Data Model

Errors

tag syntaxError;

tag referenceError;

tag typeError;

tag propertyNotFoundError;

tag argumentMismatchError;

SEMANTICERROR
= {**syntaxError**, **referenceError**, **typeError**, **propertyNotFoundError**, **argumentMismatchError**};

tuple GOBREAK
value: OBJECT,
~~label: STRING~~label: LABEL
end tuple;

tuple GOCONTINUE
value: OBJECT,
~~label: STRING~~label: LABEL
end tuple;

tuple GORETURN
value: OBJECT
end tuple;

tuple GOTHROW
value: OBJECT
end tuple;

EARLYEXIT = GOBREAK ∪ GOCONTINUE ∪ GORETURN ∪ GOTHROW;

SEMANTICEXCEPTION = EARLYEXIT ∪ SEMANTICERROR;

Objects

OBJECT = UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪
METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE;

Undefined

tag undefined;

UNDEFINED = {**undefined**};

Null

tag null;

NULL = {**null**};

Namespaces

record NAMESPACE
  name: STRING
end record;

NAMESPACEOPT = NULL ∪ NAMESPACE;

publicNamespace: NAMESPACE = **new** NAMESPACE⟨⟨"`public`"⟩⟩;

Attributes

tag **static**;

tag **constructor**;

tag **operator**;

tag **abstract**;

tag **virtual**;

tag **final**;

MEMBERMODIFIER = {**null**, **static**, **constructor**, **operator**, **abstract**, **virtual**, **final**};

tag **mayOverride**;

tag **override**;

OVERRIDEMODIFIER = {**null**, **mayOverride**, **override**};

tuple ~~ATTRIBUTE~~COMPOUNDATTRIBUTE
  namespaces: NAMESPACE{},
  local: BOOLEAN,
  extend: CLASSOPT,
  enumerable: BOOLEAN,
  dynamic: BOOLEAN,
  memberMod: MEMBERMODIFIER,
  overrideMod: OVERRIDEMODIFIER,
  prototype: BOOLEAN,
  unused: BOOLEAN
end tuple;

ATTRIBUTE = BOOLEAN ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE;

ATTRIBUTENOTFALSE = {**true**} ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE;

Classes

record CLASS
  super: CLASSOPT,
  prototype: OBJECT,
  globalMembers: GLOBALMEMBER{},
  instanceMembers: INSTANCEMEMBER{},
  dynamic: BOOLEAN,
  primitive: BOOLEAN,
  privateNamespace: NAMESPACE,
  call: INVOKER,
  construct: INVOKER
end record;

```
CLASSOPT = NULL ∪ CLASS;

proc makeBuiltInClass(superclass: CLASSOPT, dynamic: BOOLEAN, primitive: BOOLEAN): CLASS
    privateNamespace: NAMESPACE ← new NAMESPACE⟨⟨"private"⟩⟩;
    proc call(this: OBJECT, args: ARGUMENTLIST): OBJECT
        ????
    end proc;
    proc construct(this: OBJECT, args: ARGUMENTLIST): OBJECT
        ????
    end proc;
    return new CLASS⟨⟨superclass, null, {}, {}, dynamic, primitive, privateNamespace, call, construct⟩⟩
end proc;

objectClass: CLASS = makeBuiltInClass(null, false, true);

undefinedClass: CLASS = makeBuiltInClass(objectClass, false, true);

nullClass: CLASS = makeBuiltInClass(objectClass, false, true);

booleanClass: CLASS = makeBuiltInClass(objectClass, false, true);

numberClass: CLASS = makeBuiltInClass(objectClass, false, true);

stringClass: CLASS = makeBuiltInClass(objectClass, false, false);

characterClass: CLASS = makeBuiltInClass(stringClass, false, false);

namespaceClass: CLASS = makeBuiltInClass(objectClass, false, false);

attributeClass: CLASS = makeBuiltInClass(objectClass, false, false);

classClass: CLASS = makeBuiltInClass(objectClass, false, false);

functionClass: CLASS = makeBuiltInClass(objectClass, false, false);

prototypeClass: CLASS = makeBuiltInClass(objectClass, true, false);
```

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

```
proc ancestors(c: CLASS): CLASS[]
    s: CLASSOPT ← c.super;
    if s = null 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 = null then return false end if;
        return isAncestor(c, s)
    end if
end proc;
```

Return **true** if *c* is an ancestor of *d* other than *d* itself.

```
proc isProperAncestor(c: CLASS, d: CLASS): BOOLEAN
    return isAncestor(c, d) and c ≠ d
end proc;
```

Members

```
tag read;
```

tag write;

tag readWrite;

MEMBERACCESS = {**read**, **write**, **readWrite**};

INSTANCECATEGORY = {**abstract**, **virtual**, **final**};

INSTANCEDATA = SLOTID ∪ METHOD ∪ ACCESSOR;

record INSTANCEMEMBER
~~name: STRING,~~
~~namespaces: NAMESPACE{},~~partialName: PARTIALNAME,
access: MEMBERACCESS,
category: INSTANCECATEGORY,
indexable: BOOLEAN,
enumerable: BOOLEAN,
data: INSTANCEDATA ∪ NAMESPACE
end record;

GLOBALCATEGORY = {**static**, **constructor**};

GLOBALDATA = GLOBALSLOT ∪ METHOD ∪ ACCESSOR;

record GLOBALMEMBER
~~name: STRING,~~
~~namespaces: NAMESPACE{},~~partialName: PARTIALNAME,
access: MEMBERACCESS,
category: GLOBALCATEGORY,
indexable: BOOLEAN,
enumerable: BOOLEAN,
data: GLOBALDATA ∪ NAMESPACE
end record;

MEMBER = INSTANCEMEMBER ∪ GLOBALMEMBER;

MEMBERDATA = INSTANCEDATA ∪ GLOBALDATA;

MEMBERDATAOPT = NULL ∪ MEMBERDATA;

record GLOBALSLOT
type: CLASS,
value: OBJECT
end record;

record METHOD
type: SIGNATURE,
f: INSTANCEOPT
end record;

record ACCESSOR
type: CLASS,
f: INSTANCE
end record;

Method Closures

tuple METHODCLOSURE
this: OBJECT,
method: METHOD
end tuple;

Prototype Instances

record PROTOTYPE
parent: PROTOTYPEOPT,
dynamicProperties: DYNAMICPROPERTY{}
end record;

PROTOTYPEOPT = NULL ∪ PROTOTYPE;

record DYNAMICPROPERTY
name: STRING,
value: OBJECT
end record;

Class Instances

INSTANCE = FIXEDINSTANCE ∪ DYNAMICINSTANCE;

INSTANCEOPT = NULL ∪ INSTANCE;

record FIXEDINSTANCE
type: CLASS,
call: INVOKER,
construct: INVOKER,
typeofString: STRING,
slots: SLOT{}
end record;

record DYNAMICINSTANCE
type: CLASS,
call: INVOKER,
construct: INVOKER,
typeofString: STRING,
slots: SLOT{},
dynamicProperties: DYNAMICPROPERTY{}
end record;

Slots

record SLOT
id: SLOTID,
value: OBJECT
end record;

record SLOTID
type: CLASS
end record;

Qualified Names

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

tuple PARTIALNAME
namespaces: NAMESPACE{},
name: STRING
end tuple;

Objects with Limits

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

tuple LIMITEDINSTANCE
  instance: INSTANCE,
  limit: CLASS
end tuple;

OBJOPTIONALLIMIT = OBJECT ∪ LIMITEDINSTANCE;

References

tuple VARIABLEREFERENCE
  env: ENVIRONMENT,
  partialName: PARTIALNAME
end tuple;

tuple DOTREFERENCE
  base: OBJOPTIONALLIMIT,
  propName: PARTIALNAME
end tuple;

tuple BRACKETREFERENCE
  base: OBJOPTIONALLIMIT,
  args: ARGUMENTLIST
end tuple;

REFERENCE = VARIABLEREFERENCE ∪ DOTREFERENCE ∪ BRACKETREFERENCE;

OBJORREF = OBJECT ∪ REFERENCE;

tuple LIMITEDOBJORREF
  ref: OBJORREF,
  limit: CLASS
end tuple;

OBJORREFOPTIONALLIMIT = OBJORREF ∪ LIMITEDOBJORREF;

Signatures

tuple SIGNATURE
  requiredPositional: CLASS[],
  optionalPositional: CLASS[],
  optionalNamed: NAMEDPARAMETER{},
  rest: CLASSOPT,
  restAllowsNames: BOOLEAN,
  returnType: CLASS
end tuple;

tuple NAMEDPARAMETER
  name: STRING,
  type: CLASS
end tuple;

Argument Lists

tuple NAMEDARGUMENT
  name: STRING,
  value: OBJECT
end tuple;

tuple ARGUMENTLIST
  positional: OBJECT[],
  named: NAMEDARGUMENT{}
end tuple;

The first OBJECT is the `this` value.

