixExpression] do Setup[PostfixExpression]();
 [ForInBinding $\Rightarrow$ VariableDefinitionKind VariableBinding^{noin}] do
 Setup[VariableBinding^{noin}]();
 [ForInBinding $\Rightarrow$ Attributes [no line break] VariableDefinitionKind VariableBinding^{noin}] do
 Setup[VariableBinding^{noin}]()
 end proc;

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

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: $[ForInitialiser \Rightarrow Attributes \text{ [no line break] } VariableDefinition^{noin}] \text{ do}$

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: $Enabled[ForInitialiser] \leftarrow attr \neq \text{false};$

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: proc Setup[ForInitialiser] () [44]

Evaluation

```

proc Eval[ForStatement-] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [ForStatement-  $\Rightarrow$  for ( ForInitializer ; OptionalExpression1 ; OptionalExpression2 ) Substatement-] do
    runtimeLocalFrame: LOCALFRAME  $\leftarrow$  instantiateLocalFrame(CompileLocalFrame[ForStatement-], env);
    runtimeEnv: ENVIRONMENT  $\leftarrow$  [runtimeLocalFrame]  $\oplus$  env;
    try
      Eval[ForInitializer](runtimeEnv);
      dI: OBJECT  $\leftarrow$  d;
      while objectToBoolean(readReference(Eval[OptionalExpression1](runtimeEnv, run), run)) do
        try dI  $\leftarrow$  Eval[Substatement-](runtimeEnv, dI)
        catch x: SEMANTICEXCEPTION do
          if x  $\in$  CONTINUE and x.label  $\in$  Labels[ForStatement-] then
            dI  $\leftarrow$  x.value
            else throw x
          end if
        end try;
        readReference(Eval[OptionalExpression2](runtimeEnv, run), run)
      end while;
      return dI
    catch x: SEMANTICEXCEPTION do
      if x  $\in$  BREAK and x.label = default then return x.value else throw x end if
    end try;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: [ForStatement- $\Rightarrow$ **for** (ForInitialiser ; OptionalExpression₁ ; OptionalExpression₂) Substatement-] do

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: Eval[ForInitialiser](runtimeEnv);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: while toBoolean(readReference(Eval[OptionalExpression₁](runtimeEnv, run), run)) do

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

```

[ForStatement  $\Rightarrow$  for ( ForInBinding in ListExpressionallowIn ) Substatement- ] do
  try
    o: OBJECT  $\leftarrow$  readReference(Eval[ListExpressionallowIn](env, run), run);
    c: CLASS  $\leftarrow$  objectType(o);
    oldIndices: OBJECT{}  $\leftarrow$  c.enumerate(o);
    remainingIndices: OBJECT{}  $\leftarrow$  oldIndices;
    dI: OBJECT  $\leftarrow$  d;
    while remainingIndices  $\neq$  {} do
      runtimeLocalFrame: LOCALFRAME  $\leftarrow$  instantiateLocalFrame(CompileLocalFrame[ForStatement-], env);
      runtimeEnv: ENVIRONMENT  $\leftarrow$  [runtimeLocalFrame]  $\oplus$  env;
      index: OBJECT  $\leftarrow$  any element of remainingIndices;
      remainingIndices  $\leftarrow$  remainingIndices - {index};
      WriteBinding[ForInBinding](runtimeEnv, index);
      try dI  $\leftarrow$  Eval[Substatement-](runtimeEnv, dI)
      catch x: SEMANTICEXCEPTION do
        if x  $\in$  CONTINUE and x.label  $\in$  Labels[ForStatement-] then
          dI  $\leftarrow$  x.value
        else throw x
        end if
      end try;
      newIndices: OBJECT{}  $\leftarrow$  c.enumerate(o);
      if newIndices  $\neq$  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  $\leftarrow$  newIndices
    end while;
    return dI
  catch x: SEMANTICEXCEPTION do
    if x  $\in$  BREAK and x.label = default then return x.value else throw x end if
  end try
end proc;

```

proc Eval[ForInitializer] (env: ENVIRONMENT)

[ForInitializer $\Rightarrow$ «empty»] **do nothing**;

[ForInitializer $\Rightarrow$ ListExpression^{noIn}] **do**

readReference(Eval[ListExpression^{noIn}](env, run), run);

[ForInitializer $\Rightarrow$ VariableDefinition^{noIn}] **do**

Eval[VariableDefinition^{noIn}](env, undefined);

[ForInitializer $\Rightarrow$ Attributes [no line break] VariableDefinition^{noIn}] **do**

if Enabled[ForInitializer] then Eval[VariableDefinition^{noIn}](env, undefined)

end if

end proc;

proc WriteBinding[ForInBinding] (env: ENVIRONMENT, newValue: OBJECT)

[ForInBinding $\Rightarrow$ PostfixExpression] **do**

r: OBJORREF $\leftarrow$ Eval[PostfixExpression](env, run);

writeReference(r, newValue, run);

[ForInBinding $\Rightarrow$ VariableDefinitionKind VariableBinding^{noIn}] **do**

WriteBinding[VariableBinding^{noIn}](env, newValue);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: **proc** Eval[ForInitialiser] (env: ENVIRONMENT) - ... [45]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: [ForInitialiser $\Rightarrow$ VariableDefinition^{noIn}] **do**

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: [ForInitialiser $\Rightarrow$ Attributes [no line break] VariableDefinition^{noIn}] **do** - ... [46]

```

    [ForInBinding  $\Rightarrow$  Attributes [no line break] VariableDefinitionKind VariableBindingnolin] do
      WriteBinding[VariableBindingnolin](env, newValue)
    end proc;

proc Eval[OptionalExpression] (env: ENVIRONMENT, phase: PHASE): OBJORREF
  [OptionalExpression  $\Rightarrow$  ListExpressionallowin] do
    return Eval[ListExpressionallowin](env, phase);
  [OptionalExpression  $\Rightarrow$  «empty»] do return true
end proc;

```

With Statement

Syntax

WithStatement $\Rightarrow$ **with** ParenListExpression Substatement

Validation

```

CompileLocalFrame[WithStatement]: LOCALFRAME;

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

```

Setup

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

Evaluation

```

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

```

Continue and Break Statements

Syntax

ContinueStatement $\Rightarrow$
 continue
 | **continue** [no line break] Identifier

```

BreakStatement ⇒
  break
| 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 [no line break] Identifier] 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 [no line break] Identifier] 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 [no line break] Identifier] do nothing
end proc;

proc Setup[BreakStatement] ()
  [BreakStatement ⇒ break] do nothing;
  [BreakStatement ⇒ break [no line break] Identifier] do nothing
end proc;

```

Evaluation

```

proc Eval[ContinueStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [ContinueStatement ⇒ continue] do throw CONTINUE⟨value: d, label: default⟩;
  [ContinueStatement ⇒ continue [no line break] Identifier] do
    throw CONTINUE⟨value: d, label: Name[Identifier]⟩
  end proc;

proc Eval[BreakStatement] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [BreakStatement ⇒ break] do throw BREAK⟨value: d, label: default⟩;
  [BreakStatement ⇒ break [no line break] Identifier] do
    throw BREAK⟨value: d, label: Name[Identifier]⟩
  end proc;

```

Return Statement

Syntax

ReturnStatement $\Rightarrow$
 return
 | **return** [no line break] *ListExpression*^{allowIn}

Validation

```

proc Validate[ReturnStatement] (cxt: CONTEXT, env: ENVIRONMENT)
  [ReturnStatement  $\Rightarrow$  return] do
    if getEnclosingParameterFrame(env) = none then
      throw a SyntaxError exception — a return statement must be located inside a function
    end if;
  [ReturnStatement  $\Rightarrow$  return [no line break] ListExpressionallowIn] do
    frame: PARAMETERFRAMEOPT  $\leftarrow$  getEnclosingParameterFrame(env);
    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[ListExpressionallowIn](cxt, env)
  end proc;

```

Setup

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

Evaluation

```

proc Eval[ReturnStatement] (env: ENVIRONMENT): OBJECT
  [ReturnStatement  $\Rightarrow$  return] do throw RETURN(value: undefined);
  [ReturnStatement  $\Rightarrow$  return [no line break] ListExpressionallowIn] do
    a: OBJECT  $\leftarrow$  readReference(Eval[ListExpressionallowIn](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.

```

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

```

Throw Statement

Syntax

ThrowStatement $\Rightarrow$ **throw** [no line break] *ListExpression*^{allowIn}

Validation

```

proc Validate[ThrowStatement  $\Rightarrow$  throw [no line break] ListExpressionallowIn] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[ListExpressionallowIn](cxt, env)
end proc;

```

Setup

```

proc Setup[ThrowStatement ⇒ throw [no line break] ListExpressionallowin] ()
  Setup[ListExpressionallowin]()
end proc;

```

Evaluation

```

proc Eval[ThrowStatement ⇒ throw [no line break] ListExpressionallowin] (env: ENVIRONMENT): OBJECT
  a: OBJECT ← readReference(Eval[ListExpressionallowin](env, run), run);
  throw a
end proc;

```

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] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  [TryStatement ⇒ try Block CatchClauses] do
    Validate[Block](cxt, env, jt, false);
    Validate[CatchClauses](cxt, env, jt);
  [TryStatement ⇒ try Block1 CatchClausesOpt finally Block2] do
    Validate[Block1](cxt, env, jt, false);
    Validate[CatchClausesOpt](cxt, env, jt);
    Validate[Block2](cxt, env, jt, false)
  end proc;

```

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

Validate[CatchClauses] (cxt: 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] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS)
  compileFrame: LOCALFRAME ← new LOCALFRAME(localBindings: {});
  compileEnv: ENVIRONMENT ← [compileFrame] ⊕ env;
  CompileFrame[CatchClause] ← compileFrame;
  CompileEnv[CatchClause] ← compileEnv;
  Validate[Parameter](cxt, compileEnv, compileFrame);
  Validate[Block](cxt, 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  $\Rightarrow$  try Block1 CatchClausesOpt finally Block2] do
  result: OBJECTOPT  $\leftarrow$  none;
  exception: SEMANTICEXCEPTION  $\cup$  {none}  $\leftarrow$  none;
  try result  $\leftarrow$  Eval[Block1](env, d)
  catch x: SEMANTICEXCEPTION do exception  $\leftarrow$  x
  end try;
  note At this point exactly one of result and exception has a non-none value.
  if exception  $\in$  OBJECT then
    try
      r: OBJECT  $\cup$  {reject}  $\leftarrow$  Eval[CatchClausesOpt](env, exception);
      if r  $\neq$  reject then
        note The exception has been handled, so clear it.
        result  $\leftarrow$  r;
        exception  $\leftarrow$  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  $\leftarrow$  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  $\neq$  none then throw exception else return result end if
end proc;

proc Eval[CatchClausesOpt] (env: ENVIRONMENT, exception: OBJECT): OBJECT  $\cup$  {reject}
  [CatchClausesOpt  $\Rightarrow$  «empty»] do return reject;
  [CatchClausesOpt  $\Rightarrow$  CatchClauses] do return Eval[CatchClauses](env, exception)
end proc;

proc Eval[CatchClauses] (env: ENVIRONMENT, exception: OBJECT): OBJECT  $\cup$  {reject}
  [CatchClauses  $\Rightarrow$  CatchClause] do return Eval[CatchClause](env, exception);
  [CatchClauses0  $\Rightarrow$  CatchClauses1 CatchClause] do
    r: OBJECT  $\cup$  {reject}  $\leftarrow$  Eval[CatchClauses1](env, exception);
    if r  $\neq$  reject then return r else return Eval[CatchClause](env, exception) end if
  end proc;

proc Eval[CatchClause  $\Rightarrow$  catch ( Parameter ) Block] (env: ENVIRONMENT, exception: OBJECT): OBJECT  $\cup$  {reject}
  compileFrame: LOCALFRAME  $\leftarrow$  CompileFrame[CatchClause];
  runtimeFrame: LOCALFRAME  $\leftarrow$  instantiateLocalFrame(compileFrame, env);
  runtimeEnv: ENVIRONMENT  $\leftarrow$  [runtimeFrame]  $\oplus$  env;
  qname: QUALIFIEDNAME  $\leftarrow$  public::(Name[Parameter]);
  v: SINGLETONPROPERTYOPT  $\leftarrow$  findLocalSingletonProperty(runtimeFrame, {qname}, write);
  note Validate created one local variable with the name in qname, so v  $\in$  VARIABLE.
  if is(exception, v.type) then
    writeSingletonProperty(v, exception, run);
    return Eval[Block](runtimeEnv, undefined)
  else return reject
  end if
end proc;

```


Directives

Syntax

Directive $\Rightarrow$

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

AnnotatableDirective $\Rightarrow$

- | *VariableDefinition*^{allowIn} *Semicolon*
- | *FunctionDefinition*
- | *ClassDefinition*
- | *NamespaceDefinition* *Semicolon*
- | *ImportDirective* *Semicolon*
- | *UseDirective* *Semicolon*

Directives $\Rightarrow$

- | «empty»
- | *DirectivesPrefix* *Directive*^{abbrev}

DirectivesPrefix $\Rightarrow$

- | «empty»
- | *DirectivesPrefix* *Directive*^{full}

Validation

Enabled[*Directive*]: **BOOLEAN**;

```

proc Validate[Directive](cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN,
  attr: ATTRIBUTEOPTNOTFALSE)
[Directive  $\Rightarrow$  EmptyStatement] do nothing;
[Directive  $\Rightarrow$  Statement] do
  if attr  $\notin$  {none, true} then
    throw an AttributeError exception — an ordinary statement only permits the attributes true and false
  end if;
  Validate[Statement](cxt, env, {}, jt, preinst);
[Directive  $\Rightarrow$  AnnotatableDirective] do
  Validate[AnnotatableDirective](cxt, env, preinst, attr);
[Directive  $\Rightarrow$  Attributes [no line break] AnnotatableDirective] do
  Validate[Attributes](cxt, env);
  Setup[Attributes]();
  attr2: ATTRIBUTE  $\leftarrow$  Eval[Attributes](env, compile);
  attr3: ATTRIBUTE  $\leftarrow$  combineAttributes(attr, attr2);
  if attr3 = false then Enabled[Directive]  $\leftarrow$  false
  else
    Enabled[Directive]  $\leftarrow$  true;
    Validate[AnnotatableDirective](cxt, env, preinst, attr3)
  end if;

```

```

[Directive- ⇒ Attributes [no line break] { Directives- } ] do
  Validate[Attributes](cxt, env);
  Setup[Attributes]();
  attr2: ATTRIBUTE ← Eval[Attributes](env, compile);
  attr3: ATTRIBUTE ← combineAttributes(attr, attr2);
  if attr3 = false then Enabled[Directive-] ← false
  else
    Enabled[Directive-] ← true;
    localCxt: CONTEXT ← new CONTEXT(strict: cxt.strict, openNamespaces: cxt.openNamespaces);
    Validate[Directives](localCxt, env, jt, preinst, attr3)
  end if;
[Directive- ⇒ Pragma Semicolon-] do
  if attr ∈ {none, true} then Validate[Pragma](cxt)
  else
    throw an AttributeError exception — a pragma directive only permits the attributes true and false
  end if
end proc;

proc Validate[AnnotableDirective-]
  (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTE_OPT_NOT_FALSE)
[AnnotableDirective- ⇒ VariableDefinitionallowin Semicolon-] do
  Validate[VariableDefinitionallowin](cxt, env, attr);
[AnnotableDirective- ⇒ FunctionDefinition] do
  Validate[FunctionDefinition](cxt, env, preinst, attr);
[AnnotableDirective- ⇒ ClassDefinition] do
  Validate[ClassDefinition](cxt, env, preinst, attr);
[AnnotableDirective- ⇒ NamespaceDefinition Semicolon-] do
  Validate[NamespaceDefinition](cxt, env, preinst, attr);
[AnnotableDirective- ⇒ ImportDirective Semicolon-] do
  Validate[ImportDirective](cxt, env, preinst, attr);
[AnnotableDirective- ⇒ UseDirective Semicolon-] do
  if attr ∈ {none, true} then Validate[UseDirective](cxt, env)
  else
    throw an AttributeError exception — a use directive only permits the attributes true and false
  end if
end proc;

Validate[Directives] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN,
  attr: ATTRIBUTE_OPT_NOT_FALSE) propagates the call to Validate to every nonterminal in the expansion of
Directives.

Validate[DirectivesPrefix] (cxt: CONTEXT, env: ENVIRONMENT, jt: JUMPTARGETS, preinst: BOOLEAN,
  attr: ATTRIBUTE_OPT_NOT_FALSE) propagates the call to Validate to every nonterminal in the expansion of
DirectivesPrefix.