INVOKER = OBJECT × ARGUMENTLIST → OBJECT;

Unary Operators

tuple UNARYMETHOD
  operandType: CLASS,
  f: OBJECT × OBJECT × ARGUMENTLIST → OBJECT
end tuple;

Binary Operators

tuple BINARYMETHOD
  leftType: CLASS,
  rightType: CLASS,
  f: OBJECT × OBJECT → OBJECT
end tuple;

Contexts

tuple CONTEXT
  strict: BOOLEAN,
  unchecked: BOOLEAN,
  insideFunction: BOOLEAN,
  privateNamespace: NAMESPACEOPT,
  openNamespaces: NAMESPACE{},
  breakLabels: LABEL{},
  continueLabels: LABEL{}
end tuple;

initialContext: CONTEXT = CONTEXT⟨**false**, **false**, **false**, **null**, {*publicNamespace*}, {}, {}⟩;

Labels

tag default;

LABEL = STRING ∪ {**default**};

Environments

ENVIRONMENT = STATICENV ∪ DYNAMICENV;

Static Environments

record STATICFRAME
end record;

tuple STATICENV
frames: STATICFRAME[]
end tuple;

initialStaticEnv: STATICENV = STATICENV⟨[]⟩;

Dynamic Environments

record DYNAMICFRAME
end record;

tuple DEFINITION
name: QUALIFIEDNAME,
type: CLASS,
data: SLOT ∪ OBJECT ∪ ACCESSOR
end tuple;

tuple DYNAMICENV
frames: DYNAMICFRAME[]
end tuple;

initialDynamicEnv: DYNAMICENV = DYNAMICENV⟨[]⟩;

Data Operations

Numeric Utilities

proc *uInt32ToInt32*(*i*: INTEGER): INTEGER
if $i < 2^{31}$ **then return** *i* **else return** $i - 2^{32}$ **end if**
end proc;

proc *toUInt32*(*x*: FLOAT64): INTEGER
if *x* ∈ {**+∞**, **−∞**, **NaN**} **then return** 0 **end if**;
return *truncateFiniteFloat64*(*x*) **mod** 2^{32}
end proc;

proc *toInt32*(*x*: FLOAT64): INTEGER
return *uInt32ToInt32*(*toUInt32*(*x*))
end proc;

Object Utilities

objectType

objectType(*o*) returns an OBJECT *o*'s most specific type.

```
proc objectType(o: OBJECT): CLASS
    case o of
        UNDEFINED do return undefinedClass;
        NULL do return nullClass;
        BOOLEAN do return booleanClass;
        FLOAT64 do return numberClass;
        STRING do if |o| = 1 then return characterClass else return stringClass end if;
        NAMESPACE do return namespaceClass;
        COMPOUNDATTRIBUTE do return attributeClass;
        CLASS do return classClass;
        METHODCLOSURE do return functionClass;
        PROTOTYPE do return prototypeClass;
        INSTANCE do return o.type
    end case
end proc;
```

hasType

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

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

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

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

```
proc relaxedHasType(o: OBJECT, c: CLASS): BOOLEAN
    t: CLASS ← objectType(o);
    return isAncestor(c, t) or (o = null and not c.primitive)
end proc;
```

toBoolean

toBoolean(*o*) coerces an object *o* to a Boolean.

```
proc toBoolean(o: OBJECT): BOOLEAN
    case o of
        UNDEFINED ∪ NULL do return false;
        BOOLEAN do return o;
        FLOAT64 do return o ∉ {+zero, –zero, NaN};
        STRING do return o ≠ "";
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE do return true;
        INSTANCE do ????
    end case
end proc;
```

toNumber

toNumber(*o*) coerces an object *o* to a number.

```
proc toNumber(o: OBJECT): FLOAT64
    case o of
        UNDEFINED do return NaN;
        NULL ∪ {false} do return +zero;
        {true} do return 1.0;
        FLOAT64 do return o;
        STRING do ????;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE do throw typeError;
        PROTOTYPE ∪ INSTANCE do ????
    end case
end proc;
```

toString

toString(*o*) coerces an object *o* to a string.

```
proc toString(o: OBJECT): STRING
    case o of
        UNDEFINED do return "undefined";
        NULL do return "null";
        {false} do return "false";
        {true} do return "true";
        FLOAT64 do ????;
        STRING do return o;
        NAMESPACE do ????;
        COMPOUNDATTRIBUTE do ????;
        CLASS do ????;
        METHODCLOSURE do ????;
        PROTOTYPE ∪ INSTANCE do ????
    end case
end proc;
```

toPrimitive

```
proc toPrimitive(o: OBJECT, hint: OBJECT): OBJECT
    case o of
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING do return o;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do
            return toString(o)
    end case
end proc;
```

unaryPlus

unaryPlus(*o*) returns the value of the unary expression +*o*.

```
proc unaryPlus(a: OBJOPTIONALLIMIT): OBJECT
    return unaryDispatch(plusTable, null, a, ARGUMENTLIST⟨[], {}⟩)
end proc;
```

unaryNot

unaryNot(*o*) returns the value of the unary expression !*o*.

```
proc unaryNot(a: OBJECT): OBJECT
    return not toBoolean(a)
end proc;
```

Attributes

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

```
proc combineAttributes(a: ATTRIBUTENOTFALSE, b: ATTRIBUTE): ATTRIBUTE
    if b = false then return false
    elsif a = true then return b
    elsif b = true then return a
    elsif a ∈ NAMESPACE then
        if a = b then return a
        elsif b ∈ NAMESPACE then
            return COMPOUNDATTRIBUTE⟨{a, b}, false, null, false, false, null, null, false, false⟩
        else
            return COMPOUNDATTRIBUTE⟨b.namespaces ∪ {a}, b.local, b.extend, b.enumerable, b.dynamic,
                b.memberMod, b.overrideMod, b.prototype, b.unused⟩
        end if
    elsif b ∈ NAMESPACE then
        return COMPOUNDATTRIBUTE⟨a.namespaces ∪ {b}, a.local, a.extend, a.enumerable, a.dynamic,
            a.memberMod, a.overrideMod, a.prototype, a.unused⟩
    else
        Both a and b are compound attributes. Ensure that they have no duplicate or conflicting contents other than
            namespaces.
        if (a.local and b.local) or (a.extend ≠ null and b.extend ≠ null) or (a.enumerable and b.enumerable) or
            (a.dynamic and b.dynamic) or (a.memberMod ≠ null and b.memberMod ≠ null) or
            (a.overrideMod ≠ null and b.overrideMod ≠ null) or (a.prototype and b.prototype) or
            (a.unused and b.unused) then
            throw typeError
        else
            return COMPOUNDATTRIBUTE⟨a.namespaces ∪ b.namespaces, a.local or b.local,
                a.extend ≠ null ? a.extend : b.extend, a.enumerable or b.enumerable, a.dynamic or b.dynamic,
                a.memberMod ≠ null ? a.memberMod : b.memberMod,
                a.overrideMod ≠ null ? a.overrideMod : b.overrideMod, a.prototype or b.prototype,
                a.unused or b.unused⟩
        end if
    end if
end proc;
```

Objects with Limits

getObject(*o*) returns *o* without its limit, if any.

```
proc getObject(o: OBJOPTIONALLIMIT): OBJECT
    case o of
        OBJECT do return o;
        LIMITEDINSTANCE do return o.instance
    end case
end proc;
```

getObjectLimit(*o*) returns *o*'s limit or **null** if none is provided.

```
proc getObjectLimit(o: OBJOPTIONALLIMIT): CLASSOPT
    case o of
        OBJECT do return null;
        LIMITEDINSTANCE do return o.limit
    end case
end proc;
```

References

If *r* is an OBJECT, *readReference*(*r*) returns it unchanged. If *r* is a REFERENCE, this function reads *r* and returns the result.

```
proc readReference(r: OBJORREF): OBJECT
    case r of
        OBJECT do return r;
        VARIABLEREFERENCE do return readVariable(r.env, r.partialName);
        DOTREFERENCE do return readProperty(r.base, r.propName);
        BRACKETREFERENCE do return unaryDispatch(bracketReadTable, null, r.base, r.args)
    end case
end proc;
```

readRefWithLimit(*r*) reads the reference, if any, inside *r* and returns the result, retaining the same limit as *r*. If *r* has a limit *limit*, then the object read from the reference is checked to make sure that it is an instance of *limit* or one of its descendants.

```
proc readRefWithLimit(r: OBJORREFOPTIONALLIMIT): OBJOPTIONALLIMIT
    case r of
        OBJORREF do return readReference(r);
        LIMITEDOBJORREF do
            o: OBJECT ← readReference(r.ref);
            limit: CLASS ← r.limit;
            if o = null then return null end if;
            if o ∉ INSTANCE or not hasType(o, limit) then throw typeError end if;
            return LIMITEDINSTANCE⟨o, limit⟩
    end case
end proc;
```

If *r* is a reference, *writeReference*(*r*, *o*) writes *o* into *r*. An error occurs if *r* is not a reference. *r*'s limit, if any, is ignored.

```
proc writeReference(r: OBJORREFOPTIONALLIMIT, o: OBJECT)
    case r of
        OBJECT do throw referenceError;
        VARIABLEREFERENCE do writeVariable(r.env, r.partialName, o);
        DOTREFERENCE do writeProperty(r.base, r.propName, o);
        BRACKETREFERENCE do
            args: ARGUMENTLIST ← ARGUMENTLIST⟨[o] ⊕ r.args.positional, r.args.named⟩;
            unaryDispatch(bracketWriteTable, null, r.base, args);
        LIMITEDOBJORREF do writeReference(r.ref, o)
    end case
end proc;
```

If *r* is a REFERENCE, *deleteReference*(*r*) deletes it. If *r* is an OBJECT, this function signals an error.

```
proc deleteReference(r: OBJORREF): OBJECT
    case r of
        OBJECT do throw referenceError;
        VARIABLEREFERENCE do return deleteVariable(r.env, r.partialName);
        DOTREFERENCE do return deleteProperty(r.base, r.propName);
        BRACKETREFERENCE do
            return unaryDispatch(bracketDeleteTable, null, r.base, r.args)
    end case
end proc;
```

referenceBase(*r*) returns REFERENCE *r*'s base or**null** if there is none. *r*'s limit and the base's limit, if any, are ignored.

```
proc referenceBase(r: OBJORREFOPTIONALLIMIT): OBJECT
    case r of
        ~~OBJECT do return null;~~
        ~~REFERENCE do return getObject(r.base);~~OBJECT ∪ VARIABLEREFERENCE do return null;
        DOTREFERENCE ∪ BRACKETREFERENCE do return getObject(r.base);
        LIMITEDOBJORREF do return referenceBase(r.ref)
    end case
end proc;
```

Slots

```
proc findSlot(o: OBJECT, id: SLOTID): SLOT
    o must be an INSTANCE;
    matchingSlots: SLOT{} ← {s | ∀s ∈ o.slots such that s.id = id};
    Note that exactly one slot should match: |matchingSlots| = 1;
    return eltof matchingSlots
end proc;
```

Member Lookup

Reading a Property

```
proc readProperty(ol: OBJOPTIONALLIMIT, pn: PARTIALNAME): OBJECT
    ns: NAMESPACE ← resolveObjectNamespace(getObject(ol), pn, {read, readWrite});
    qn: QUALIFIEDNAME ← QUALIFIEDNAME⟨ns, pn.name⟩;
    return readQualifiedProperty(ol, qn, false)
end proc;

proc readQualifiedProperty(ol: OBJOPTIONALLIMIT, qn: QUALIFIEDNAME, indexableOnly: BOOLEAN): OBJECT
    d: MEMBERDATAOPT ~~← null~~;
    case ol of
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪
                METHODCLOSURE ∪ FIXEDINSTANCE do
            d ← mostSpecificMember(objectType(ol), false, qn, {read, readWrite}, indexableOnly);
        CLASS do d ← mostSpecificMember(ol, true, qn, {read, readWrite}, indexableOnly);
        PROTOTYPE do
            if qn.namespace ≠ publicNamespace then throw propertyNotFoundError
            elsif some p ∈ ol.dynamicProperties satisfies p.name = qn.name then
                return p.value
            elsif ol.parent = null then return undefined
            else return readQualifiedProperty(ol.parent, qn, indexableOnly)
            end if;
        DYNAMICINSTANCE do
            d ← mostSpecificMember(objectType(ol), false, qn, {read, readWrite}, indexableOnly);
            if d = null and qn.namespace = publicNamespace then
                if some p ∈ ol.dynamicProperties satisfies p.name = qn.name then
                    return p.value
                else return undefined
                end if
            end if;
        LIMITEDINSTANCE do
            d ← mostSpecificMember(ol.limit.super, false, qn, {read, readWrite}, indexableOnly)
    end case;
    o: OBJECT ← getObject(ol);
    case d of
        {null} do throw propertyNotFoundError;
        GLOBALSLOT do return d.value;
        SLOTID do return findSlot(o, d).value;
        METHOD do return METHODCLOSURE⟨o, d⟩;
        ACCESSOR do return d.f.call(o, ARGUMENTLIST⟨[], {}⟩)
    end case
end proc;
```

Writing a Property

```
proc writeProperty(ol: OBJOPTIONALLIMIT, pn: PARTIALNAME, newValue: OBJECT)
    ns: NAMESPACE ← resolveObjectNamespace(getObject(ol), pn, {write, readWrite});
    qn: QUALIFIEDNAME ← QUALIFIEDNAME⟨ns, pn.name⟩;
    writeQualifiedProperty(ol, qn, false, newValue)
end proc;

proc writeQualifiedProperty(ol: OBJOPTIONALLIMIT, qn: QUALIFIEDNAME, indexableOnly: BOOLEAN,
        newValue: OBJECT)
    d: MEMBERDATAOPT ~~← null~~;
    case ol of
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪
                METHODCLOSURE do
            throw propertyNotFoundError;
        CLASS do
            d ← mostSpecificMember(ol, true, qn, {write, readWrite}, indexableOnly);
        PROTOTYPE do
            if qn.namespace ≠ publicNamespace then throw propertyNotFoundError end if;
            writeDynamicProperty(ol, qn.name, newValue);
            return;
        FIXEDINSTANCE do
            d ← mostSpecificMember(objectType(ol), false, qn, {write, readWrite}, indexableOnly);
        DYNAMICINSTANCE do
            d ← mostSpecificMember(objectType(ol), false, qn, {write, readWrite}, indexableOnly);
            if d = null and qn.namespace = publicNamespace then
                d ← mostSpecificMember(objectType(ol), false, qn, {read, write, readWrite}, indexableOnly);
                if d ≠ null then throw propertyNotFoundError end if;
                writeDynamicProperty(ol, qn.name, newValue);
                return
            end if;
        LIMITEDINSTANCE do
            d ← mostSpecificMember(ol.limit.super, false, qn, {write, readWrite}, indexableOnly)
    end case;
    o: OBJECT ← getObject(ol);
    d cannot be a METHOD at this point because all METHOD properties are read-only;
    case d of
        {null} do throw propertyNotFoundError;
        GLOBALSLOT do
            if not relaxedHasType(newValue, d.type) then throw typeError end if;
            d.value ← newValue;
        SLOTID do
            if not relaxedHasType(newValue, d.type) then throw typeError end if;
            findSlot(o, d).value ← newValue;
        ~~METHOD do ⊥;~~
        ACCESSOR do
            if not relaxedHasType(newValue, d.type) then throw typeError end if;
            d.f.call(o, ARGUMENTLIST⟨[newValue], {}⟩)
    end case
end proc;

proc writeDynamicProperty(o: PROTOTYPE ∪ DYNAMICINSTANCE, name: STRING, newValue: OBJECT)
    if some p ∈ o.dynamicProperties satisfies p.name = name then p.value ← newValue
    else
        o.dynamicProperties ← o.dynamicProperties ∪ {new DYNAMICPROPERTY⟨⟨name, newValue⟩⟩}
    end if
end proc;
```

```
proc deleteProperty(o: OBJOPTIONALLIMIT, pn: PARTIALNAME): BOOLEAN
    ????
end proc;

proc deleteQualifiedProperty(o: OBJECT, name: STRING, ns: NAMESPACE, indexableOnly: BOOLEAN): BOOLEAN
    ????
end proc;

proc mostSpecificMember(c: CLASSOPT, global: BOOLEAN, qn: QUALIFIEDNAME, accesses: MEMBERACCESS{},
        indexableOnly: BOOLEAN): MEMBERDATAOPT
    if c = null then return null end if;
    qn2: QUALIFIEDNAME ← qn;
    members: MEMBER{} ← global ? c.globalMembers : c.instanceMembers;
    if some m ∈ members satisfies m.access ∈ accesses and ~~qn.name = m.name and~~
            ~~qn.namespace ∈ m.namespaces and~~qn.name = m.partialName.name and
            qn.namespace ∈ m.partialName.namespaces and (not indexableOnly or m.indexable) then
        d: MEMBERDATA ∪ NAMESPACE ← m.data;
        if d ∉ NAMESPACE then return d end if;
        qn2 ← QUALIFIEDNAME⟨d, qn.name⟩
    end if;
    return mostSpecificMember(c.super, global, qn2, accesses, indexableOnly)
end proc;

proc resolveMemberNamespace(c: CLASS, global: BOOLEAN, pn: PARTIALNAME, accesses: MEMBERACCESS{}):
        NAMESPACEOPT
    s: CLASSOPT ← c.super;
    if s ≠ null then
        ns: NAMESPACEOPT ← resolveMemberNamespace(s, global, pn, accesses);
        if ns ≠ null then return ns end if
    end if;
    members: MEMBER{} ← global ? c.globalMembers : c.instanceMembers;
    matches: MEMBER{} ← {m | ∀m ∈ members such that
            ~~m.access ∈ accesses and pn.name = m.name and pn.namespaces ∩ m.namespaces ≠ {}};~~m.access ∈ acc
            esses and pn.name = m.partialName.name and pn.namespaces ∩ m.partialName.namespaces ≠ {}};
    if matches ≠ {} then
        if |matches| > 1 then throw propertyNotFoundError end if;
        ~~matchingNamespaces: NAMESPACE{} ← pn.namespaces ∩ (eltof matches).namespaces;~~matchingNamespaces:
                NAMESPACE{} ← pn.namespaces ∩ (eltof matches).partialName.namespaces;
        return eltof matchingNamespaces
    end if;
    return null
end proc;

proc resolveObjectNamespace(o: OBJECT, pn: PARTIALNAME, accesses: MEMBERACCESS{}): NAMESPACE
    ns: NAMESPACEOPT ← o ∈ CLASS ? resolveMemberNamespace(o, true, pn, accesses) :
            resolveMemberNamespace(objectType(o), false, pn, accesses);
    if ns ≠ null then return ns end if;
    if publicNamespace ∈ pn.namespaces then return publicNamespace end if;
    throw propertyNotFoundError
end proc;
```