```

Setup

```

proc Setup[Directive-] ()
  [Directive- ⇒ EmptyStatement] do nothing;
  [Directive- ⇒ Statement-] do Setup[Statement-]();
  [Directive- ⇒ AnnotableDirective-] do Setup[AnnotableDirective-]();

```

```

[Directive- ⇒ Attributes [no line break] AnnotatableDirective-] do
  if Enabled[Directive-] then Setup[AnnotatableDirective-]() end if;
[Directive- ⇒ Attributes [no line break] { Directives } ] do
  if Enabled[Directive-] then Setup[Directives]() end if;
[Directive- ⇒ Pragma Semicolon-] do nothing
end proc;

proc Setup[AnnotatableDirective-] ()
  [AnnotatableDirective- ⇒ VariableDefinitionallowIn Semicolon-] do
    Setup[VariableDefinitionallowIn]();
  [AnnotatableDirective- ⇒ FunctionDefinition] do Setup[FunctionDefinition]();
  [AnnotatableDirective- ⇒ ClassDefinition] do Setup[ClassDefinition]();
  [AnnotatableDirective- ⇒ NamespaceDefinition Semicolon-] do nothing;
  [AnnotatableDirective- ⇒ ImportDirective Semicolon-] do nothing;
  [AnnotatableDirective- ⇒ UseDirective 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);
  [Directive- ⇒ Attributes [no line break] AnnotatableDirective-] do
    if Enabled[Directive-] then return Eval[AnnotatableDirective-](env, d)
    else return d
  end if;
  [Directive- ⇒ Attributes [no line break] { Directives } ] do
    if Enabled[Directive-] then return Eval[Directives](env, d) else return d end if;
  [Directive- ⇒ Pragma Semicolon-] do return d
end proc;

proc Eval[AnnotatableDirective-] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [AnnotatableDirective- ⇒ VariableDefinitionallowIn Semicolon-] do
    return Eval[VariableDefinitionallowIn](env, d);
  [AnnotatableDirective- ⇒ FunctionDefinition] do return d;
  [AnnotatableDirective- ⇒ ClassDefinition] do return Eval[ClassDefinition](env, d);
  [AnnotatableDirective- ⇒ NamespaceDefinition Semicolon-] do return d;
  [AnnotatableDirective- ⇒ ImportDirective Semicolon-] do return d;
  [AnnotatableDirective- ⇒ UseDirective Semicolon-] do return d
end proc;

proc Eval[Directives] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [Directives ⇒ «empty»] do return d;

```

```

[Directives  $\Rightarrow$  DirectivesPrefix Directiveabbrev] do
  o: OBJECT  $\leftarrow$  Eval[DirectivesPrefix](env, d);
  return Eval[Directiveabbrev](env, o)
end proc;

proc Eval[DirectivesPrefix] (env: ENVIRONMENT, d: OBJECT): OBJECT
  [DirectivesPrefix  $\Rightarrow$  «empty»] do return d;
  [DirectivesPrefix0  $\Rightarrow$  DirectivesPrefix1 Directivefull] do
    o: OBJECT  $\leftarrow$  Eval[DirectivesPrefix1](env, d);
    return Eval[Directivefull](env, o)
  end proc;

```

Attributes

Syntax

```

Attributes  $\Rightarrow$ 
  Attribute
  | AttributeCombination

AttributeCombination  $\Rightarrow$  Attribute [no line break] Attributes

Attribute  $\Rightarrow$ 
  AttributeExpression
  | true
  | false
  | ReservedNamespace

```

Validation

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

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

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

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*.

```

proc Setup[Attribute] ()
  [Attribute  $\Rightarrow$  AttributeExpression] do Setup[AttributeExpression]();
  [Attribute  $\Rightarrow$  true] do nothing;
  [Attribute  $\Rightarrow$  false] do nothing;
  [Attribute  $\Rightarrow$  ReservedNamespace] 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);
    return objectToAttribute(a, phase);
  [Attribute ⇒ true] do return true;
  [Attribute ⇒ false] do return false;
  [Attribute ⇒ ReservedNamespace] do return Eval[ReservedNamespace](env, phase)
end proc;
```

Use Directive

Syntax

UseDirective ⇒ **use namespace** ParenListExpression

Validation

```
proc Validate[UseDirective ⇒ use namespace ParenListExpression] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[ParenListExpression](cxt, env);
  Setup[ParenListExpression]();
  values: OBJECT[] ← EvalAsList[ParenListExpression](env, compile);
  namespaces: NAMESPACE{} ← {};
  for each v ∈ values do
    if v ∉ NAMESPACE then throw a TypeError exception end if;
    namespaces ← namespaces ∪ {v}
  end for each;
  cxt.openNamespaces ← cxt.openNamespaces ∪ namespaces
end proc;
```

Import Directive

Syntax

ImportDirective ⇒
import PackageName
import Identifier = PackageName

Validation

```

proc Validate[ImportDirective] (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTEOPTNOTFALSE)
  [ImportDirective ⇒ import PackageName] do
    if not preinst then
      throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
    end if;
    frame: FRAME ← env[0];
    if frame ∉ PACKAGE then
      throw a SyntaxError exception — a package may be imported only into a package scope
    end if;
    if attr ∉ {none, true} then
      throw an AttributeError exception — an unnamed import directive only permits the attributes true and
        false
    end if;
    pkgName: STRING ← Name[PackageName];
    pkg: PACKAGE ← locatePackage(pkgName);
    importPackageInto(pkg, frame);
  [ImportDirective ⇒ import Identifier = PackageName] do
    if not preinst then
      throw a SyntaxError exception — a package may be imported only in a preinstantiated scope
    end if;
    frame: FRAME ← env[0];
    if frame ∉ PACKAGE then
      throw a SyntaxError exception — a package may be imported only into a package scope
    end if;
    a: COMPOUNDATTRIBUTE ← toCompoundAttribute(attr);
    if a.dynamic then
      throw an AttributeError exception — a package definition cannot have the dynamic attribute
    end if;
    if a.prototype then
      throw an AttributeError exception — a package definition cannot have the prototype attribute
    end if;
    pkgName: STRING ← Name[PackageName];
    pkg: PACKAGE ← locatePackage(pkgName);
    v: VARIABLE ← new VARIABLE{type: Package, value: pkg, immutable: true, setup: none, initializer: none};
    defineSingletonProperty(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v);
    importPackageInto(pkg, frame)
  end proc;

proc locatePackage(name: STRING): PACKAGE
  Look for a package bound to name in the implementation's list of available packages. If one is found, let pkg: PACKAGE
  be that package; otherwise, throw an implementation-defined error.
  initialize: (() → ()) ∪ {none, busy} ← pkg.initialize;
  case initialize of
    {none} do nothing;
    {busy} do throw an UninitializedError exception — circular package dependency;
    () → () do initialize(); note pkg.initialize = none;
  end case;
  return pkg
end proc;

```

```

proc importPackageInto(source: PACKAGE, destination: PACKAGE)
  for each b ∈ source.localBindings do
    if not (b.explicit or b.content = forbidden or (some d ∈ destination.localBindings satisfies
      b.qname = d.qname and accessesOverlap(b.accesses, d.accesses))) then
      destination.localBindings ← destination.localBindings ∪ {b}
    end if
  end for each
end proc;

```

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 PragmaItems] (cxt: CONTEXT)
  Validate[PragmaItems](cxt)
end proc;

```

Validate[*PragmaItems*] (*cxt*: CONTEXT) propagates the call to *Validate* to every nonterminal in the expansion of *PragmaItems*.

```

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

```

```

proc Validate[PragmaExpr] (cxt: CONTEXT, optional: BOOLEAN)
  [PragmaExpr ⇒ Identifier] do
    processPragma(cxt, Name[Identifier], undefined, optional);
  [PragmaExpr ⇒ Identifier ( PragmaArgument )] do
    arg: OBJECT ← Value[PragmaArgument];
    processPragma(cxt, Name[Identifier], arg, optional)
end proc;

```

```
Value[PragmaArgument]: OBJECT;  
Value[PragmaArgument ⇒ true] = true;  
Value[PragmaArgument ⇒ false] = false;  
Value[PragmaArgument ⇒ Number] = Value[Number];  
Value[PragmaArgument ⇒ - Number] = generalNumberNegate(Value[Number]);  
Value[PragmaArgument ⇒ - NegatedMinLong] = (-263)long;  
Value[PragmaArgument ⇒ String] = Value[String];  
  
proc processPragma(cxt: CONTEXT, name: STRING, value: OBJECT, optional: BOOLEAN)  
  if name = "strict" then  
    if value ∈ {true, undefined} then cxt.strict ← true; return end if;  
    if value = false then cxt.strict ← false; return end if  
  end if;  
  if name = "ecmascript" then  
    if value ∈ {undefined, 4.0164} then return end if;  
    if value ∈ {1.0164, 2.0164, 3.0164} then  
      An implementation may optionally modify cxt 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;
```

Definitions

Variable Definition

Syntax

VariableDefinition^{*} ⇒ VariableDefinitionKind VariableBindingList^{*}

VariableDefinitionKind ⇒
var
| const

VariableBindingList^{*} ⇒
VariableBinding^{*}
| VariableBindingList^{*} , VariableBinding^{*}

VariableBinding^{*} ⇒ TypedIdentifier^{*} VariableInitialisation^{*}

VariableInitialisation^{*} ⇒
«empty»
| VariableInitializer

VariableInitializer^{*} ⇒
AssignmentExpression^{*}
| AttributeCombination

TypedIdentifier^{*} ⇒
Identifier^{*}
| Identifier^{*} : TypeExpression^{*}

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Validation

```
proc Validate[VariableDefinition ⇒ VariableDefinitionKind VariableBindingList]  
  (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE)  
  Validate[VariableBindingList](cxt, env, attr, Immutable[VariableDefinitionKind], false)  
end proc;
```

```
Immutable[VariableDefinitionKind]: BOOLEAN;  
Immutable[VariableDefinitionKind ⇒ var] = false;  
Immutable[VariableDefinitionKind ⇒ const] = true;
```

```
Validate[VariableBindingList] (cxt: CONTEXT, env: ENVIRONMENT, attr: ATTRIBUTEOPTNOTFALSE,  
  immutable: BOOLEAN, noInitializer: BOOLEAN) propagates the call to Validate to every nonterminal in the  
  expansion of VariableBindingList.
```

```
CompileEnv[VariableBinding]: ENVIRONMENT;
```

```
CompileVar[VariableBinding]: VARIABLE ∪ DYNAMICVAR ∪ INSTANCEVARIABLE;
```

```
OverriddenVar[VariableBinding]: INSTANCEVARIABLEOPT;
```

```
Multiname[VariableBinding]: MULTINAME;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *noInitialiser*: **BOOLEAN**)

```

proc Validate[VariableBinding* ⇒ TypedIdentifier VariableInitialisation*] (cxt: CONTEXT, env: ENVIRONMENT,
  attr: ATTRIBUTE OPT NOT FALSE, immutable: BOOLEAN, noInitializer: BOOLEAN)
  Validate[TypedIdentifier](cxt, env);
  Validate[VariableInitialisation*](cxt, env);
  CompileEnv[VariableBinding*] ← env;
  name: STRING ← Name[TypedIdentifier];
  if not cxt.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;
    category: PROPERTYCATEGORY ← a.category;
    if env[0] ∈ CLASS then if category = none then category ← final end if
  else
    if category ≠ none then
      throw an AttributeError exception — non-class variables cannot have a static, virtual, or final
        attribute
    end if;
  end if;
  case category of
    {none, static} do
      initializer: INITIALISEROPT ← Initializer[VariableInitialisation*];
      if noInitializer and initializer ≠ 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*]();
        return type
      end proc;
      v: VARIABLE ← new VARIABLE{value: none, immutable: immutable, setup: variableSetup,
        initializer: initializer, initializerEnv: env};
      multiline: MULTINAME ← defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit,
        readWrite, v);
      Multiname[VariableBinding*] ← multiline;
      CompileVar[VariableBinding*] ← v;
    {virtual, final} do
      note not noInitializer;
      c: CLASS ← env[0];
      v: INSTANCEVARIABLE ← new INSTANCEVARIABLE{final: category = final, immutable: immutable};
      vOverridden: INSTANCEVARIABLEOPT ← defineInstanceProperty(c, cxt, name, a.namespaces,
        a.overrideMod, a.explicit, v);
      enumerable: BOOLEAN ← a.enumerable;
      if vOverridden ≠ none and vOverridden.enumerable then enumerable ← true
      end if;
      v.enumerable ← enumerable;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: noInitializer: BOOLEAN

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: INITIALISEROPT ← Initialiser[VariableInitialisation*]; ... [48]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: initializer, initializerEnv: env;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: not noInitializer;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: new INSTANCEVARIABLE{final: category = final, immutable: immutable,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: OverriddenVar[VariableBinding*] ←

```

    OverriddenVar[VariableBinding] ← vOverridden;
    CompileVar[VariableBinding] ← v
  end case
end if
end proc;

```

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

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

```

Name[TypedIdentifier]: STRING;
Name[TypedIdentifier ⇒ Identifier] = Name[Identifier];
Name[TypedIdentifier ⇒ Identifier : TypeExpression] = Name[Identifier];

```

```

Plain[TypedIdentifier]: BOOLEAN;
Plain[TypedIdentifier ⇒ Identifier] = true;
Plain[TypedIdentifier ⇒ Identifier : TypeExpression] = false;

```

```

proc Validate[TypedIdentifier] (cxt: CONTEXT, env: ENVIRONMENT)
  [TypedIdentifier ⇒ Identifier] do nothing;
  [TypedIdentifier ⇒ Identifier : TypeExpression] do
    Validate[TypeExpression](cxt, env)
  end proc;

```

Setup

```

proc Setup[VariableDefinition ⇒ VariableDefinitionKind VariableBindingList] ()
  Setup[VariableBindingList]()
end proc;

```

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

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: **Validate**[*VariableInitialiser*]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *VariableInitialiser*.