Operator Dispatch

Unary Operators

unaryDispatch(*table*, *this*, *operand*, *args*) dispatches the unary operator described by *table* applied to the `this` value *this*, the operand *operand*, and zero or more positional and/or named arguments *args*. If *operand* has a non-**null** limit class, lookup is restricted to operators defined on the proper ancestors of that limit.

```
proc unaryDispatch(table: UNARYMETHOD{}, this: OBJECT, operand: OBJOPTIONALLIMIT, args: ARGUMENTLIST):
        OBJECT
    applicableOps: UNARYMETHOD{} ← {m | ∀m ∈ table such that limitedHasType(operand, m.operandType)};
    if some best ∈ applicableOps satisfies
            (every m2 ∈ applicableOps satisfies isAncestor(m2.operandType, best.operandType)) then
        return best.f(this, getObject(operand), args)
    end if;
    throw propertyNotFoundError
end proc;
```

limitedHasType(*o*, *c*) returns **true** if *o* is a member of class *c* with the added condition that, if *o* has a non-**null** limit class *limit*, *c* is a proper ancestor of *limit*.

```
proc limitedHasType(o: OBJOPTIONALLIMIT, c: CLASS): BOOLEAN
    a: OBJECT ← getObject(o);
    limit: CLASSOPT ← getObjectLimit(o);
    if hasType(a, c) then
        if limit = null then return true else return isProperAncestor(c, limit) end if
    else return false
    end if
end proc;
```

Binary Operators

isBinaryDescendant(*m1*, *m2*) is **true** if *m1* is at least as specific as *m2* as defined by the procedure below.

```
proc isBinaryDescendant(m1: BINARYMETHOD, m2: BINARYMETHOD): BOOLEAN
    return isAncestor(m2.leftType, m1.leftType) and isAncestor(m2.rightType, m1.rightType)
end proc;
```

binaryDispatch(*table*, *left*, *right*) dispatches the binary operator described by *table* applied to the operands *left* and *right*. If *left* has a non-**null** limit *leftLimit*, the lookup is restricted to operator definitions with an ancestor of *leftLimit* for the left operand. Similarly, if *right* has a non-**null** limit *rightLimit*, the lookup is restricted to operator definitions with an ancestor of *rightLimit* for the right operand.

```
proc binaryDispatch(table: BINARYMETHOD{}, left: OBJOPTIONALLIMIT, right: OBJOPTIONALLIMIT): OBJECT
    applicableOps: BINARYMETHOD{} ← {m | ∀m ∈ table such that
            limitedHasType(left, m.leftType) and limitedHasType(right, m.rightType)};
    if some best ∈ applicableOps satisfies (every m2 ∈ applicableOps satisfies isBinaryDescendant(best, m2)) then
        return best.f(getObject(left), getObject(right))
    end if;
    throw propertyNotFoundError
end proc;
```

~~Validation Environments~~

```
~~tuple VALIDATIONENV~~
    ~~enclosingClass: CLASSOPT,~~
    ~~labels: STRING[],~~
    ~~canReturn: BOOLEAN,~~
    ~~constants: DEFINITION[]~~
~~end tuple~~;
```

initialValidationEnv: VALIDATIONENV = VALIDATIONENV⟨**null**, [], **false**, []⟩;

Return a VALIDATIONENV with label *label* prepended to *v*.
proc *addLabel*(*v*: VALIDATIONENV, *label*: STRING): VALIDATIONENV
return VALIDATIONENV⟨*v*.enclosingClass, [*label*] ⊕ *v*.labels, *v*.canReturn, *v*.constants⟩
end proc;

Return **true** if this code is inside a class body.
proc *insideClass*(*v*: VALIDATIONENV): BOOLEAN
return *v*.**enclosingClass** ≠ null**Contexts**

addBreakLabel(*c*, *l*) returns a new CONTEXT that is the same as *c* except that it includes the label *l* in the context's set of labels that are valid targets for a `break` statement.
proc *addBreakLabel*(*c*: CONTEXT, *l*: LABEL): CONTEXT
return CONTEXT⟨*c*.strict, *c*.unchecked, *c*.insideFunction, *c*.privateNamespace, *c*.openNamespaces,
c.breakLabels ∪ {*l*}, *c*.continueLabels⟩
end proc;

addContinueLabels(*c*, *ls*) returns a new CONTEXT that is the same as *c* except that it includes the labels *ls* in the context's set of labels that are valid targets for a `continue` statement.
proc *addContinueLabels*(*c*: CONTEXT, *ls*: LABEL{}): CONTEXT
return CONTEXT⟨*c*.strict, *c*.unchecked, *c*.insideFunction, *c*.privateNamespace, *c*.openNamespaces,
c.breakLabels, *c*.continueLabels ∪ *ls*⟩
end proc;

Environments

Static Environments

Dynamic Environments

record DYNAMICENV
parent: DYNAMICENVOPT,
enclosingClass: CLASSOPT,
readerDefinitions: DEFINITION[],
readerPassthroughs: QUALIFIEDNAME[],
writerDefinitions: DEFINITION[],
writerPassthroughs: QUALIFIEDNAME[]
end record;

DYNAMICENVOPT = NULL ∪ DYNAMICENV;

If the DYNAMICENVENVIRONMENT is from within a class's body, return that class; otherwise, throw an exception.
proc *lexicalClass*(*e*: DYNAMICENV): CLASS*lexicalClass*(*e*: ENVIRONMENT): CLASS
????
end proc;

proc *dynamicEnvUses*(*e*: DYNAMICENV): NAMESPACE{}*readVariable*(*e*: ENVIRONMENT, *pn*: PARTIALNAME): OBJECT
????
end proc;

initialDynamicEnv: DYNAMICENV = **new** DYNAMICENV⟨⟨**null**, **null**, [], [], [], []⟩⟩;

~~**tuple** DEFINITION~~
~~name: QUALIFIEDNAME,~~
~~type: CLASS,~~
~~data: SLOT ∪ OBJECT ∪ ACCESSOR~~
~~**end tuple**;~~