```

proc Setup[VariableBinding* ⇒ TypedIdentifier VariableInitialisation*] ()
  env: ENVIRONMENT ← CompileEnv[VariableBinding];
  v: VARIABLE ∪ DYNAMICVAR ∪ INSTANCEVARIABLE ← CompileVar[VariableBinding];
  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;
    DYNAMICVAR do Setup[VariableInitialisation]();
    INSTANCEVARIABLE do
      t: CLASSOPT ← SetupAndEval[TypedIdentifier](env);
      if t = none then
        overriddenVar: INSTANCEVARIABLEOPT ← OverriddenVar[VariableBinding];
        if overriddenVar ≠ none then t ← overriddenVar.type
        else t ← Object
        end if
      end if;
      v.type ← t;
      Setup[VariableInitialisation]();
      initializer: INITIALIZEROPT ← Initializer[VariableInitialisation];
      defaultValue: OBJECTOPT ← none;
      if initializer ≠ none then defaultValue ← initializer(env, compile)
      elseif 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
      end if;
      v.defaultValue ← defaultValue
    end case
  end proc;

```

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

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

Evaluation

```

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

```

Eval[VariableBindingList] (env: ENVIRONMENT) propagates the call to Eval to every nonterminal in the expansion of VariableBindingList.

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: INITIALISEROPT ← Initialiser[VariableInitialisation];

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if initializer ≠ none then defaultValue ← initializer(env, compile)

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: Setup[VariableInitialiser];

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: VariableInitialiser.

```

proc Eval[VariableBinding* ⇒ TypedIdentifier* VariableInitialisation*] (env: ENVIRONMENT)
  case CompileVar[VariableBinding*] of
    VARIABLE do
      innerFrame: NONWITHFRAME ← env[0];
      properties: SINGLETONPROPERTY {} ← {b.content | ∀ b ∈ innerFrame.localBindings such that
        b.qname ∈ Multiname[VariableBinding*]};
      note The properties set consists of exactly one VARIABLE element because innerFrame was constructed with
        that VARIABLE inside Validate.
      v: VARIABLE ← the one element of properties;
      initializer: INITIALIZER ∪ {none, busy} ← v.initializer;
      case initializer of
        {none} do nothing;
        {busy} do throw a ReferenceError exception;
      INITIALIZER do
        v.initializer ← busy;
        value: OBJECT ← initializer(v.initializerEnv, run);
        writeVariable(v, value, true)
      end case;
    DYNAMICVAR do
      initializer: INITIALIZEROPT ← Initializer[VariableInitialisation*];
      if initializer ≠ none then
        value: OBJECT ← initializer(env, run);
        lexicalWrite(env, Multiname[VariableBinding*], value, false, run)
      end if;
    INSTANCEVARIABLE do nothing
  end case
end proc;

```

```

proc WriteBinding[VariableBinding* ⇒ TypedIdentifier* VariableInitialisation*]
  (env: ENVIRONMENT, newValue: OBJECT)
  case CompileVar[VariableBinding*] of
    VARIABLE do
      innerFrame: NONWITHFRAME ← env[0];
      properties: SINGLETONPROPERTY {} ← {b.content | ∀ b ∈ innerFrame.localBindings such that
        b.qname ∈ Multiname[VariableBinding*]};
      note The properties set consists of exactly one VARIABLE element because innerFrame was constructed with
        that VARIABLE inside Validate.
      v: VARIABLE ← the one element of properties;
      writeVariable(v, newValue, false);
    DYNAMICVAR do
      lexicalWrite(env, Multiname[VariableBinding*], newValue, false, run)
  end case
end proc;

```

```

Initializer[VariableInitialisation*]: INITIALIZEROPT;
Initializer[VariableInitialisation* ⇒ «empty»] = none;
Initializer[VariableInitialisation* ⇒ = VariableInitializer] = Eval[VariableInitializer*];

```

```

proc Eval[VariableInitializer] (env: ENVIRONMENT, phase: PHASE): OBJECT
  [VariableInitializer ⇒ AssignmentExpression] do
    return readReference(Eval[AssignmentExpression*](env, phase), phase);
  [VariableInitializer ⇒ AttributeCombination] do
    return Eval[AttributeCombination](env, phase)
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: INITIALISER ∪ {none, busy} ← v.initializer; - ... [49]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: INITIALISER do - ... [50]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initializer: INITIALISEROPT ← Initialiser[VariableInitialisation*]; - ... [51]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: Initialiser[VariableInitialisation*]: INITIALISEROPT; - ... [52]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: [VariableInitialiser ⇒ AttributeCombination] do

```

proc SetupAndEval[TypedIdentifier] (env: ENVIRONMENT): CLASSOPT
  [TypedIdentifier ⇒ Identifier] do return none;
  [TypedIdentifier ⇒ Identifier : TypeExpression] do
    return SetupAndEval[TypeExpression](env)
  end proc;

```

Simple Variable Definition

Syntax

A *SimpleVariableDefinition* represents the subset of *VariableDefinition* expansions that may be used when the variable definition is used as a *Substatement* instead of a *Directive* in non-strict mode. In strict mode variable definitions may not be used as substatements.

SimpleVariableDefinition ⇒ **var** *UntypedVariableBindingList*

UntypedVariableBindingList ⇒
UntypedVariableBinding
 | *UntypedVariableBindingList* , *UntypedVariableBinding*

UntypedVariableBinding ⇒ *Identifier* *VariableInitialisation*^{allowIn}

Validation

```

proc Validate[SimpleVariableDefinition ⇒ var UntypedVariableBindingList] (cxt: CONTEXT, env: ENVIRONMENT)
  if cxt.strict or getRegionalFrame(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](cxt, env)
end proc;

```

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

```

proc Validate[UntypedVariableBinding ⇒ Identifier VariableInitialisationallowIn] (cxt: CONTEXT, env: ENVIRONMENT)
  Validate[VariableInitialisationallowIn](cxt, env);
  defineHoistedVar(env, Name[Identifier], undefined)
end proc;

```

Setup

```

proc Setup[SimpleVariableDefinition ⇒ var UntypedVariableBindingList] ()
  Setup[UntypedVariableBindingList]()
end proc;

```

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

```

proc Setup[UntypedVariableBinding ⇒ Identifier VariableInitialisationallowIn] ()
  Setup[VariableInitialisationallowIn]()
end proc;

```

Evaluation

```

proc Eval[SimpleVariableDefinition  $\Rightarrow$  var UntypedVariableBindingList] (env: ENVIRONMENT, d: OBJECT): OBJECT
  Eval[UntypedVariableBindingList](env);
  return d;
end proc;

proc Eval[UntypedVariableBindingList] (env: ENVIRONMENT)
  [UntypedVariableBindingList  $\Rightarrow$  UntypedVariableBinding] do
    Eval[UntypedVariableBinding](env);
  [UntypedVariableBindingList0  $\Rightarrow$  UntypedVariableBindingList1 , UntypedVariableBinding] do
    Eval[UntypedVariableBindingList1](env);
    Eval[UntypedVariableBinding](env)
  end proc;

proc Eval[UntypedVariableBinding  $\Rightarrow$  Identifier VariableInitialisationallowIn] (env: ENVIRONMENT)
  initializer: INITIALISEROPT  $\leftarrow$  Initializer[VariableInitialisationallowIn];
  if initializer  $\neq$  none then
    value: OBJECT  $\leftarrow$  initializer(env, run);
    qname: QUALIFIEDNAME  $\leftarrow$  public::(Name[Identifier]);
    lexicalWrite(env, {qname}, value, false, run)
  end if
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: ~~initializer: INITIALISEROPT $\leftarrow$ Initializer[VariableInitialisation^{allowIn}]; - ... [53]~~**Function Definition****Syntax**

FunctionDefinition $\Rightarrow$ **function** FunctionName FunctionCommon

FunctionName $\Rightarrow$
 Identifier
 | **get** [no line break] Identifier
 | **set** [no line break] Identifier

FunctionCommon $\Rightarrow$ (Parameters) Result Block

Validation

OverriddenProperty[FunctionDefinition]: INSTANCEPROPERTYOPT;

```

proc ValidateStatic[FunctionDefinition ⇒ function FunctionName FunctionCommon] (cxt: CONTEXT,
  env: ENVIRONMENT, preinst: BOOLEAN, a: COMPOUNDATTRIBUTE, unchecked: BOOLEAN, hoisted: BOOLEAN)
  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 f ← 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, initializer: none⟩;
        defineSingletonProperty(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
        {get} do
          getter: GETTER ← new GETTER⟨call: EvalStaticGet[FunctionCommon], env: boundEnv⟩;
          defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, read, getter);
        {set} do
          setter: SETTER ← new SETTER⟨call: EvalStaticSet[FunctionCommon], env: boundEnv⟩;
          defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, write, setter)
        end case
      end case;
      OverriddenProperty[FunctionDefinition] ← none
    end proc;
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: none;


```

proc ValidateInstance[FunctionDefinition ⇒ function FunctionName FunctionCommon]
  (cxt: 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](cxt, env, instanceFunction, handling);
  signature: PARAMETERFRAME ← CompileFrame[FunctionCommon];
  m: INSTANCEPROPERTY;
  case handling of
    {normal} do
      m ← new INSTANCEMETHOD{final: final, signature: signature, length: signatureLength(signature),
        call: EvalInstanceCall[FunctionCommon]};
      {get} do
        m ← new INSTANCEGETTER{final: final, signature: signature, call: EvalInstanceGet[FunctionCommon]};
      {set} do
        m ← new INSTANCESSETTER{final: final, signature: signature, call: EvalInstanceSet[FunctionCommon]}
      end case;
      mOverridden: INSTANCEPROPERTYOPT ← defineInstanceProperty(c, cxt, Name[FunctionName], a.namespaces,
        a.overrideMod, a.explicit, m);
      enumerable: BOOLEAN ← a.enumerable;
      if mOverridden ≠ none and mOverridden.enumerable then enumerable ← true end if;
      m.enumerable ← enumerable;
      OverriddenProperty[FunctionDefinition] ← mOverridden
    end proc;

proc ValidateConstructor[FunctionDefinition ⇒ function FunctionName FunctionCommon]
  (cxt: 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](cxt, env, constructorFunction, normal);
  if c.init ≠ none then
    throw a DefinitionError exception — duplicate constructor definition
  end if;
  c.init ← EvalInstanceInit[FunctionCommon];
  OverriddenProperty[FunctionDefinition] ← none
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: enumerable: a.enumerable,
signature: CompileFrame[FunctionCommon].

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: enumerable: a.enumerable,
signature: CompileFrame[FunctionCommon].

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: enumerable: a.enumerable,
signature: CompileFrame[FunctionCommon].

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: OverriddenProperty[FunctionDefinition] ←

```

proc Validate[FunctionDefinition ⇒ function FunctionName FunctionCommon]
  (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTE_OPT_NOT_FALSE)
  a: COMPOUND_ATTRIBUTE ← 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.category of
      {static} do
        ValidateStatic[FunctionDefinition](cxt, env, preinst, a, false, false);
      {none} do
        if Name[FunctionName] = frame.name then
          ValidateConstructor[FunctionDefinition](cxt, env, frame, a)
        else ValidateInstance[FunctionDefinition](cxt, env, frame, a, false)
        end if;
      {virtual} do ValidateInstance[FunctionDefinition](cxt, env, frame, a, false);
      {final} do ValidateInstance[FunctionDefinition](cxt, env, frame, a, true)
    end case
  else
    if a.category ≠ none then
      throw an AttributeError exception — non-class functions cannot have a static, virtual, or final
        attribute
    end if;
    unchecked: BOOLEAN ← not cxt.strict and Handling[FunctionName] = normal and Plain[FunctionCommon];
    hoisted: BOOLEAN ← unchecked and attr = none and
      (frame ∈ PACKAGE or (frame ∈ LOCAL_FRAME and env[1] ∈ PARAMETER_FRAME));
    ValidateStatic[FunctionDefinition](cxt, 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]: PARAMETER_FRAME;

```

```

proc Validate[FunctionCommon ⇒ ( Parameters ) Result Block]
  (cxt: CONTEXT, env: ENVIRONMENT, kind: FUNCTIONKIND, handling: HANDLING)
  localCxt: CONTEXT ← new CONTEXT{strict: cxt.strict, openNamespaces: cxt.openNamespaces};
  superconstructorCalled: BOOLEAN ← kind ≠ constructorFunction;
  compileFrame: PARAMETERFRAME ← new PARAMETERFRAME{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, JUMPTARGETS{breakTargets: {}, continueTargets: {}}, false)
end proc;

proc ValidateStaticFunction[FunctionCommon ⇒ ( Parameters ) Result Block]
  (cxt: CONTEXT, env: ENVIRONMENT, kind: STATICFUNCTIONKIND): UNINSTANTIATEDFUNCTION
  Validate[FunctionCommon](cxt, env, kind, normal);
  length: INTEGER ← ParameterCount[Parameters];
  case kind of
    {plainFunction} do
      return new UNINSTANTIATEDFUNCTION{type: Function, length: length,
        call: EvalStaticCall[FunctionCommon], construct: none, instantiations: {}};
    {uncheckedFunction, prototypeFunction} do
      return new UNINSTANTIATEDFUNCTION{type: PrototypeFunction, length: length,
        call: EvalStaticCall[FunctionCommon], construct: EvalPrototypeConstruct[FunctionCommon],
        instantiations: {}}
  end case
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Setup

```

proc Setup[FunctionDefinition ⇒ function FunctionName FunctionCommon] ()
  overriddenProperty: INSTANCEPROPERTYOPT ← OverriddenProperty[FunctionDefinition];
  case overriddenProperty of
    {none} do Setup[FunctionCommon]();
    INSTANCEMETHOD ∪ INSTANCEGETTER ∪ INSTANCESSETTER do
      SetupOverride[FunctionCommon](overriddenProperty.signature);
    INSTANCEVARIABLE do
      overriddenSignature: PARAMETERFRAME;
      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: overriddenProperty.type};
        {set} do
          v: VARIABLE ← new VARIABLE{type: overriddenProperty.type, value: none, immutable: false,
            setup: none, initializer: none};
          parameters: PARAMETER[] ← [PARAMETER{var: v, default: none}];
          overriddenSignature ← new PARAMETERFRAME{localBindings: {}, kind: instanceFunction,
            handling: set, callsSuperconstructor: false, superconstructorCalled: false, this: none,
            parameters: parameters, rest: none, returnType: Void}
      end case;
      SetupOverride[FunctionCommon](overriddenSignature)
    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;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initialiser: none;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Evaluation

```

proc EvalStaticCall[FunctionCommon  $\Rightarrow$  ( Parameters ) Result Block]
  (this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note The check that phase  $\neq$  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  $\leftarrow$  f.env;
  runtimeThis: OBJECTOPT  $\leftarrow$  none;
  compileFrame: PARAMETERFRAME  $\leftarrow$  CompileFrame[FunctionCommon];
  if compileFrame.kind  $\in$  {uncheckedFunction, prototypeFunction} then
    if this  $\in$  PRIMITIVEOBJECT then runtimeThis  $\leftarrow$  getPackageFrame(runtimeEnv)
    else runtimeThis  $\leftarrow$  this
    end if
  end if;
  runtimeFrame: PARAMETERFRAME  $\leftarrow$  instantiateParameterFrame(compileFrame, runtimeEnv, runtimeThis);
  assignArguments(runtimeFrame, f, args, phase);
  result: OBJECT;
  try Eval[Block]([runtimeFrame]  $\oplus$  runtimeEnv, undefined); result  $\leftarrow$  undefined
  catch x: SEMANTICEXCEPTION do
    if x  $\in$  RETURN then result  $\leftarrow$  x.value else throw x end if
  end try;
  coercedResult: OBJECT  $\leftarrow$  as(result, runtimeFrame.returnType, false);
  return coercedResult
end proc;

proc EvalStaticGet[FunctionCommon  $\Rightarrow$  ( Parameters ) Result Block]
  (runtimeEnv: ENVIRONMENT, phase: PHASE): OBJECT
  note The check that phase  $\neq$  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  $\leftarrow$  CompileFrame[FunctionCommon];
  runtimeFrame: PARAMETERFRAME  $\leftarrow$  instantiateParameterFrame(compileFrame, runtimeEnv, none);
  assignArguments(runtimeFrame, none, [], phase);
  result: OBJECT;
  try
    Eval[Block]([runtimeFrame]  $\oplus$  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  $\in$  RETURN then result  $\leftarrow$  x.value else throw x end if
  end try;
  coercedResult: OBJECT  $\leftarrow$  as(result, runtimeFrame.returnType, false);
  return coercedResult
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: coercedResult: OBJECT $\leftarrow$ implicitCoerce(result, runtimeFrame.returnType, false);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: coercedResult: OBJECT $\leftarrow$ implicitCoerce(result, runtimeFrame.returnType, false);

```

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 ← as(result, runtimeFrame.returnType, 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 ← as(result, runtimeFrame.returnType, false);
  return coercedResult
end proc;

```

```

proc EvalInstanceSet[FunctionCommon  $\Rightarrow$  ( Parameters ) Result Block]
  (this: OBJECT, newValue: OBJECT, phase: PHASE)
  note The check that phase  $\neq$  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  $\leftarrow$  CompileEnv[FunctionCommon];
  compileFrame: PARAMETERFRAME  $\leftarrow$  CompileFrame[FunctionCommon];
  runtimeFrame: PARAMETERFRAME  $\leftarrow$  instantiateParameterFrame(compileFrame, env, this);
  assignArguments(runtimeFrame, none, [newValue], phase);
  try Eval[Block]([runtimeFrame]  $\oplus$  env, undefined)
  catch x: SEMANTICEXCEPTION do if x  $\notin$  RETURN then throw x end if
  end try
end proc;

proc EvalInstanceInit[FunctionCommon  $\Rightarrow$  ( 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  $\leftarrow$  CompileEnv[FunctionCommon];
  compileFrame: PARAMETERFRAME  $\leftarrow$  CompileFrame[FunctionCommon];
  runtimeFrame: PARAMETERFRAME  $\leftarrow$  instantiateParameterFrame(compileFrame, env, this);
  assignArguments(runtimeFrame, none, args, phase);
  if not runtimeFrame.callsSuperconstructor then
    c: CLASS  $\leftarrow$  getEnclosingClass(env);
    callInit(this, c.super, [], run);
    runtimeFrame.superconstructorCalled  $\leftarrow$  true
  end if;
  try Eval[Block]([runtimeFrame]  $\oplus$  env, undefined)
  catch x: SEMANTICEXCEPTION do if x  $\notin$  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;
  archetype: OBJECT ← dotRead(f, {public::"prototype"}, phase);
  if archetype ∈ {null, undefined} then archetype ← ObjectPrototype
  elseif objectType(archetype) ≠ Object then
    throw a TypeError exception — bad prototype value
  end if;
  o: OBJECT ← createSimpleInstance(Object, archetype, 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 ← undefined
  catch x: SEMANTICEXCEPTION do
    if x ∈ RETURN then result ← x.value else throw x end if
  end try;
  coercedResult: OBJECT ← as(result, runtimeFrame.returnType, false);
  if coercedResult ∈ 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 ≠ [] or rest ≠ none then
        throw a SyntaxError exception — a getter cannot take any parameters
      end if;
    {set} do
      if |parameters| ≠ 1 or rest ≠ none then
        throw a SyntaxError exception — a setter must take exactly one parameter
      end if;
      if parameters[0].default ≠ none then
        throw a SyntaxError exception — a setter's parameter cannot be optional
      end if
    end case
  end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: OBJECT ← dotRead(f, {public::"prototype"}, phase); - ... [54]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: o: OBJECT ← createSimpleInstance(Object, super, none, none, none);

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: coercedResult: OBJECT ← implicitCoerce(result, runtimeFrame.returnType, false);


```

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;
  dotWrite(argumentsObject, {arrayPrivate::"length"}, nArgsulong, 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;
    writeSingletonProperty(v, arg, phase);
    if argumentsObject ≠ none then
      note Create an alias of v as the ith entry of the arguments object.
      note v ∈ DYNAMICVAR;
      qname: QUALIFIEDNAME ← objectToQualifiedName(iulong, phase);
      argumentsObject.localBindings ← argumentsObject.localBindings ∪ {LOCALBINDING{qname: qname,
        accesses: readWrite, explicit: false, enumerable: false, content: v}}
    end if
  elseif 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
  elseif 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;
  writeSingletonProperty(parameter.var, default, phase);

```

```

    i ← i + 1
  end while
end proc;

```

```

proc signatureLength(signature: PARAMETERFRAME): INTEGER
  return |signature.parameters|
end proc;

```

Syntax

```

Parameters ⇒
  «empty»
  | NonemptyParameters

NonemptyParameters ⇒
  ParameterInit
  | ParameterInit , NonemptyParameters
  | RestParameter

Parameter ⇒ ParameterAttributes TypedIdentifierallowIn

ParameterAttributes ⇒
  «empty»
  | const

ParameterInit ⇒
  Parameter
  | Parameter = AssignmentExpressionallowIn

RestParameter ⇒
  ...
  | ... ParameterAttributes Identifier

Result ⇒
  «empty»
  | : TypeExpressionallowIn

```

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] (ctx: 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[NonemptyParameters0 ⇒ ParameterInit , NonemptyParameters1]
  = Plain[ParameterInit] and Plain[NonemptyParameters1];
Plain[NonemptyParameters ⇒ RestParameter] = false;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

```

ParameterCount[NonemptyParameters]: INTEGER;
ParameterCount[NonemptyParameters ⇒ ParameterInit] = 1;
ParameterCount[NonemptyParameters0 ⇒ ParameterInit , NonemptyParameters1]
    = 1 + ParameterCount[NonemptyParameters1];
ParameterCount[NonemptyParameters ⇒ RestParameter] = 0;

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

Name[Parameter ⇒ ParameterAttributes TypedIdentifierallowIn]: STRING = Name[TypedIdentifierallowIn];

Plain[Parameter ⇒ ParameterAttributes TypedIdentifierallowIn]: BOOLEAN
    = Plain[TypedIdentifierallowIn] and not HasConst[ParameterAttributes];

CompileVar[Parameter]: DYNAMICVAR ∪ VARIABLE;

proc Validate[Parameter ⇒ ParameterAttributes TypedIdentifierallowIn]
    (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME ∪ LOCALFRAME)
    Validate[TypedIdentifierallowIn](cxt, env);
    immutable: BOOLEAN ← HasConst[ParameterAttributes];
    name: STRING ← Name[TypedIdentifierallowIn];
    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, initializer: none);
        defineSingletonProperty(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 = AssignmentExpressionallowIn] = false;

proc Validate[ParameterInit] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [ParameterInit ⇒ Parameter] do Validate[Parameter](cxt, env, compileFrame);
    [ParameterInit ⇒ Parameter = AssignmentExpressionallowIn] do
        Validate[Parameter](cxt, env, compileFrame);
        Validate[AssignmentExpressionallowIn](cxt, env)
    end proc;

proc Validate[RestParameter] (cxt: CONTEXT, env: ENVIRONMENT, compileFrame: PARAMETERFRAME)
    [RestParameter ⇒ ...] do
        note compileFrame.kind ≠ uncheckedFunction;
        v: VARIABLE ← new VARIABLE(type: Array, value: none, immutable: true, setup: none, initializer: none);
        compileFrame.rest ← v;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: **new** VARIABLE(value: none, immutable: immutable, setup: none, initialiser: none);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initialiser: none);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

```
[RestParameter ⇒ ... ParameterAttributes Identifier] do
  note compileFrame.kind ≠ uncheckedFunction;
  v: VARIABLE ← new VARIABLE⟨type: Array, value: none, immutable: HasConst[ParameterAttributes],
    setup: none, initializer: none⟩;
  compileFrame.rest ← v;
  name: STRING ← Name[Identifier];
  defineSingletonProperty(env, name, {public}, none, false, readWrite, v)
end proc;

Plain[Result]: BOOLEAN;
Plain[Result ⇒ «empty»] = true;
Plain[Result ⇒ : TypeExpressionallowIn] = false;

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

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initialiser: none;

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 , NonemptyParameters1] do
    Setup[ParameterInit](compileEnv, compileFrame);
    Setup[NonemptyParameters1](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 TypedIdentifierallowIn]
  (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[TypedIdentifierallowIn](compileEnv);
      if type = none then type ← Object end if;
      v.type ← type
    end case;
  if compileFrame ∈ PARAMETERFRAME then
    p: PARAMETER ← PARAMETER{var: v, default: default};
    compileFrame.parameters ← compileFrame.parameters ⊕ [p]
  end if
end proc;

proc SetupOverride[Parameter ⇒ ParameterAttributes TypedIdentifierallowIn] (compileEnv: ENVIRONMENT,
  compileFrame: PARAMETERFRAME, default: OBJECTOPT, overriddenParameter: PARAMETER)
  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[TypedIdentifierallowIn](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;

```

Class Definition

Syntax

```

ClassDefinition ⇒ class Identifier Inheritance Block

Inheritance ⇒
  «empty»
  | extends TypeExpressionallowIn

```

Validation

```

Class[ClassDefinition]: CLASS;

```

```

proc Validate[ClassDefinition ⇒ class Identifier Inheritance Block]
  (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTE_OPT_NOT_FALSE)
  if not preinst then
    throw a SyntaxError exception — a class may be defined only in a preinstantiated scope
  end if;
  super: CLASS ← Validate[Inheritance](cxt, 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.category of
    {none} do final ← false;
    {static} do
      if env[0] ∉ CLASS then
        throw an AttributeError exception — non-class property 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 and super ≠ Object);
  c: CLASS ← new CLASS(localBindings: {}, instanceProperties: {}, super: super, prototype: super.prototype,
    complete: false, name: Name[Identifier], typeofString: "object", privateNamespace: privateNamespace,
    dynamic: dynamic, final: final, defaultValue: null, defaultHint: hintNumber,
    bracketRead: super.bracketRead, bracketWrite: super.bracketWrite, bracketDelete: super.bracketDelete,
    read: super.read, write: super.write, delete: super.delete, enumerate: super.enumerate, init: none,
    is: ordinaryIs, as: ordinaryAs);
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| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  return as(args[0], c, 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

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: super,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return implicitCoerce(args[0], c, false)

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

|

```
end proc;
c.construct ← cConstruct;
Class[ClassDefinition] ← c;
v: VARIABLE ← new VARIABLE(type: Class, value: c, immutable: true, setup: none, initializer: none);
defineSingletonProperty(env, Name[Identifier], a.namespaces, a.overrideMod, a.explicit, readWrite, v);
innerCxt: CONTEXT ← new CONTEXT(strict: cxt.strict,
    openNamespaces: cxt.openNamespaces ∪ {privateNamespace});
ValidateUsingFrame[Block](innerCxt, env, JUMPTARGETS{breakTargets: {}, continueTargets: {}}, preinst, c);
if c.init = none then c.init ← super.init end if;
c.complete ← true
end proc;

proc Validate[Inheritance] (cxt: CONTEXT, env: ENVIRONMENT): CLASS
[Inheritance ⇒ «empty»] do return Object;
[Inheritance ⇒ extends TypeExpressionallowIn] do
    Validate[TypeExpressionallowIn](cxt, env);
    return SetupAndEval[TypeExpressionallowIn](env)
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: initialiser: none;

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;
```

Namespace Definition

Syntax

NamespaceDefinition ⇒ namespace Identifier

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Validation

```
proc Validate[NamespaceDefinition ⇒ namespace Identifier]  
  (cxt: CONTEXT, env: ENVIRONMENT, preinst: BOOLEAN, attr: ATTRIBUTE_OPT_NOT_FALSE)  
  if not preinst then  
    throw a SyntaxError exception — a namespace may be defined only in a preinstantiated scope  
  end if;  
  a: COMPOUND_ATTRIBUTE ← 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.category of  
    {none} do nothing;  
    {static} do  
      if env[0] ∉ CLASS then  
        throw an AttributeError exception — non-class property definitions cannot have a static attribute  
      end if;  
    {virtual, final} do  
      throw an AttributeError exception — a namespace definition cannot have the virtual or final attribute  
    end case;  
  name: STRING ← Name[Identifier];  
  ns: NAMESPACE ← new NAMESPACE(name: name);  
  v: VARIABLE ← new VARIABLE(type: Namespace, value: ns, immutable: true, setup: none, initializer: none);  
  defineSingletonProperty(env, name, a.namespaces, a.overrideMod, a.explicit, readWrite, v)  
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: initialiser: none);