~~**proc** *lookupVariable*(*e*: DYNAMICENV, *name*: STRING, *internalIsNamespace*: BOOLEAN): OBJORREF~~**proc** *writeVariable*(*e*: ENVIRONMENT, *pn*: PARTIALNAME, *newValue*: OBJECT)
????
end proc;

proc ~~*lookupQualifiedVariable*(*e*: DYNAMICENV, *namespace*: NAMESPACE, *name*: STRING): OBJORREF~~*deleteVariable*(*e*: ENVIRONMENT, *pn*: PARTIALNAME): BOOLEAN
????
end proc;

Return the value of `this`. Throw an exception if there is no `this` defined.

proc ~~*lookupThis*(*e*: DYNAMICENV): OBJECT~~*lookupThis*(*e*: ENVIRONMENT): OBJECT
????
end proc;

Expressions

Syntax

β ∈ {allowIn, noIn}

Terminal Actions

Name[**Identifier**]: STRING;
Eval[**Number**]: FLOAT64;
Eval[**String**]: STRING;

Identifiers

Syntax

Identifier ⇒
    Identifier
  | **get**
  | **set**
  | **exclude**
  | **include**
  | **named**

Semantics

Name[*Identifier*]: STRING;
Name[*Identifier* ⇒ **Identifier**] = *Name*[**Identifier**];
Name[*Identifier* ⇒ **get**] = "get";
Name[*Identifier* ⇒ **set**] = "set";
Name[*Identifier* ⇒ **exclude**] = "exclude";
Name[*Identifier* ⇒ **include**] = "include";
Name[*Identifier* ⇒ **named**] = "named";

Qualified Identifiers

Syntax

Qualifier ⇒
    Identifier
  | **`public`**
  | **`private`**

SimpleQualifiedIdentifier ⇒
    Identifier
  | *Qualifier* **`::`** *Identifier*

ExpressionQualifiedIdentifier ⇒ *ParenExpression* **`::`** *Identifier*

QualifiedIdentifier ⇒
    SimpleQualifiedIdentifier
  | *ExpressionQualifiedIdentifier*

~~**Validation**~~

~~**proc** *Validate*[*Qualifier*] (*v*: VALIDATIONENV)~~
~~[*Qualifier* ⇒ *Identifier*] **do** ????;~~
~~[*Qualifier* ⇒ **`public`**] **do nothing**;~~
~~[*Qualifier* ⇒ **`private`**] **do if not** *insideClass*(*v*) **then throw syntaxError end if**~~
~~**end proc**;~~

~~**proc** *Validate*[*SimpleQualifiedIdentifier*] (*v*: VALIDATIONENV)~~
~~[*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do nothing**;~~

~~**[*SimpleQualifiedIdentifier* ⇒ *Qualifier* : : *Identifier*]** do *Validate*~~**[*Qualifier*](*v*)**Validation and Evaluation**

proc *Validate*[*Qualifier*] (*c*: CONTEXT, *e*: STATICENV): NAMESPACE
[*Qualifier* ⇒ *Identifier*] **do**
a: OBJECT ← *readVariable*(*e*, PARTIALNAME⟨*c*.openNamespaces, *Name*[*Identifier*]⟩);
if *a* ∉ NAMESPACE **then throw typeError end if**;
return *a*;
[*Qualifier* ⇒ **`public`**] **do return** *publicNamespace*;
[*Qualifier* ⇒ **`private`**] **do**
p: NAMESPACEOPT ← *c*.privateNamespace;
if *p* = **null then throw syntaxError end if**;
return *p*
end proc;

~~**proc** *Validate*[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **`::`** *Identifier*] (*v*: VALIDATIONENV)~~
~~*Validate*[*ParenExpression*](*v*);~~
????*Name*[*SimpleQualifiedIdentifier*]: PARTIALNAME;

proc *Validate*[*SimpleQualifiedIdentifier*] (*c*: CONTEXT, *e*: STATICENV)
[*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do**
Name[*SimpleQualifiedIdentifier*] ← PARTIALNAME⟨*c*.openNamespaces, *Name*[*Identifier*]⟩;
[*SimpleQualifiedIdentifier* ⇒ *Qualifier* **`::`** *Identifier*] **do**
q: NAMESPACE ← *Validate*[*Qualifier*](*c*, *e*);
Name[*SimpleQualifiedIdentifier*] ← PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

Validate[*QualifiedIdentifier*]: VALIDATIONENV → ();
Validate[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] = *Validate*[*SimpleQualifiedIdentifier*];
Validate[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] = *Validate*[*ExpressionQualifiedIdentifier*];

Evaluation

proc *Eval*[*Qualifier*] (*e*: DYNAMICENV): NAMESPACE
[*Qualifier* ⇒ *Identifier*] **do**
a: OBJECT ← *readReference*(*lookupVariable*(*e*, *Name*[*Identifier*], **true**));
if *a* ∉ NAMESPACE **then throw typeError end if**;
return *a*;
[*Qualifier* ⇒ **public**] **do return** *publicNamespace*;
[*Qualifier* ⇒ **private**] **do**
q: CLASSOPT ← *e*.enclosingClass;
if *q* = **null then** ⊥ **end if**;
return *q*.privateNamespace*Name*[*ExpressionQualifiedIdentifier*]: PARTIALNAME;

proc *Validate*[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **::** *Identifier*] (*c*: CONTEXT, *e*: STATICENV)
Validate[*ParenExpression*](*c*, *e*);
q: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
if *q* ∉ NAMESPACE **then throw typeError end if**;
Name[*ExpressionQualifiedIdentifier*] ← PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

proc *Eval*[*SimpleQualifiedIdentifier*] (*e*: DYNAMICENV): OBJORREF
[*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do**
return *lookupVariable*(*e*, *Name*[*Identifier*], **false**);
[*SimpleQualifiedIdentifier* ⇒ *Qualifier* **::** *Identifier*] **do**
q: NAMESPACE ← *Eval*[*Qualifier*](*e*);
return *lookupQualifiedVariable*(*e*, *q*, *Name*[*Identifier*])*Name*[*QualifiedIdentifier*]: PARTIALNAME;

proc *Validate*[*QualifiedIdentifier*] (*c*: CONTEXT, *e*: STATICENV)
[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] **do**
Validate[*SimpleQualifiedIdentifier*](*c*, *e*);
Name[*QualifiedIdentifier*] ← *Name*[*SimpleQualifiedIdentifier*];
[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] **do**
Validate[*ExpressionQualifiedIdentifier*](*c*, *e*);
Name[*QualifiedIdentifier*] ← *Name*[*ExpressionQualifiedIdentifier*]
end proc;

proc *Eval*[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **::** *Identifier*] (*e*: DYNAMICENV): OBJORREF
q: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
if *q* ∉ NAMESPACE **then throw typeError end if**;
return *lookupQualifiedVariable*(*e*, *q*, *Name*[*Identifier*])
end proc;

Eval[*QualifiedIdentifier*]: DYNAMICENV → OBJORREF;
Eval[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] = *Eval*[*SimpleQualifiedIdentifier*];
Eval[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] = *Eval*[*ExpressionQualifiedIdentifier*];

proc *Name*[*SimpleQualifiedIdentifier*] (*e*: DYNAMICENV): PARTIALNAME
[*SimpleQualifiedIdentifier* ⇒ *Identifier*] **do**
return PARTIALNAME⟨*dynamicEnvUses*(*e*), *Name*[*Identifier*]⟩;

[*SimpleQualifiedIdentifier* ⇒ *Qualifier* **::** *Identifier*] **do**
q: NAMESPACE ← *Eval*[*Qualifier*](*e*);
return PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

proc *Name*[*ExpressionQualifiedIdentifier* ⇒ *ParenExpression* **::** *Identifier*] (*e*: DYNAMICENV): PARTIALNAME
q: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
if *q* ∉ NAMESPACE **then throw typeError end if**;
return PARTIALNAME⟨{*q*}, *Name*[*Identifier*]⟩
end proc;

Name[*QualifiedIdentifier*]: DYNAMICENV → PARTIALNAME;
Name[*QualifiedIdentifier* ⇒ *SimpleQualifiedIdentifier*] = *Name*[*SimpleQualifiedIdentifier*];
Name[*QualifiedIdentifier* ⇒ *ExpressionQualifiedIdentifier*] = *Name*[*ExpressionQualifiedIdentifier*];

Unit Expressions

Syntax

UnitExpression ⇒
ParenListExpression
| **Number** [no line break] **String**
| *UnitExpression* [no line break] **String**

Validation

proc *Validate*[*UnitExpression*] (*v*: VALIDATIONENV)
[*UnitExpression* ⇒ *ParenListExpression*] **do** *Validate*[*ParenListExpression*](*v*);**proc** *Validate*[*UnitExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*UnitExpression* ⇒ *ParenListExpression*] **do** *Validate*[*ParenListExpression*](*c*, *e*);
[*UnitExpression* ⇒ **Number** [no line break] **String**] **do** ????;
[*UnitExpression* ⇒ *UnitExpression* [no line break] **String**] **do** ????
end proc;

Evaluation

proc *Eval*[*UnitExpression*] (*e*: DYNAMICENV): OBJORREF**proc** *Eval*[*UnitExpression*] (*e*: ENVIRONMENT): OBJORREF
[*UnitExpression* ⇒ *ParenListExpression*] **do return** *Eval*[*ParenListExpression*](*e*);
[*UnitExpression* ⇒ **Number** [no line break] **String**] **do** ????;
[*UnitExpression* ⇒ *UnitExpression* [no line break] **String**] **do** ????
end proc;

Primary Expressions

Syntax

PrimaryExpression ⇒
  null
| **true**
| **false**
| **public**
| **Number**
| **String**
| **this**
| **RegularExpression**
| *UnitExpression*
| *ArrayLiteral*
| *ObjectLiteral*
| *FunctionExpression*

ParenExpression ⇒ **(** *AssignmentExpression*$^{\text{allowIn}}$ **)**

ParenListExpression ⇒
  ParenExpression
| **(** *ListExpression*$^{\text{allowIn}}$ **,** *AssignmentExpression*$^{\text{allowIn}}$ **)**

Validation

~~**proc** *Validate*[*PrimaryExpression*] (*v*: VALIDATIONENV)~~**proc** *Validate*[*PrimaryExpression*] (*c*: CONTEXT, *e*: STATICENV)
  [*PrimaryExpression* ⇒ **null**] **do nothing**;
  [*PrimaryExpression* ⇒ **true**] **do nothing**;
  [*PrimaryExpression* ⇒ **false**] **do nothing**;
  [*PrimaryExpression* ⇒ **public**] **do nothing**;
  [*PrimaryExpression* ⇒ **Number**] **do nothing**;
  [*PrimaryExpression* ⇒ **String**] **do nothing**;
  [*PrimaryExpression* ⇒ **this**] **do** ????;
  [*PrimaryExpression* ⇒ **RegularExpression**] **do nothing**;
  ~~[*PrimaryExpression* ⇒ *UnitExpression*] **do** *Validate*[*UnitExpression*](*v*);~~[*PrimaryExpression* ⇒ *UnitExpression*] **do** *Validate*[*UnitExpression*](*c*, *e*);
  [*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *ObjectLiteral*] **do** ????;
  ~~[*PrimaryExpression* ⇒ *FunctionExpression*] **do** *Validate*[*FunctionExpression*](*v*)~~[*PrimaryExpression* ⇒ *FunctionExpression*] **do** *Validate*[*FunctionExpression*](*c*, *e*)
end proc;

~~*Validate*[*ParenExpression* ⇒ **(** *AssignmentExpression*$^{\text{allowIn}}$ **)**]: VALIDATIONENV → ()~~*Validate*[*ParenExpression* ⇒ **(** *AssignmentExpression*$^{\text{allowIn}}$ **)**]: CONTEXT × STATICENV → () = *Validate*[*AssignmentExpression*$^{\text{allowIn}}$];

~~**proc** *Validate*[*ParenListExpression*] (*v*: VALIDATIONENV)~~
~~[*ParenListExpression* ⇒ *ParenExpression*] **do** *Validate*[*ParenExpression*](*v*);~~**proc** *Validate*[*ParenListExpression*] (*c*: CONTEXT, *e*: STATICENV)
  [*ParenListExpression* ⇒ *ParenExpression*] **do** *Validate*[*ParenExpression*](*c*, *e*);
  [*ParenListExpression* ⇒ **(** *ListExpression*$^{\text{allowIn}}$ **,** *AssignmentExpression*$^{\text{allowIn}}$ **)**] **do**
    ~~*Validate*[*ListExpression*$^{\text{allowIn}}$](*v*);~~
    ~~*Validate*[*AssignmentExpression*$^{\text{allowIn}}$](*v*)~~*Validate*[*ListExpression*$^{\text{allowIn}}$](*c*, *e*);
    Validate[*AssignmentExpression*$^{\text{allowIn}}$](*c*, *e*)
end proc;

Evaluation

~~**proc** *Eval*[*PrimaryExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*PrimaryExpression*] (*e*: ENVIRONMENT): OBJORREF
  [*PrimaryExpression* ⇒ **`null`**] **do return null**;
  [*PrimaryExpression* ⇒ **`true`**] **do return true**;
  [*PrimaryExpression* ⇒ **`false`**] **do return false**;
  [*PrimaryExpression* ⇒ **`public`**] **do return** *publicNamespace*;
  [*PrimaryExpression* ⇒ **Number**] **do return** *Eval*[**Number**];
  [*PrimaryExpression* ⇒ **String**] **do return** *Eval*[**String**];
  [*PrimaryExpression* ⇒ **`this`**] **do return** *lookupThis*(*e*);
  [*PrimaryExpression* ⇒ **RegularExpression**] **do** ????;
  [*PrimaryExpression* ⇒ *UnitExpression*] **do return** *Eval*[*UnitExpression*](*e*);
  [*PrimaryExpression* ⇒ *ArrayLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *ObjectLiteral*] **do** ????;
  [*PrimaryExpression* ⇒ *FunctionExpression*] **do return** *Eval*[*FunctionExpression*](*e*)
end proc;

~~*Eval*[*ParenExpression* ⇒ **(** *AssignmentExpression*$^{\text{allowIn}}$ **)**]: DYNAMICENV → OBJORREF~~*Eval*[*ParenExpression* ⇒ **(** *AssignmentExpression*$^{\text{allowIn}}$ **)**]: ENVIRONMENT → OBJORREF = *Eval*[*AssignmentExpression*$^{\text{allowIn}}$];

~~**proc** *Eval*[*ParenListExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*ParenListExpression*] (*e*: ENVIRONMENT): OBJORREF
  [*ParenListExpression* ⇒ *ParenExpression*] **do return** *Eval*[*ParenExpression*](*e*);
  [*ParenListExpression* ⇒ **(** *ListExpression*$^{\text{allowIn}}$ **,** *AssignmentExpression*$^{\text{allowIn}}$ **)**] **do**
    readReference(*Eval*[*ListExpression*$^{\text{allowIn}}$](*e*));
    return *readReference*(*Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*))
end proc;

~~**proc** *EvalAsList*[*ParenListExpression*] (*e*: DYNAMICENV): OBJECT[]~~**proc** *EvalAsList*[*ParenListExpression*] (*e*: ENVIRONMENT): OBJECT[]
  [*ParenListExpression* ⇒ *ParenExpression*] **do**
    elt: OBJECT ← *readReference*(*Eval*[*ParenExpression*](*e*));
    return [*elt***]**;
  [*ParenListExpression* ⇒ **(** *ListExpression*$^{\text{allowIn}}$ **,** *AssignmentExpression*$^{\text{allowIn}}$ **)**] **do**
    elts: OBJECT[] ← *EvalAsList*[*ListExpression*$^{\text{allowIn}}$](*e*);
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^{\text{allowIn}}$](*e*));
    return *elts* ⊕ **[***elt***]**
end proc;

Function Expressions

Syntax

FunctionExpression ⇒
    `function` *FunctionSignature Block*
  | **`function`** *Identifier FunctionSignature Block*

Validation

~~**proc** *Validate*[*FunctionExpression*] (*v*: VALIDATIONENV)~~**proc** *Validate*[*FunctionExpression*] (*c*: CONTEXT, *e*: STATICENV)
  [*FunctionExpression* ⇒ **`function`** *FunctionSignature Block*] **do** ????;
  [*FunctionExpression* ⇒ **`function`** *Identifier FunctionSignature Block*] **do** ????
end proc;

Evaluation

```
~~proc Eval[FunctionExpression] (e: DynamicEnv): ObjOrRef~~proc Eval[FunctionExpression] (e: Environment): ObjO
    rRef
  [FunctionExpression ⇒ function FunctionSignature Block] do ????;
  [FunctionExpression ⇒ function Identifier FunctionSignature Block] do ????
end proc;
```

Object Literals

Syntax

```
ObjectLiteral ⇒
    { }
  | { FieldList }

FieldList ⇒
    LiteralField
  | FieldList , LiteralField

LiteralField ⇒ FieldName : AssignmentExpression^allowIn

FieldName ⇒
    Identifier
  | String
  | Number
  | ParenExpression
```