Programs

Syntax

```
Program ⇒  
  Directives  
  PackageDefinition Program
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: Directives

Processing

```
Process[Program]: OBJECT;  
Process[Program ⇒ Directives]  
  begin  
    cxt: CONTEXT ← new CONTEXT(strict: false, openNamespaces: {public, internal});  
    initialEnvironment: ENVIRONMENT ← [createGlobalObject()];  
    Validate[Directives](cxt, initialEnvironment, JUMP_TARGETS(breakTargets: {}, continueTargets: {}), true,  
      none);  
    Setup[Directives]();  
    return Eval[Directives](initialEnvironment, undefined)  
  end;  
Process[Program0 ⇒ PackageDefinition Program1]  
  begin  
    Process[PackageDefinition];  
    return Process[Program1]  
  end;
```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: Evaluation [55]

Package Definition

Syntax

PackageDefinition $\Rightarrow$ **package** *PackageNameOpt* *Block*

PackageNameOpt $\Rightarrow$

«empty»

| *PackageName*

PackageName $\Rightarrow$

String

| *PackageIdentifiers*

PackageIdentifiers $\Rightarrow$

Identifier

| *PackageIdentifiers* . *Identifier*

Processing

Process[*PackageDefinition* $\Rightarrow$ **package** *PackageNameOpt* *Block*]: VOID

begin

name: STRING $\leftarrow$ **Name**[*PackageNameOpt*];

cxt: CONTEXT $\leftarrow$ **new** CONTEXT{strict: **false**, openNamespaces: {*public*, *internal*}};

globalObject: PACKAGE $\leftarrow$ *createGlobalObject*();

pkgInternal: NAMESPACE $\leftarrow$ **new** NAMESPACE{name: “internal”};

pkg: PACKAGE $\leftarrow$ **new** PACKAGE{localBindings:

{*stdExplicitConstBinding*(*internal*: “internal”, *Namespace*, *internal*)}, *archetype*: *ObjectPrototype*,

name: *name*, *initialize*: **busy**, *sealed*: **true**, *internalNamespace*: *pkgInternal*};

initialEnvironment: ENVIRONMENT $\leftarrow$ [*pkg*, *globalObject*];

Validate[*Block*](*cxt*, *initialEnvironment*, JUMPTARGETS{*breakTargets*: {}, *continueTargets*: {}}, **true**);

Setup[*Block*]();

proc *evalPackage*()

pkg.initialize $\leftarrow$ **busy**;

Eval[*Block*](*initialEnvironment*, **undefined**);

pkg.initialize $\leftarrow$ **none**

end proc;

pkg.initialize $\leftarrow$ *evalPackage*;

Bind *name* to package *pkg* in the system’s list of packages in an implementation-defined manner.

end;

Name[*PackageNameOpt*]: STRING;

Name[*PackageNameOpt* $\Rightarrow$ «empty»] = an implementation-supplied name;

Name[*PackageNameOpt* $\Rightarrow$ *PackageName*] = **Name**[*PackageName*];

Name[*PackageName*]: STRING;

Name[*PackageName* $\Rightarrow$ **String**] = **Value**[**String**] processed in an implementation-defined manner;

Name[*PackageName* $\Rightarrow$ *PackageIdentifiers*] = **Names**[*PackageIdentifiers*] processed in an implementation-defined manner;

Names[*PackageIdentifiers*]: STRING[];

Names[*PackageIdentifiers* $\Rightarrow$ *Identifier*] = [**Name**[*Identifier*]];

Names[*PackageIdentifiers*₀ $\Rightarrow$ *PackageIdentifiers*₁ . *Identifier*] = **Names**[*PackageIdentifiers*₁] $\oplus$ [**Name**[*Identifier*]];

packageDatabase: PACKAGE {} $\leftarrow$ {};

Predefined Identifiers

```

proc createGlobalObject(): PACKAGE
  return new PACKAGE(localBindings: {
    stdExplicitConstBinding(internal::"internal", Namespace, internal),
    stdConstBinding(public::"explicit", Attribute, global_explicit),
    stdConstBinding(public::"enumerable", Attribute, global_enumerable),
    stdConstBinding(public::"dynamic", Attribute, global_dynamic),
    stdConstBinding(public::"static", Attribute, global_static),
    stdConstBinding(public::"virtual", Attribute, global_virtual),
    stdConstBinding(public::"final", Attribute, global_final),
    stdConstBinding(public::"prototype", Attribute, global_prototype),
    stdConstBinding(public::"unused", Attribute, global_unused),
    stdFunction(public::"override", global_override, 1),
    stdConstBinding(public::"Object", Class, Object),
    stdConstBinding(public::"Never", Class, Never),
    stdConstBinding(public::"Void", Class, Void),
    stdConstBinding(public::"Null", Class, Null),
    stdConstBinding(public::"Boolean", Class, Boolean),
    stdConstBinding(public::"GeneralNumber", Class, GeneralNumber),
    stdConstBinding(public::"long", Class, long),
    stdConstBinding(public::"ulong", Class, ulong),
    stdConstBinding(public::"float", Class, float),
    stdConstBinding(public::"Number", Class, Number),
    stdConstBinding(public::"sbyte", Class, sbyte),
    stdConstBinding(public::"byte", Class, byte),
    stdConstBinding(public::"short", Class, short),
    stdConstBinding(public::"ushort", Class, ushort),
    stdConstBinding(public::"int", Class, int),
    stdConstBinding(public::"uint", Class, uint),
    stdConstBinding(public::"Character", Class, Character),
    stdConstBinding(public::"String", Class, String),
    stdConstBinding(public::"Array", Class, Array),
    stdConstBinding(public::"Namespace", Class, Namespace),
    stdConstBinding(public::"Attribute", Class, Attribute),
    stdConstBinding(public::"Date", Class, Date),
    stdConstBinding(public::"RegExp", Class, RegExp),
    stdConstBinding(public::"Class", Class, Class),
    stdConstBinding(public::"Function", Class, Function),
    stdConstBinding(public::"PrototypeFunction", Class, PrototypeFunction),
    stdConstBinding(public::"Package", Class, Package),
    stdConstBinding(public::"Error", Class, Error),
    stdConstBinding(public::"ArgumentError", Class, ArgumentError),
    stdConstBinding(public::"AttributeError", Class, AttributeError),
    stdConstBinding(public::"ConstantError", Class, ConstantError),
    stdConstBinding(public::"DefinitionError", Class, DefinitionError),
    stdConstBinding(public::"EvalError", Class, EvalError),
    stdConstBinding(public::"RangeError", Class, RangeError),
    stdConstBinding(public::"ReferenceError", Class, ReferenceError),
    stdConstBinding(public::"SyntaxError", Class, SyntaxError),
    stdConstBinding(public::"TypeError", Class, TypeError),
    stdConstBinding(public::"UninitializedError", Class, UninitializedError),
    stdConstBinding(public::"URIError", Class, URIError),
    archetype: ObjectPrototype, name: "", initialise: none, sealed: false, internalNamespace: internal)
  }
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: {**LOCALBINDING**(qname: internal::"internal", accesses: **readWrite**, explicit: **true**, enumerable: **false**, content: new **VARIABLE**(type: Namespace, value: internal, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"explicit", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: explicitAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"enumerable", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: enumerableAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"dynamic", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: dynamicAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"static", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: staticAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"virtual", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: virtualAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"final", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: finalAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"prototype", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: prototypeAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"unused", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Attribute, value: unusedAttribute, immutable: **true**, setup: **none**, initialiser: **none**)), **LOCALBINDING**(qname: public::"override", accesses: **readWrite**, explicit: **false**, enumerable: **false**, content: new **VARIABLE**(type: Function, value: overrideAttribute, immutable: ... [56])

Built-in Namespaces

```
public: NAMESPACE = new NAMESPACE{name: "public"};

internal: NAMESPACE = new NAMESPACE{name: "internal"};
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Built-in Attributes

```
global explicit: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: true, enumerable: false,
dynamic: false, category: none, overrideMod: none, prototype: false, unused: false};

global enumerable: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false,
enumerable: true, dynamic: false, category: none, overrideMod: none, prototype: false, unused: false};

global dynamic: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false,
enumerable: false, dynamic: true, category: none, overrideMod: none, prototype: false, unused: false};

global static: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false,
dynamic: false, category: static, overrideMod: none, prototype: false, unused: false};

global virtual: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false,
dynamic: false, category: virtual, overrideMod: none, prototype: false, unused: false};

global final: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false,
dynamic: false, category: final, overrideMod: none, prototype: false, unused: false};

global prototype: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false,
enumerable: false, dynamic: false, category: none, overrideMod: none, prototype: true, unused: false};

global unused: COMPOUNDATTRIBUTE = COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false,
dynamic: false, category: none, overrideMod: none, prototype: false, unused: true};

proc global override(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function does not check phase and therefore can be used in constant expressions.
overrideMod: OVERRIDEMODIFIER;
if args = [] then overrideMod ← true
elseif |args| = 1 then
    arg: OBJECT ← args[0];
    if arg ∉ {true, false, undefined} then throw a TypeError exception end if;
    overrideMod ← arg
else throw an ArgumentError exception — too many arguments supplied
end if;
return COMPOUNDATTRIBUTE{namespaces: {}, explicit: false, enumerable: false, dynamic: false,
category: none, overrideMod: overrideMod, prototype: false, unused: false}
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: explicitAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: enumerableAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: dynamicAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: staticAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: virtualAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: finalAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: prototypeAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: unusedAttribute: COMPOUNDATTRIBUTE

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: overrideAttribute: SIMPLEINSTANCE = system
Function(callOverrideAttribute, 1); ... [57]

Built-in Functions

Built-in Classes

```
proc dummyCall(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
???
end proc;
```

```

proc dummyConstruct(args: OBJECT[], phase: PHASE): OBJECT
  ???
end proc;

prototypesSealed: BOOLEAN = false;

```

Object

```

Object: CLASS = new CLASS{localBindings: {}, instanceProperties: {}, super: none, prototype: ObjectPrototype,
  complete: true, name: "Object", typeofString: "object", dynamic: true, final: false,
  defaultValue: undefined, defaultHint: hintNumber, bracketRead: ordinaryBracketRead,
  bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
  write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callObject,
  construct: constructObject, init: none, is: ordinaryIs, as: asObject};

```

```

proc callObject(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in constant expressions.
  if args > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return defaultArg(args, 0, undefined)
end proc;

```

```

proc constructObject(args: OBJECT[], phase: PHASE): OBJECT
  return callObject(null, args, phase)
end proc;

```

```

proc asObject(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
  return o
end proc;

```

```

ObjectPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {
  stdConstBinding(public::"constructor", Class, Object),
  stdFunction(public::"toString", Object toString, 0),
  stdFunction(public::"toLocaleString", Object toLocaleString, 0),
  stdFunction(public::"valueOf", Object valueOf, 0),
  stdFunction(public::"hasOwnProperty", Object hasOwnProperty, 1),
  stdFunction(public::"isPrototypeOf", Object isPrototypeOf, 1),
  stdFunction(public::"propertyIsEnumerable", Object propertyIsEnumerable, 1),
  stdFunction(public::"sealProperty", Object sealProperty, 1)},
  archetype: none, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none, env: none};

```

```

proc Object toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in constant expressions.
  note This function ignores any arguments passed to it in args.
  c: CLASS ← objectType(this);
  return "[object " ⊕ c.name ⊕ "]"
end proc;

```

```

proc Object toLocaleString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  if phase = compile then
    throw a ConstantError exception — toLocaleString cannot be called from constant expressions
  end if;
  toStringMethod: OBJECT ← dotRead(this, {public::"toString"}, phase);
  return call(this, toStringMethod, args, phase)
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *super*: none,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** SIMPLEINSTANCE{*localBindings*: {}, *super*: none,

```

proc Object valueOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in constant expressions.
  note This function ignores any arguments passed to it in args.
  return this
end proc;

proc Object hasOwnProperty(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  if phase = compile then
    throw a ConstantError exception — hasOwnProperty cannot be called from constant expressions
  end if;
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  qname: QUALIFIEDNAME ← objectToQualifiedName(args[0], phase);
  return hasProperty(this, qname, true)
end proc;

proc Object isPrototypeOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  if phase = compile then
    throw a ConstantError exception — isPrototypeOf cannot be called from constant expressions
  end if;
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  o: OBJECT ← args[0];
  return this ∈ archetypes(o)
end proc;

proc Object propertyIsEnumerable(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  if phase = compile then
    throw a ConstantError exception — propertyIsEnumerable cannot be called from constant expressions
  end if;
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  qname: QUALIFIEDNAME ← objectToQualifiedName(args[0], phase);
  c: CLASS ← objectType(this);
  mBase: INSTANCEPROPERTYOPT ← findBaseInstanceProperty(c, {qname}, read);
  if mBase ≠ none then
    m: INSTANCEPROPERTY ← getDerivedInstanceProperty(c, mBase, read);
    if m.enumerable then return true end if
  end if;
  mBase ← findBaseInstanceProperty(c, {qname}, write);
  if mBase ≠ none then
    m: INSTANCEPROPERTY ← getDerivedInstanceProperty(c, mBase, write);
    if m.enumerable then return true end if
  end if;
  if this ∉ BINDINGOBJECT then return false end if;
  return some b ∈ this.localBindings satisfies b.qname = qname and b.enumerable
end proc;

```

```

proc Object sealProperty(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  if phase = compile then
    throw a ConstantError exception — sealProperty cannot be called from constant expressions
  end if;
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT ← defaultArg(args, 0, true);
  if arg = false then sealObject(this)
  elseif arg = true then sealObject(this); sealAllLocalProperties(this)
  elseif arg ∈ CHAR16 ∪ STRING then
    qname: QUALIFIEDNAME ← objectToQualifiedName(arg, phase);
    if not hasProperty(this, qname, true) then
      throw a ReferenceError exception — property not found
    end if;
    sealLocalProperty(this, qname)
  end if;
  return undefined
end proc;

```

Never

```

Never: CLASS = new CLASS(localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
  complete: true, name: "Never", typeofString: "", dynamic: false, final: true, defaultValue: none,
  bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: callNever, construct: constructNever, init: none, is: ordinaryIs,
  as: asNever);

```

```

proc callNever(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  throw a TypeError exception — no coercions to Never are possible
end proc;

```

```

proc constructNever(args: OBJECT[], phase: PHASE): OBJECT
  return callNever(null, args, phase)
end proc;

```

```

proc asNever(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
  throw a TypeError exception — no coercions to Never are possible
end proc;

```

Void

```

Void: CLASS = new CLASS(localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
  complete: true, name: "Void", typeofString: "undefined", dynamic: false, final: true,
  defaultValue: undefined, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: callVoid, construct: constructVoid, init: none, is: ordinaryIs, as: asVoid);