Validation

```
proc Validate[LiteralField ⇒ FieldName : AssignmentExpression^allowIn]
      ~~(v: ValidationEnv): String{}~~(c: Context, e: StaticEnv): String{}
  ~~names: String{} ← Validate[FieldName](v);~~
  ~~Validate[AssignmentExpression^allowIn](v);~~names: String{} ← Validate[FieldName](c, e);
  Validate[AssignmentExpression^allowIn](c, e);
  return names
end proc;

~~proc Validate[FieldName] (v: ValidationEnv): String{}~~proc Validate[FieldName] (c: Context, e: StaticEnv): Strin
    g{}
  [FieldName ⇒ Identifier] do return {Name[Identifier]};
  [FieldName ⇒ String] do return {Eval[String]};
  [FieldName ⇒ Number] do ????;
  [FieldName ⇒ ParenExpression] do ????
end proc;
```

Evaluation

```
proc Eval[LiteralField ⇒ FieldName : AssignmentExpression^allowIn]
      ~~(e: DynamicEnv): NamedArgument~~(e: Environment): NamedArgument
  name: String ← Eval[FieldName](e);
  value: Object ← readReference(Eval[AssignmentExpression^allowIn](e));
  return NamedArgument⟨name, value⟩
end proc;

~~proc Eval[FieldName] (e: DynamicEnv): String~~proc Eval[FieldName] (e: Environment): String
  [FieldName ⇒ Identifier] do return Name[Identifier];
```

```
    [FieldName ⇒ String] do return Eval[String];
    [FieldName ⇒ Number] do ????;
    [FieldName ⇒ ParenExpression] do ????
  end proc;
```

Array Literals

Syntax

```
ArrayLiteral ⇒ [ ElementList ]

ElementList ⇒
      LiteralElement
    | ElementList , LiteralElement

LiteralElement ⇒
      «empty»
    | AssignmentExpression^allowIn
```

Super Expressions

Syntax

```
SuperExpression ⇒
      super
    | FullSuperExpression

FullSuperExpression ⇒ super ParenExpression
```

Validation

```
~~proc Validate[SuperExpression] (v: ValidationEnv)~~
    ~~[SuperExpression ⇒ super] do if not insideClass(v) then throw syntaxError end if;~~
    ~~[SuperExpression ⇒ FullSuperExpression] do Validate[FullSuperExpression](v)~~
~~end proc;~~

~~proc Validate[FullSuperExpression ⇒ super ParenExpression] (v: ValidationEnv)~~
    ~~if not insideClass(v) then throw syntaxError end if;~~
~~Validate[ParenExpression](v)~~proc Validate[SuperExpression] (c: Context, e: StaticEnv)
    [SuperExpression ⇒ super] do
        if c.privateNamespace = null then throw syntaxError end if;
    [SuperExpression ⇒ FullSuperExpression] do Validate[FullSuperExpression](c, e)
end proc;

proc Validate[FullSuperExpression ⇒ super ParenExpression] (c: Context, e: StaticEnv)
    if c.privateNamespace = null then throw syntaxError end if;
    Validate[ParenExpression](c, e)
end proc;
```

Evaluation

```
~~proc Eval[SuperExpression] (e: DynamicEnv): ObjOrRefOptionalLimit~~proc Eval[SuperExpression] (e: Environment): ObjOrRefOptionalLimit
    [SuperExpression ⇒ super] do
        this: Object ← lookupThis(e);
        limit: Class ← lexicalClass(e);
        return LimitedObjOrRef⟨this, limit⟩;
```

```
    [SuperExpression ⇒ FullSuperExpression] do return Eval[FullSuperExpression](e)
end proc;

proc Eval[FullSuperExpression ⇒ super ParenExpression]
        ~~(e: DynamicEnv): ObjOrRefOptionalLimit~~(e: Environment): ObjOrRefOptionalLimit
    r: ObjOrRef ← Eval[ParenExpression](e);
    limit: Class ← lexicalClass(e);
    return LimitedObjOrRef⟨r, limit⟩
end proc;
```

Postfix Expressions

Syntax

```
PostfixExpression ⇒
      AttributeExpression
    | FullPostfixExpression
    | ShortNewExpression

PostfixExpressionOrSuper ⇒
      PostfixExpression
    | SuperExpression

AttributeExpression ⇒
      SimpleQualifiedIdentifier
    | AttributeExpression MemberOperator
    | AttributeExpression Arguments

FullPostfixExpression ⇒
      PrimaryExpression
    | ExpressionQualifiedIdentifier
    | FullNewExpression
    | FullPostfixExpression MemberOperator
    | SuperExpression DotOperator
    | FullPostfixExpression Arguments
    | FullSuperExpression Arguments
    | PostfixExpressionOrSuper [no line break] ++
    | PostfixExpressionOrSuper [no line break] --

FullNewExpression ⇒
      new FullNewSubexpression Arguments
    | new FullSuperExpression Arguments

FullNewSubexpression ⇒
      PrimaryExpression
    | QualifiedIdentifier
    | FullNewExpression
    | FullNewSubexpression MemberOperator
    | SuperExpression DotOperator

ShortNewExpression ⇒
      new ShortNewSubexpression
    | new SuperExpression

ShortNewSubexpression ⇒
      FullNewSubexpression
    | ShortNewExpression
```

Validation

~~*Validate*[*PostfixExpression*]: VALIDATIONENV → ();~~*Validate*[*PostfixExpression*]: CONTEXT × STATICENV → ();
Validate[*PostfixExpression* ⇒ *AttributeExpression*] = *Validate*[*AttributeExpression*];
Validate[*PostfixExpression* ⇒ *FullPostfixExpression*] = *Validate*[*FullPostfixExpression*];
Validate[*PostfixExpression* ⇒ *ShortNewExpression*] = *Validate*[*ShortNewExpression*];

~~*Validate*[*PostfixExpressionOrSuper*]: VALIDATIONENV → ();~~*Validate*[*PostfixExpressionOrSuper*]: CONTEXT × STATICENV → ();
Validate[*PostfixExpressionOrSuper* ⇒ *PostfixExpression*] = *Validate*[*PostfixExpression*];
Validate[*PostfixExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

~~**proc** *Validate*[*AttributeExpression*] (*v*: VALIDATIONENV)~~**proc** *Validate*[*AttributeExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*AttributeExpression* ⇒ *SimpleQualifiedIdentifier*] **do**
~~*Validate*[*SimpleQualifiedIdentifier*](*v*);~~*Validate*[*SimpleQualifiedIdentifier*](*c*, *e*);
[*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *MemberOperator*] **do**
~~*Validate*[*AttributeExpression*$_1$](*v*);~~
~~*Validate*[*MemberOperator*](*v*);~~*Validate*[*AttributeExpression*$_1$](*c*, *e*);
Validate[*MemberOperator*](*c*, *e*);
[*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *Arguments*] **do**
~~*Validate*[*AttributeExpression*$_1$](*v*);~~
~~*Validate*[*Arguments*](*v*)~~
~~**end proc**~~;

~~**proc** *Validate*[*FullPostfixExpression*] (*v*: VALIDATIONENV)~~
~~[*FullPostfixExpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*v*);~~*Validate*[*AttributeExpression*$_1$](*c*, *e*);
Validate[*Arguments*](*c*, *e*)
end proc;

proc *Validate*[*FullPostfixExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*FullPostfixExpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*c*, *e*);
[*FullPostfixExpression* ⇒ *ExpressionQualifiedIdentifier*] **do**
~~*Validate*[*ExpressionQualifiedIdentifier*](*v*);~~
~~[*FullPostfixExpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*v*);~~*Validate*[*ExpressionQualifiedIdentifier*](*c*, *e*);
[*FullPostfixExpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*c*, *e*);
[*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *MemberOperator*] **do**
~~*Validate*[*FullPostfixExpression*$_1$](*v*);~~
~~*Validate*[*MemberOperator*](*v*);~~*Validate*[*FullPostfixExpression*$_1$](*c*, *e*);
Validate[*MemberOperator*](*c*, *e*);
[*FullPostfixExpression* ⇒ *SuperExpression DotOperator*] **do**
~~*Validate*[*SuperExpression*](*v*);~~
~~*Validate*[*DotOperator*](*v*);~~*Validate*[*SuperExpression*](*c*, *e*);
Validate[*DotOperator*](*c*, *e*);
[*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *Arguments*] **do**
~~*Validate*[*FullPostfixExpression*$_1$](*v*);~~
~~*Validate*[*Arguments*](*v*);~~*Validate*[*FullPostfixExpression*$_1$](*c*, *e*);
Validate[*Arguments*](*c*, *e*);
[*FullPostfixExpression* ⇒ *FullSuperExpression Arguments*] **do**
~~*Validate*[*FullSuperExpression*](*v*);~~
~~*Validate*[*Arguments*](*v*);~~*Validate*[*FullSuperExpression*](*c*, *e*);
Validate[*Arguments*](*c*, *e*);
[*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **++**] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*);~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*);