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *super*: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *implicitCoerce*: *ordinaryImplicitC*
oerceNoNull);

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *super*: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *implicitCoerce*: *ordinaryImplicitC*
oerceNoNull);

```
proc callVoid(this: OBJECT, args: OBJECT[], phase: PHASE): UNDEFINED
  note This function does not check phase and therefore can be used in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return undefined
end proc;

proc constructVoid(args: OBJECT[], phase: PHASE): UNDEFINED
  note This function does not check phase and therefore can be used in constant expressions.
  if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
  end if;
  return undefined
end proc;

proc asVoid(o: OBJECT, c: CLASS, silent: BOOLEAN): UNDEFINED
  if o ∈ NULL ∪ UNDEFINED then return undefined else throw a TypeError exception end if
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Null

```
Null: CLASS = new CLASS(localBindings: {}, instanceProperties: {}, super: Object, prototype: none,
  complete: true, name: "Null", typeofString: "object", dynamic: false, final: true, defaultValue: null,
  bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: callNull, construct: constructNull, init: none, is: ordinaryIs, as: asNull);

proc callNull(this: OBJECT, args: OBJECT[], phase: PHASE): NULL
  note This function does not check phase and therefore can be used in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return null
end proc;

proc constructNull(args: OBJECT[], phase: PHASE): NULL
  note This function does not check phase and therefore can be used in constant expressions.
  if |args| ≠ 0 then throw an ArgumentError exception — no arguments can be supplied
  end if;
  return null
end proc;

proc asNull(o: OBJECT, c: CLASS, silent: BOOLEAN): NULL
  if o = null then return o else throw a TypeError exception end if
end proc;
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: Object,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: call: dummyCall,
construct: dummyConstruct,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: ordinaryImplicitCoerceNull;

Boolean

```
Boolean: CLASS = new CLASS(localBindings: {}, instanceProperties: {}, super: Object, prototype: BooleanPrototype,
  complete: true, name: "Boolean", typeofString: "boolean", dynamic: false, final: true,
  defaultValue: false, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: callBoolean, construct: constructBoolean, init: none, is: ordinaryIs,
  as: asBoolean);
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: Object,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: call: dummyCall,
construct: dummyConstruct,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: ordinaryImplicitCoerceNonNull;


```

proc callBoolean(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  note This function does not check phase and therefore can be used in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return objectToBoolean(defaultArg(args, 0, false))
end proc;

```

```

proc constructBoolean(args: OBJECT[], phase: PHASE): OBJECT
  return callBoolean(null, args, phase)
end proc;

```

```

proc asBoolean(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
  if o ∈ BOOLEAN then return o else throw a TypeError exception end if
end proc;

```

```

BooleanPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE(localBindings: {
  stdConstBinding(public::"constructor", Class, Boolean),
  stdFunction(public::"toString", Boolean, toString, 0)},
archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
env: none);

```

```

proc Boolean toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions.
  note This function ignores any arguments passed to it in args.
  a: BOOLEAN ← objectToBoolean(this);
  return objectToString(a, phase)
end proc;

```

GeneralNumber

```

GeneralNumber: CLASS = new CLASS(localBindings: {}, instanceProperties: {}, super: Object,
  prototype: GeneralNumberPrototype, complete: true, name: "GeneralNumber", typeofString: "object",
  dynamic: false, final: false, defaultValue: NaN164, defaultHint: hintNumber,
  bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
  bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
  enumerate: ordinaryEnumerate, call: callGeneralNumber, construct: constructGeneralNumber, init: none,
  is: ordinaryIs, as: asGeneralNumber);

```

```

proc callGeneralNumber(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT ← defaultArg(args, 0, +zero164);
  return objectToGeneralNumber(arg, phase)
end proc;

```

```

proc constructGeneralNumber(args: OBJECT[], phase: PHASE): OBJECT
  return callGeneralNumber(null, args, phase)
end proc;

```

```

proc asGeneralNumber(o: OBJECT, c: CLASS, silent: BOOLEAN): GENERALNUMBER
  if o ∈ GENERALNUMBER then return o else throw a TypeError exception end if
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** **SIMPLEINSTANCE**(*localBindings*: {}, *super*: *ObjectPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *super*: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNonNull*;

GeneralNumberPrototype: **SIMPLEINSTANCE** = **new** SIMPLEINSTANCE{localBindings: {
 stdConstBinding(public::"constructor", Class, GeneralNumber),
 stdFunction(public::"toString", GeneralNumber toString, 1),
 stdFunction(public::"toFixed", GeneralNumber toFixed, 1),
 stdFunction(public::"toExponential", GeneralNumber toExponential, 1),
 stdFunction(public::"toPrecision", GeneralNumber toPrecision, 1)},
 archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**,
 env: **none**};

proc *GeneralNumber toString*(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function can be used in constant expressions if *this* and the argument can be converted to primitives in constant expressions.
note This function is generic and can be applied even if *this* is not a general number.
x: GENERALNUMBER ← *objectToGeneralNumber*(this, phase);
radix: EXTENDEDINTEGER ← *objectToImpreciseInteger*(defaultArg(args, 0, 10.0₁₆₄), phase);
if *radix* = NaN **then** *radix* ← 10 **end if**;
if *radix* ∈ {+∞, -∞} **or** *radix* < 2 **or** *radix* > 36 **then**
throw a *RangeError* exception — bad radix
end if;
if *radix* = 10 **then** **return** *generalNumberToString*(*x*)
else
return *x* converted to a string containing a base-*radix* number in an implementation-defined manner
end if
end proc;

precisionLimit: INTEGER = an implementation-defined integer not less than 20;

proc *GeneralNumber toFixed*(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
note This function can be used in constant expressions if *this* and the argument can be converted to primitives in constant expressions.
note This function is generic and can be applied even if *this* is not a general number.
if |args| > 1 **then**
throw an *ArgumentError* exception — at most one argument can be supplied
end if;
x: GENERALNUMBER ← *objectToGeneralNumber*(this, phase);
fractionDigits: EXTENDEDINTEGER ← *objectToImpreciseInteger*(defaultArg(args, 0, +zero₁₆₄), phase);
if *fractionDigits* = NaN **then** *fractionDigits* ← 0 **end if**;
if *fractionDigits* ∈ {+∞, -∞} **or** *fractionDigits* < 0 **or** *fractionDigits* > *precisionLimit* **then**
throw a *RangeError* exception
end if;
if *x* ∉ FINITEGENERALNUMBER **then** **return** *generalNumberToString*(*x*) **end if**;
r: RATIONAL ← *toRational*(*x*);
if |*r*| ≥ 10²¹ **then** **return** *generalNumberToString*(*x*) **end if**;
sign: STRING ← "";
if *r* < 0 **then** *sign* ← "-"; *r* ← -*r* **end if**;
n: INTEGER ← ⌊*r* × 10^{*fractionDigits*} + 1/2⌋;
digits: STRING ← *integerToString*(*n*);
if *fractionDigits* > 0 **then**
if |*digits*| ≤ *fractionDigits* **then**
digits ← repeat('0', *fractionDigits* + 1 - |*digits*|) ⊕ *digits*
end if;
k: INTEGER ← |*digits*| - *fractionDigits*;
digits ← *digits*[0 ... *k* - 1] ⊕ "." ⊕ *digits*[*k* ...]
end if;
return *sign* ⊕ *digits*
end proc;

```

proc GeneralNumber toExponential(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the argument can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a general number.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
  fractionDigits: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +zero164), phase);
  if fractionDigits = NaN then ??? end if;
  if fractionDigits ∈ {+∞, -∞} or fractionDigits < 0 or fractionDigits > precisionLimit then
    throw a RangeError exception
  end if;
  if x ∉ FINITEGENERALNUMBER then return generalNumberToString(x) end if;
  r: RATIONAL ← toRational(x);
  ???
end proc;

```

```

proc GeneralNumber toPrecision(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the argument can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a general number.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  x: GENERALNUMBER ← objectToGeneralNumber(this, phase);
  fractionDigits: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +zero164), phase);
  if fractionDigits = NaN then ??? end if;
  if fractionDigits ∈ {+∞, -∞} or fractionDigits < 0 or fractionDigits > precisionLimit then
    throw a RangeError exception
  end if;
  if x ∉ FINITEGENERALNUMBER then return generalNumberToString(x) end if;
  r: RATIONAL ← toRational(x);
  ???
end proc;

```

long

```

long: CLASS = new CLASS(localBindings: {
  stdConstBinding(public::"MAX_VALUE", ulong, (263 - 1)long),
  stdConstBinding(public::"MIN_VALUE", ulong, (-263)long),
  instanceProperties: {}, super: GeneralNumber, prototype: longPrototype, complete: true, name: "long",
  typeofString: "long", dynamic: false, final: true, defaultValue: 0long, bracketRead: ordinaryBracketRead,
  bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
  write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callLong,
  construct: constructLong, init: none, is: ordinaryIs, as: asLong};

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: = **new CLASS**(*localBindings*: {},
super: *GeneralNumber*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *prototype*: *GeneralNumberPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNotNull*;

```

proc callLong(this: OBJECT, args: OBJECT[], phase: PHASE): LONG
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT ← defaultArg(args, 0, +zero164);
  i: INTEGER ← objectToPreciseInteger(arg, phase);
  if −263 ≤ i ≤ 263 − 1 then return ilong
  else throw a RangeError exception — i is out of the LONG range
  end if
end proc;

proc constructLong(args: OBJECT[], phase: PHASE): LONG
  return callLong(null, args, phase)
end proc;

proc asLong(o: OBJECT, c: CLASS, silent: BOOLEAN): LONG
  if o ∉ GENERALNUMBER then throw a TypeError exception end if;
  i: INTEGEROPT ← checkInteger(o);
  if i ≠ none and −263 ≤ i ≤ 263 − 1 then return ilong
  else throw a RangeError exception — i is out of the LONG range
  end if
end proc;

longPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {
  stdConstBinding(public::“MAX VALUE”, ulong, (264 − 1)ulong),
  stdConstBinding(public::“MIN VALUE”, ulong, 0ulong)},
  archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
  construct: none, env: none};

```

ulong

```

new CLASS{localBindings: {
  stdConstBinding(public::“MAX VALUE”, ulong, (264 − 1)ulong),
  stdConstBinding(public::“MIN VALUE”, ulong, 0ulong)},
  instanceProperties: {}, super: GeneralNumber, prototype: ulongPrototype, complete: true, name: “ulong”,
  typeofString: “ulong”, dynamic: false, final: true, defaultValue: 0ulong, bracketRead: ordinaryBracketRead,
  bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
  write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callULong,
  construct: constructULong, init: none, is: ordinaryIs, as: asULong};

proc callULong(this: OBJECT, args: OBJECT[], phase: PHASE): ULONG
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT ← defaultArg(args, 0, +zero164);
  i: INTEGER ← objectToPreciseInteger(arg, phase);
  if 0 ≤ i ≤ 264 − 1 then return iulong
  else throw a RangeError exception — i is out of the ULONG range
  end if
end proc;

proc constructULong(args: OBJECT[], phase: PHASE): ULONG
  return callULong(null, args, phase)
end proc;

```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: = **new** CLASS{localBindings: {},
super: *GeneralNumber*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *prototype*: *GeneralNumberPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *call*: *dummyCall*,
construct: *dummyConstruct*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNoNull*;

```

proc asULong(o: OBJECT, c: CLASS, silent: BOOLEAN): ULONG
  if o ∉ GENERALNUMBER then throw a TypeError exception end if;
  i: INTEGEROPT ← checkInteger(o);
  if i ≠ none and 0 ≤ i ≤ 264 − 1 then return iulong
  else throw a RangeError exception — i is out of the ULONG range
  end if
end proc;

ulongPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {
  stdConstBinding(public::"constructor", Class, ulong)},
  archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
  construct: none, env: none};

```

float

```

float: CLASS = new CLASS{localBindings: {
  stdConstBinding(public::"MAX_VALUE", float, (3.4028235×1038)f32),
  stdConstBinding(public::"MIN_VALUE", float, (10-45)f32),
  stdConstBinding(public::"NaN", float, NaNf32),
  stdConstBinding(public::"NEGATIVE_INFINITY", float, −∞f32),
  stdConstBinding(public::"POSITIVE_INFINITY", float, +∞f32)},
  instanceProperties: {}, super: GeneralNumber, prototype: floatPrototype, complete: true, name: "float",
  typeofString: "float", dynamic: false, final: true, defaultValue: NaNf32, bracketRead: ordinaryBracketRead,
  bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
  write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callFloat,
  construct: constructFloat, init: none, is: ordinaryIs, as: asFloat};

proc callFloat(this: OBJECT, args: OBJECT[], phase: PHASE): FLOAT32
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  arg: OBJECT ← defaultArg(args, 0, +zerof32);
  return objectToFloat32(arg, phase)
end proc;

proc constructFloat(args: OBJECT[], phase: PHASE): FLOAT32
  return callFloat(none, args, phase)
end proc;

proc asFloat(o: OBJECT, c: CLASS, silent: BOOLEAN): FLOAT32
  if o ∉ GENERALNUMBER then return toFloat32(o) else throw a TypeError exception end if
end proc;

floatPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {
  stdConstBinding(public::"constructor", Class, float)},
  archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
  construct: none, env: none};

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** CLASS{localBindings: {},
super: GeneralNumber,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: prototype: GeneralNumberPrototype,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: call: dummyCall,
construct: dummyConstruct,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNoNull;

Number

```
Number: CLASS = new CLASS{localBindings: {
  stdConstBinding(public::"MAX_VALUE", Number, (1.7976931348623157×10308)164),
  stdConstBinding(public::"MIN_VALUE", Number, (5×10-324)164),
  stdConstBinding(public::"NaN", Number, NaN164),
  stdConstBinding(public::"NEGATIVE_INFINITY", Number, -∞164),
  stdConstBinding(public::"POSITIVE_INFINITY", Number, +∞164)},
instanceProperties: {}, super: GeneralNumber, prototype: NumberPrototype, complete: true,
name: "Number", typeofString: "number", dynamic: false, final: true, defaultValue: NaN164,
bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite,
bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete,
enumerate: ordinaryEnumerate, call: callNumber, construct: constructNumber, init: none, is: ordinaryIs,
as: asNumber};

proc callNumber(this: OBJECT, args: OBJECT[], phase: PHASE): FLOAT64
note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
if |args| > 1 then
  throw an ArgumentError exception — at most one argument can be supplied
end if;
arg: OBJECT ← defaultArg(args, 0, +zero164);
return objectToFloat64(arg, phase)
end proc;

proc constructNumber(args: OBJECT[], phase: PHASE): FLOAT64
return callNumber(null, args, phase)
end proc;

proc asNumber(o: OBJECT, c: CLASS, silent: BOOLEAN): FLOAT64
if o ∈ GENERALNUMBER then return toFloat64(o) else throw a TypeError exception end if
end proc;

NumberPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {
  stdConstBinding(public::"constructor", Class, Number)},
archetype: GeneralNumberPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none,
construct: none, env: none};
```

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: = new CLASS{localBindings: {},
super: GeneralNumber,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: prototype: GeneralNumberPrototype,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: call: dummyCall,
construct: dummyConstruct,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: ordinaryImplicitCoerceNoNull;

```

proc makeBuiltInIntegerClass(name: STRING, low: INTEGER, high: INTEGER): CLASS
  proc call(this: OBJECT, args: OBJECT[], phase: PHASE): OBJECT
    note This function can be used in constant expressions if this can be converted to a primitive in constant
      expressions.
    if |args| > 1 then
      throw an ArgumentError exception — at most one argument can be supplied
    end if;
    arg: OBJECT ← defaultArg(args, 0, +zero164);
    x: FLOAT64 ← objectToFloat64(arg, phase);
    i: INTEGEROPT ← checkInteger(x);
    if i ≠ none and low ≤ i ≤ high then
      note -zero164 is coerced to +zero164.
      return realToFloat64(i)
    end if;
    throw a RangeError exception
  end proc;
  proc construct(args: OBJECT[], phase: PHASE): OBJECT
    return call(null, args, phase)
  end proc;
  proc is(o: OBJECT, c: CLASS): BOOLEAN
    if o ∉ FLOAT64 then return false end if;
    i: INTEGEROPT ← checkInteger(o);
    return i ≠ none and low ≤ i ≤ high
  end proc;
  proc as(o: OBJECT, c: CLASS, silent: BOOLEAN): OBJECT
    if o ∉ GENERALNUMBER then throw a TypeError exception end if;
    i: INTEGEROPT ← checkInteger(o);
    if i ≠ none and low ≤ i ≤ high then
      note -zero132, +zero132, and -zero164 are all coerced to +zero164.
      return realToFloat64(i)
    end if;
    throw a RangeError exception
  end proc;
  return new CLASS(localBindings: {
    stdConstBinding(public::"MAX_VALUE", Number, realToFloat64(high)),
    stdConstBinding(public::"MIN_VALUE", Number, realToFloat64(low)),
    instanceProperties: {}, super: Number.prototype, complete: true, name: name,
    typeOfString: "number", dynamic: false, final: true, defaultValue: +zero164,
    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, as: as
  })
end proc;

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);

```

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: ????

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: ????

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: if *o* ∈ FLOAT64 then [58]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: end if [59]

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: return new CLASS{localBindings:
{}, super: Number,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: implicitCoerce}

Character

Character: **CLASS** $\leftarrow$ **new** **CLASS**{localBindings: {stdFunction(public::"fromCharCode", *Character* fromCharCode, 1)}, instanceProperties: {}, super: *Object*, prototype: *CharacterPrototype*, complete: **true**, name: "Character", typeofString: "character", dynamic: **false**, final: **true**, defaultValue: "«NUL»", bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callCharacter, construct: constructCharacter, init: **none**, is: ordinaryIs, as: asCharacter};

proc callCharacter(*this*: **OBJECT**, *args*: **OBJECT**[], *phase*: **PHASE**): **CHAR16**
note This function can be used in constant expressions if *this* can be converted to a primitive in constant expressions.
if |*args*| $\neq$ 1 **then**
 throw an *ArgumentError* exception — exactly one argument must be supplied
end if;
s: **STRING** $\leftarrow$ objectToString(*args*[0], *phase*);
if |*s*| $\neq$ 1 **then throw** a *RangeError* exception — only one character may be given **end if**;
return *s*[0]
end proc;

proc constructCharacter(*args*: **OBJECT**[], *phase*: **PHASE**): **CHAR16**
if |*args*| = 0 **then return** "«NUL»" **else return** callCharacter(**null**, *args*, *phase*) **end if**
end proc;

proc asCharacter(*o*: **OBJECT**, *c*: **CLASS**, *silent*: **BOOLEAN**): **CHAR16**
if *o* $\in$ **CHAR16** **then return** *o* **else throw** a *TypeError* exception **end if**
end proc;

proc *Character* fromCharCode(*this*: **OBJECT**, *f*: **SIMPLEINSTANCE**, *args*: **OBJECT**[], *phase*: **PHASE**): **OBJECT**
note This function can be used in constant expressions if the argument can be converted to a primitive in constant expressions.
if |*args*| $\neq$ 1 **then**
 throw an *ArgumentError* exception — exactly one argument must be supplied
end if;
i: **EXTENDEDINTEGER** $\leftarrow$ objectToImpreciseInteger(*args*[0], *phase*);
if *i* $\notin$ {+ ∞ , - ∞ , NaN} **and** 0 $\leq$ *i* $\leq$ 0xFFFF **then return** integerToChar16(*i*)
 else throw a *RangeError* exception — character code out of range
end if
end proc;

CharacterPrototype: **SIMPLEINSTANCE** $\leftarrow$ **new** **SIMPLEINSTANCE**{localBindings: {stdConstBinding(public::"constructor", *Class*, *Character*)}, archetype: *StringPrototype*, sealed: prototypesSealed, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

String

String: **CLASS** $\leftarrow$ **new** **CLASS**{localBindings: {stdFunction(public::"fromCharCode", *String* fromCharCode, 1)}, instanceProperties: {stringLengthGetter}, super: *Object*, prototype: *StringPrototype*, complete: **true**, name: "String", typeofString: "string", dynamic: **false**, final: **true**, defaultValue: "", bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: readString, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: callString, construct: constructString, init: **none**, is: ordinaryIs, as: asString};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** **CLASS**{localBindings: {}, super: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: call: dummyCall, construct: dummyConstruct,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** **SIMPLEINSTANCE**{localBindings: {}, super: *ObjectPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: = **new** **CLASS**{localBindings: {}, super: *Object*, instanceProperties: {}, prototype: *StringPrototype*, complete: **true**, name: "String", typeofString: "string", dynamic: **false**, final: **true**, defaultValue: **null**, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: **none**, is: ordinaryIs, implicitCoerce: ordinaryImplicitCoerceNull; - ... [60]


```

proc readString(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, phase: PHASE): OBJECTOPT
  note o ∈ STRING because readString is only called on instances of class String.
  if limit = String then
    i: INTEGEROPT ← multinameToArrayIndex(multiname);
    if i ≠ none then if i < |o| then return o[i] else return undefined end if end if
  end if;
  return ordinaryRead(o, limit, multiname, env, phase)
end proc;

proc callString(this: OBJECT, args: OBJECT[], phase: PHASE): STRING
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  return objectToString(defaultArg(args, 0, ""), phase)
end proc;

proc constructString(args: OBJECT[], phase: PHASE): STRING
  return callString(null, args, phase)
end proc;

proc asString(o: OBJECT, c: CLASS, silent: BOOLEAN): STRING
  if o ∈ CHAR16 ∪ STRING then return toString(o) else throw a TypeError exception end if
end proc;

stringLengthGetter: INSTANCEGETTER = new INSTANCEGETTER(multiname: {public::"length"}, final: true,
  enumerable: false, call: String_length);

proc String_length(this: OBJECT, phase: PHASE): OBJECT
  note this ∈ STRING because this getter cannot be extracted from the String class.
  length: INTEGER ← |this|;
  return realToFloat64(length)
end proc;

proc String_fromCharCode(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if the arguments can be converted to primitives in constant
  expressions.
  s: STRING ← "";
  for each arg ∈ args do
    i: EXTENDEDINTEGER ← objectToImpreciseInteger(arg, phase);
    if i ∉ {+∞, -∞, NaN} and 0 ≤ i ≤ 0x10FFFF then s ← s ⊕ integerToUTF16(i)
    else throw a RangeError exception — character code out of range
  end if
end for each;
  return s
end proc;

```

```

StringPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE(localBindings: {
  stdConstBinding(public::"constructor", Class, String),
  stdFunction(public::"toString", String_toString, 0),
  stdFunction(public::"charAt", String_charAt, 1),
  stdFunction(public::"charCodeAt", String_charCodeAt, 1),
  stdFunction(public::"concat", String_concat, 1),
  stdFunction(public::"indexOf", String_indexOf, 1),
  stdFunction(public::"lastIndexOf", String_lastIndexOf, 1),
  stdFunction(public::"localeCompare", String_localeCompare, 1),
  stdFunction(public::"match", String_match, 1),
  stdFunction(public::"replace", String_replace, 1),
  stdFunction(public::"search", String_search, 1),
  stdFunction(public::"slice", String_slice, 2),
  stdFunction(public::"split", String_split, 2),
  stdFunction(public::"substring", String_substring, 2),
  stdFunction(public::"toLowerCase", String_toLowerCase, 0),
  stdFunction(public::"toLocaleLowerCase", String_toLocaleLowerCase, 0),
  stdFunction(public::"toUpperCase", String_toUpperCase, 0),
  stdFunction(public::"toLocaleUpperCase", String_toLocaleUpperCase, 0)},
  archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none,
  env: none);

proc String_toString(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  note This function is generic and can be applied even if this is not a string.
  note This function ignores any arguments passed to it in args.
  return objectToString(this, phase)
end proc;

proc String_charAt(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the argument can be converted to primitives in
  constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  position: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +zero164), phase);
  if position = NaN then position ← 0 end if;
  if position ∉ {+∞, -∞} and 0 ≤ position < |s| then return [s[position]]
  else return ""
  end if
end proc;

```

```

proc String_charCodeAt(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the argument can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 1 then
    throw an ArgumentError exception — at most one argument can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  position: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +zero164), phase);
  if position = NaN then position ← 0 end if;
  if position ∉ {+∞, -∞} and 0 ≤ position < |s| then
    return realToFloat64(char16ToInteger(s[position]))
  else return NaN164
  end if
end proc;

proc String_concat(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the argument can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  for each arg ∈ args do s ← s ⊕ objectToString(arg, phase) end for each;
  return s
end proc;

proc String_indexOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the arguments can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| ∉ {1, 2} then
    throw an ArgumentError exception — at least one and at most two arguments must be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  pattern: STRING ← objectToString(args[0], phase);
  position: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 1, +zero164), phase);
  if position ∈ {-∞, NaN} then position ← 0
  elseif position = +∞ or position > |s| then position ← |s|
  elseif position < 0 then position ← 0
  end if;
  while position + |pattern| ≤ |s| do
    if s[position ... position + |pattern| - 1] = pattern then
      return realToFloat64(position)
    end if;
    position ← position + 1
  end while;
  return -1.0164
end proc;

```

```

proc String.lastIndexOf(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the arguments can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args|  $\notin$  {1, 2} then
    throw an ArgumentError exception — at least one and at most two arguments must be supplied
  end if;
  s: STRING  $\leftarrow$  objectToString(this, phase);
  pattern: STRING  $\leftarrow$  objectToString(args[0], phase);
  position: EXTENDEDINTEGER  $\leftarrow$  objectToImpreciseInteger(defaultArg(args, 1,  $+\infty_{f64}$ ), phase);
  if position =  $-\infty$  then position  $\leftarrow$  0
  elseif position  $\in$  { $+\infty$ , NaN} or position > |s| then position  $\leftarrow$  |s|
  elseif position < 0 then position  $\leftarrow$  0
  end if;
  if position + |pattern| > |s| then position  $\leftarrow$  |s| - |pattern| end if;
  while position  $\geq$  0 do
    if s[position ... position + |pattern| - 1] = pattern then
      return realToFloat64(position)
    end if;
    position  $\leftarrow$  position - 1
  end while;
  return  $-1.0_{f64}$ 
end proc;

proc String.localeCompare(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — localeCompare cannot be called from constant expressions
  end if;
  if |args| < 1 then
    throw an ArgumentError exception — at least one argument must be supplied
  end if;
  s1: STRING  $\leftarrow$  objectToString(this, phase);
  s2: STRING  $\leftarrow$  objectToString(args[0], phase);
  Let result: OBJECT be a value of type Number that is the result of a locale-sensitive string comparison of s1 and s2. The
  two strings are compared in an implementation-defined fashion. The result is intended to order string in the sort order
  specified by the system default locale, and will be negative, zero, or positive, depending on whether s1 comes before s2
  in the sort order, the strings are equal, or s1 comes after s2 in the sort order, respectively. The result shall not be
  NaNf64. The comparison shall be a consistent comparison function on the set of all strings.
  return result
end proc;

proc String.match(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — match cannot be called from constant expressions
  end if;
  if |args|  $\neq$  1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  s: STRING  $\leftarrow$  objectToString(this, phase);
  ???
end proc;

```

```

proc String_replace(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — replace cannot be called from constant expressions
  end if;
  if |args| ≠ 2 then
    throw an ArgumentError exception — exactly two arguments must be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  ???
end proc;

proc String_search(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — search cannot be called from constant expressions
  end if;
  if |args| ≠ 1 then
    throw an ArgumentError exception — exactly one argument must be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  ???
end proc;

proc String_slice(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the arguments can be converted to primitives in constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  start: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +zero164), phase);
  end: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +∞164), phase);
  if start ∈ {-∞, NaN} then start ← 0
  elseif start = +∞ or start > |s| then start ← |s|
  elseif start < 0 then start ← start + |s|; if start < 0 then start ← 0 end if
  end if;
  if end = -∞ then end ← 0
  elseif end ∈ {+∞, NaN} or end > |s| then end ← |s|
  elseif end < 0 then end ← end + |s|; if end < 0 then end ← 0 end if
  end if;
  if start < end then return s[start ... end - 1] else return "" end if
end proc;

proc String_split(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — split cannot be called from constant expressions
  end if;
  if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  ???
end proc;

```

```

proc String_substring(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this and the arguments can be converted to primitives in
    constant expressions.
  note This function is generic and can be applied even if this is not a string.
  if |args| > 2 then
    throw an ArgumentError exception — at most two arguments can be supplied
  end if;
  s: STRING ← objectToString(this, phase);
  start: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +ZERO164), phase);
  end: EXTENDEDINTEGER ← objectToImpreciseInteger(defaultArg(args, 0, +∞164), phase);
  if start ∈ {−∞, NaN} then start ← 0
  elseif start = +∞ or start > |s| then start ← |s|
  elseif start < 0 then start ← 0
  end if;
  if end = −∞ then end ← 0
  elseif end ∈ {+∞, NaN} or end > |s| then end ← |s|
  elseif end < 0 then end ← 0
  end if;
  if start ≤ end then return s[start ... end − 1]
  else return s[end ... start − 1]
  end if
end proc;

proc String_toLowerCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  r: STRING ← "";
  for each ch ∈ s do r ← r ⊕ charToLowerCaseFull(ch) end for each;
  return r
end proc;

proc String_toLocaleLowerCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — toLocaleLowerCase cannot be called from constant expressions
  end if;
  s: STRING ← objectToString(this, phase);
  r: STRING ← "";
  for each ch ∈ s do r ← r ⊕ charToLowerCaseLocalized(ch) end for each;
  return r
end proc;

proc String_toUpperCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function can be used in constant expressions if this can be converted to a primitive in constant expressions.
  note This function is generic and can be applied even if this is not a string.
  s: STRING ← objectToString(this, phase);
  r: STRING ← "";
  for each ch ∈ s do r ← r ⊕ charToUpperCaseFull(ch) end for each;
  return r
end proc;

```

```

proc String toLocaleUpperCase(this: OBJECT, f: SIMPLEINSTANCE, args: OBJECT[], phase: PHASE): OBJECT
  note This function is generic and can be applied even if this is not a string.
  if phase = compile then
    throw a ConstantError exception — toLocaleUpperCase cannot be called from constant expressions
  end if;
  s: STRING ← objectToString(this, phase);
  r: STRING ← "";
  for each ch ∈ s do r ← r ⊕ charToUpperLocalized(ch) end for each;
  return r
end proc;

```

Array

```

Array: CLASS = new CLASS{localBindings: {}, instanceProperties: {}, super: Object, prototype: ArrayPrototype,
  complete: true, name: "Array", typeofString: "object", privateNamespace: arrayPrivate, dynamic: true,
  final: true, defaultValue: null, defaultHint: hintNumber, bracketRead: ordinaryBracketRead,
  bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead,
  write: writeArray, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall,
  construct: dummyConstruct, init: none, is: ordinaryIs, as: ordinaryAs};

```

arrayLimit: INTEGER = $2^{64} - 1$;

arrayPrivate: NAMESPACE = **new** NAMESPACE{*name*: "private"};

```

proc writeArray(o: OBJECT, limit: CLASS, multiname: MULTINAME, env: ENVIRONMENTOPT, createIfMissing: BOOLEAN,
  newValue: OBJECT, phase: {run}): {none, ok}
  result: {none, ok} ← ordinaryWrite(o, limit, multiname, env, createIfMissing, newValue, phase);
  if result = ok then
    i: INTEGEROPT ← multinameToArrayIndex(multiname);
    if i ≠ none then
      length: ULONG ← readInstanceSlot(o, arrayPrivate::"length", phase);
      if i ≥ length.value then
        length ← (i + 1)ulong;
        doWrite(o, {arrayPrivate::"length"}, length, phase)
      end if
    end if
  end if;
  return result
end proc;

```

```

proc multinameToArrayIndex(multiname: MULTINAME): INTEGEROPT
  if |multiname| ≠ 1 then return none end if;
  qname: QUALIFIEDNAME ← the one element of multiname;
  if qname.namespace ≠ public then return none end if;
  name: STRING ← qname.id;
  if name ≠ [] then
    if name = "0" then return 0
    elseif name[0] ≠ '0' and (every ch ∈ name satisfies ch ∈ {'0' ... '9'}) then
      i: INTEGER ← stringToInteger(name, 10);
      if i < arrayLimit then return i end if
    end if
  end if;
  return none
end proc;

```

```

ArrayPrototype: SIMPLEINSTANCE = new SIMPLEINSTANCE{localBindings: {}, archetype: ObjectPrototype,
  sealed: prototypesSealed, type: Object, slots: {}, call: none, construct: none, env: none};

```

Namespace

Namespace: **CLASS** = **new CLASS**(localBindings: {}, instanceProperties: {}, super: *Object*,
prototype: *NamespacePrototype*, complete: **true**, name: "Namespace", typeofString: "namespace",
dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: *hintString*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, as: *ordinaryAs*);

proc *callNamespace*(this: **OBJECT**, args: **OBJECT**[], phase: **PHASE**): **NAMESPACE** U **NULL**
note This function can be used in constant expressions.
if |args| ≠ 1 **then**
 throw an *ArgumentError* exception — exactly one argument must be supplied
end if;
arg: **OBJECT** ← args[0];
if arg ∈ **NAMESPACE** U **NULL** **then return** arg **else throw** a *TypeError* exception **end if**
end proc;

proc *constructNamespace*(args: **OBJECT**[], phase: **PHASE**): **NAMESPACE**
note This function can be used in constant expressions if its argument is a string.
if |args| > 1 **then**
 throw an *ArgumentError* exception — at most one argument can be supplied
end if;
arg: **OBJECT** ← defaultArg(args, 0, **undefined**);
if arg ∈ **NULL** U **UNDEFINED** **then**
 if phase = **compile** **then**
 throw a *ConstantError* exception — constant expressions cannot construct new anonymous namespaces
 end if;
 return new NAMESPACE(name: "anonymous")
elseif arg ∈ **CHAR16** U **STRING** **then**
 name: **STRING** ← *toString*(arg);
 if name = "" **then return** *public*
 else
 return a namespace generated from the URI in name in an implementation-defined manner. Constructing a
 namespace twice using the same name shall return the same namespace. Constructing namespaces using different
 values of name may or may not return the same namespace, depending on whether the implementation considers
 the differences in the names to be significant. Constructing a namespace from name shall not return any of the
 private or internal namespaces that are constructed elsewhere in this specification.
 end if
 else throw a *TypeError* exception
 end if
end proc;

uriNamespace: **NAMESPACE** = **new NAMESPACE**(name: "URI Namespace");

NamespacePrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**(localBindings: {
 stdFunction(public: "toString", *Namespace toString*, 0),
 archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**,
 env: **none**);

proc *Namespace toString*(this: **OBJECT**, f: **SIMPLEINSTANCE**, args: **OBJECT**[], phase: **PHASE**): **STRING**
note This function can be used in constant expressions.
note This function ignores any arguments passed to it in args.
if this ∉ **NAMESPACE** **then throw** a *TypeError* exception **end if**;
return this.name
end proc;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: prototype: *ObjectPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;
;

Attribute

Attribute: **CLASS** = **new CLASS**{localBindings: {}, instanceProperties: {}, super: Object, prototype: ObjectPrototype, complete: **true**, name: "Attribute", typeofString: "object", dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: hintString, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: **none**, is: ordinaryIs, as: ordinaryAs};

Date

Date: **CLASS** = **new CLASS**{localBindings: {}, instanceProperties: {}, super: Object, prototype: DatePrototype, complete: **true**, name: "Date", typeofString: "object", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: hintString, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: **none**, is: ordinaryIs, as: ordinaryAs};

DatePrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**{localBindings: {}, archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: **none**, construct: **none**, env: **none**};

RegExp

RegExp: **CLASS** = **new CLASS**{localBindings: {}, instanceProperties: {}, super: Object, prototype: RegExpPrototype, complete: **true**, name: "RegExp", typeofString: "object", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: hintNumber, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: **none**, is: ordinaryIs, as: ordinaryAs};

RegExpPrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**{localBindings: {}, archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: **none**, construct: **none**, env: **none**};

Class

Class: **CLASS** = **new CLASS**{localBindings: {}, instanceProperties: {classPrototypeGetter}, super: Object, prototype: ClassPrototype, complete: **true**, name: "Class", typeofString: "function", dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: hintString, bracketRead: ordinaryBracketRead, bracketWrite: ordinaryBracketWrite, bracketDelete: ordinaryBracketDelete, read: ordinaryRead, write: ordinaryWrite, delete: ordinaryDelete, enumerate: ordinaryEnumerate, call: dummyCall, construct: dummyConstruct, init: **none**, is: ordinaryIs, as: ordinaryAs};

classPrototypeGetter: **INSTANCEGETTER** = **new INSTANCEGETTER**{multiname: {public: "prototype"}, final: **true**, enumerable: **false**, call: Class.prototype};

proc *Class.prototype*(this: OBJECT, phase: PHASE): OBJECT
note *this* ∈ CLASS because this getter cannot be extracted from the Class class.
prototype: OBJECT ← *this.prototype*;
if *prototype* = **none** **then return undefined else return prototype end if**
end proc;

ClassPrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**{localBindings: {}, archetype: ObjectPrototype, sealed: prototypesSealed, type: Object, slots: {}, call: **none**, construct: **none**, env: **none**};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: Object,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: Object,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: ObjectPrototype,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: Object,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: ObjectPrototype,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: instanceProperties: {},
 prototype: ObjectPrototype,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: ordinaryImplicitCoerceNull};

Function

Function: **CLASS** = **new CLASS**(localBindings: {}, instanceProperties: {ivarFunctionLength}, super: *Object*, prototype: *FunctionPrototype*, complete: **true**, name: "Function", typeofString: "function", dynamic: **false**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, as: *ordinaryAs*);

ivarFunctionLength: **INSTANCEVARIABLE** = **new INSTANCEVARIABLE**(multiname: {public::"length"}, final: **true**, enumerable: **false**, type: *Number*, defaultValue: **none**, immutable: **true**);

FunctionPrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**(localBindings: {}, archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**);

PrototypeFunction

PrototypeFunction: **CLASS** = **new CLASS**(localBindings: {}, instanceProperties: {**new INSTANCEVARIABLE**(multiname: {public::"prototype"}, final: **true**, enumerable: **false**, type: *Object*, defaultValue: **undefined**, immutable: **false**)}, super: *Function*, prototype: *FunctionPrototype*, complete: **true**, name: "Function", typeofString: "function", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, as: *ordinaryAs*);

Package

Package: **CLASS** = **new CLASS**(localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *ObjectPrototype*, complete: **true**, name: "Package", typeofString: "object", dynamic: **true**, final: **true**, defaultValue: **null**, defaultHint: **hintString**, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, as: *ordinaryAs*);

Error

Error: **CLASS** = **new CLASS**(localBindings: {}, instanceProperties: {}, super: *Object*, prototype: *ErrorPrototype*, complete: **true**, name: "Error", typeofString: "object", dynamic: **true**, final: **false**, defaultValue: **null**, defaultHint: **hintNumber**, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, as: *ordinaryAs*);

ErrorPrototype: **SIMPLEINSTANCE** = **new SIMPLEINSTANCE**(localBindings: {}, archetype: *ObjectPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**);

proc *systemError*(e: **CLASS**, msg: **STRING** **U** **UNDEFINED**): **OBJECT**
 return *e*.construct([msg], run)
end proc;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: instanceProperties: {**new INSTANCEVARIABLE**(multiname: {public::"length"}, final: **true**, enumerable: **false**, type: *Number*, defaultValue: **none**, immutable: **true**)},

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: *ordinaryImplicitCoerceNull* ; -

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: *ObjectPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: **proc** *systemFunction*(call: (**OBJECT** **x** **SIMPLEINSTANCE** **x** **OBJECT**[] **x** **PHASE** **→** **OBJECT**) **U** {**none**}, length: **INTEGER**): **SIMPLEINSTANCE** [61]

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: *Function*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: *ordinaryImplicitCoerceNull* ; -

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: *ordinaryImplicitCoerceNull* ;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: *Object*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: *ordinaryImplicitCoerceNull* ;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: *ObjectPrototype*,

ArgumentError

ArgumentError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *ArgumentErrorPrototype*, complete: **true**, name: "ArgumentError", typeofString: "object",
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

ArgumentErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

AttributeError

AttributeError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *AttributeErrorPrototype*, complete: **true**, name: "AttributeError", typeofString: "object",
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

AttributeErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

ConstantError

ConstantError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *ConstantErrorPrototype*, complete: **true**, name: "ConstantError", typeofString: "object",
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

ConstantErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

DefinitionError

DefinitionError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *DefinitionErrorPrototype*, complete: **true**, name: "DefinitionError", typeofString: "object",
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

DefinitionErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

EvalError

EvalError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *EvalErrorPrototype*, complete: **true**, name: "EvalError", typeofString: "object",
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

EvalErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: implicitCoerce: *ordinaryImplicitCoerceNull*;

Waldemar Horwat 6/4/2003 3:42 PM
Deleted: super: *ErrorPrototype*,

RangeError

RangeError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *RangeErrorPrototype*, complete: **true**, name: “RangeError”, typeofString: “object”,
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

RangeErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

ReferenceError

ReferenceError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *ReferenceErrorPrototype*, complete: **true**, name: “ReferenceError”, typeofString: “object”,
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

ReferenceErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

SyntaxError

SyntaxError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *SyntaxErrorPrototype*, complete: **true**, name: “SyntaxError”, typeofString: “object”,
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

SyntaxErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

TypeError

TypeError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *TypeErrorPrototype*, complete: **true**, name: “TypeError”, typeofString: “object”,
dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*,
bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*,
write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*,
construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*};

TypeErrorPrototype: **SIMPLEINSTANCE** = new **SIMPLEINSTANCE**{localBindings: {}, *archetype*: *ErrorPrototype*,
sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**};

UninitializedError

UninitializedError: **CLASS** = new **CLASS**{localBindings: {}, instanceProperties: {}, *super*: *Error*,
prototype: *UninitializedErrorPrototype*, complete: **true**, name: “UninitializedError”,
typeofString: “object”, dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*,
bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*,
bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*,
enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*,
as: *ordinaryAs*};

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*;
;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*;
;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*;
;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*;
;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *ErrorPrototype*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *super*: *Error*,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: *implicitCoerce*: *ordinaryImplicitCoerceNull*;
;

UninitializedErrorPrototype: **SIMPLEINSTANCE** = **new** **SIMPLEINSTANCE**(localBindings: {}, *archetype*: *ErrorPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**);

URIError

URIError: **CLASS** = **new** **CLASS**(localBindings: {}, *instanceProperties*: {}, *super*: *Error*, prototype: *URIErrorPrototype*, complete: **true**, name: "URIError", typeofString: "object", dynamic: **true**, final: **false**, defaultValue: **null**, *defaultHint*: *hintNumber*, bracketRead: *ordinaryBracketRead*, bracketWrite: *ordinaryBracketWrite*, bracketDelete: *ordinaryBracketDelete*, read: *ordinaryRead*, write: *ordinaryWrite*, delete: *ordinaryDelete*, enumerate: *ordinaryEnumerate*, call: *dummyCall*, construct: *dummyConstruct*, init: **none**, is: *ordinaryIs*, *as*: *ordinaryAs*);

URIErrorPrototype: **SIMPLEINSTANCE** = **new** **SIMPLEINSTANCE**(localBindings: {}, *archetype*: *ErrorPrototype*, sealed: *prototypesSealed*, type: *Object*, slots: {}, call: **none**, construct: **none**, env: **none**);

Index

Nonterminals

Update fields to generate this index

Tags

Update fields to generate this index

Semantic Domains

Update fields to generate this index

Globals

Update fields to generate this index

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: 22-May-2003

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: ErrorPrototype,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: Error,

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: implicitCoerce: ordinaryImplicitCoerceNull;

Waldemar Horwat 6/4/2003 3:42 PM

Deleted: super: ErrorPrototype,