[*FullPostfixExpression* ⇒ *PostfixExpressionOrSuper* [no line break] **- -**] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*FullNewExpression*] (*v*: VALIDATIONENV)~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*)
end proc;

proc *Validate*[*FullNewExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*FullNewExpression* ⇒ **new** *FullNewSubexpression Arguments*] **do**
~~*Validate*[*FullNewSubexpression*](*v*);~~
~~*Validate*[*Arguments*](*v*);~~*Validate*[*FullNewSubexpression*](*c*, *e*);
Validate[*Arguments*](*c*, *e*);
[*FullNewExpression* ⇒ **new** *FullSuperExpression Arguments*] **do**
~~*Validate*[*FullSuperExpression*](*v*);~~
~~*Validate*[*Arguments*](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*FullNewSubexpression*] (*v*: VALIDATIONENV)~~
~~[*FullNewSubexpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*v*);~~
~~[*FullNewSubexpression* ⇒ *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*v*);~~
~~[*FullNewSubexpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*v*);~~*Validate*[*FullSuperExpression*](*c*, *e*);
Validate[*Arguments*](*c*, *e*)
end proc;

proc *Validate*[*FullNewSubexpression*] (*c*: CONTEXT, *e*: STATICENV)
[*FullNewSubexpression* ⇒ *PrimaryExpression*] **do** *Validate*[*PrimaryExpression*](*c*, *e*);
[*FullNewSubexpression* ⇒ *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*c*, *e*);
[*FullNewSubexpression* ⇒ *FullNewExpression*] **do** *Validate*[*FullNewExpression*](*c*, *e*);
[*FullNewSubexpression*$_0$ ⇒ *FullNewSubexpression*$_1$ *MemberOperator*] **do**
~~*Validate*[*FullNewSubexpression*$_1$](*v*);~~
~~*Validate*[*MemberOperator*](*v*);~~*Validate*[*FullNewSubexpression*$_1$](*c*, *e*);
Validate[*MemberOperator*](*c*, *e*);
[*FullNewSubexpression* ⇒ *SuperExpression DotOperator*] **do**
~~*Validate*[*SuperExpression*](*v*);~~
~~*Validate*[*DotOperator*](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*ShortNewExpression*] (*v*: VALIDATIONENV)~~*Validate*[*SuperExpression*](*c*, *e*);
Validate[*DotOperator*](*c*, *e*)
end proc;

proc *Validate*[*ShortNewExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*ShortNewExpression* ⇒ **new** *ShortNewSubexpression*] **do**
~~*Validate*[*ShortNewSubexpression*](*v*);~~
~~[*ShortNewExpression* ⇒ **new** *SuperExpression*] **do** *Validate*[*SuperExpression*](*v*)~~
~~**end proc**;~~

~~*Validate*[*ShortNewSubexpression*]: VALIDATIONENV → ();~~*Validate*[*ShortNewSubexpression*](*c*, *e*);
[*ShortNewExpression* ⇒ **new** *SuperExpression*] **do** *Validate*[*SuperExpression*](*c*, *e*)
end proc;

Validate[*ShortNewSubexpression*]: CONTEXT × STATICENV → ();
Validate[*ShortNewSubexpression* ⇒ *FullNewSubexpression*] = *Validate*[*FullNewSubexpression*];
Validate[*ShortNewSubexpression* ⇒ *ShortNewExpression*] = *Validate*[*ShortNewExpression*];

Evaluation

~~*Eval*[*PostfixExpression*]: DYNAMICENV → OBJORREF;~~*Eval*[*PostfixExpression*]: ENVIRONMENT → OBJORREF;
Eval[*PostfixExpression* ⇒ *AttributeExpression*] = *Eval*[*AttributeExpression*];
Eval[*PostfixExpression* ⇒ *FullPostfixExpression*] = *Eval*[*FullPostfixExpression*];
Eval[*PostfixExpression* ⇒ *ShortNewExpression*] = *Eval*[*ShortNewExpression*];

~~*Eval*[*PostfixExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~*Eval*[*PostfixExpressionOrSuper*]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
Eval[*PostfixExpressionOrSuper* ⇒ *PostfixExpression*] = *Eval*[*PostfixExpression*];
Eval[*PostfixExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

~~**proc** *Eval*[*AttributeExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*AttributeExpression*] (*e*: ENVIRONMENT): OBJORREF
[*AttributeExpression* ⇒ *SimpleQualifiedIdentifier*] **do**
~~**return** *Eval*[*SimpleQualifiedIdentifier*](*e*);~~**return** VARIABLEREFERENCE⟨*e*, *Name*[*SimpleQualifiedIdentifier*]⟩;
[*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *MemberOperator*] **do**
a: OBJECT ← *readReference*(*Eval*[*AttributeExpression*$_1$](*e*));
return *Eval*[*MemberOperator*](*e*, *a*);
[*AttributeExpression*$_0$ ⇒ *AttributeExpression*$_1$ *Arguments*] **do**
r: OBJORREF ← *Eval*[*AttributeExpression*$_1$](*e*);
f: OBJECT ← *readReference*(*r*);
base: OBJECT ← *referenceBase*(*r*);
args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
return *unaryDispatch*(*callTable*, *base*, *f*, *args*)
end proc;

~~**proc** *Eval*[*FullPostfixExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*FullPostfixExpression*] (*e*: ENVIRONMENT): OBJORREF
[*FullPostfixExpression* ⇒ *PrimaryExpression*] **do return** *Eval*[*PrimaryExpression*](*e*);
[*FullPostfixExpression* ⇒ *ExpressionQualifiedIdentifier*] **do**
~~**return** *Eval*[*ExpressionQualifiedIdentifier*](*e*);~~**return** VARIABLEREFERENCE⟨*e*, *Name*[*ExpressionQualifiedIdentifier*]⟩;
[*FullPostfixExpression* ⇒ *FullNewExpression*] **do return** *Eval*[*FullNewExpression*](*e*);
[*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *MemberOperator*] **do**
a: OBJECT ← *readReference*(*Eval*[*FullPostfixExpression*$_1$](*e*));
return *Eval*[*MemberOperator*](*e*, *a*);
[*FullPostfixExpression* ⇒ *SuperExpression DotOperator*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*SuperExpression*](*e*));
return *Eval*[*DotOperator*](*e*, *a*);
[*FullPostfixExpression*$_0$ ⇒ *FullPostfixExpression*$_1$ *Arguments*] **do**
r: OBJORREF ← *Eval*[*FullPostfixExpression*$_1$](*e*);
f: OBJECT ← *readReference*(*r*);
base: OBJECT ← *referenceBase*(*r*);
args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
return *unaryDispatch*(*callTable*, *base*, *f*, *args*);
[*FullPostfixExpression* ⇒ *FullSuperExpression Arguments*] **do**
r: OBJORREFOPTIONALLIMIT ← *Eval*[*FullSuperExpression*](*e*);
f: OBJOPTIONALLIMIT ← *readRefWithLimit*(*r*);
base: OBJECT ← *referenceBase*(*r*);
args: ARGUMENTLIST ← *Eval*[*Arguments*](*e*);
return *unaryDispatch*(*callTable*, *base*, *f*, *args*);

```
    [FullPostfixExpression ⇒ PostfixExpressionOrSuper [no line break] ++] do
        r: ObjOrRefOptionalLimit ← Eval[PostfixExpressionOrSuper](e);
        a: ObjOptionalLimit ← readRefWithLimit(r);
        b: Object ← unaryDispatch(incrementTable, null, a, ArgumentList⟨[], {}⟩);
        writeReference(r, b);
        return getObject(a);
    [FullPostfixExpression ⇒ PostfixExpressionOrSuper [no line break] --] do
        r: ObjOrRefOptionalLimit ← Eval[PostfixExpressionOrSuper](e);
        a: ObjOptionalLimit ← readRefWithLimit(r);
        b: Object ← unaryDispatch(decrementTable, null, a, ArgumentList⟨[], {}⟩);
        writeReference(r, b);
        return getObject(a)
end proc;

~~proc Eval[FullNewExpression] (e: DynamicEnv): ObjOrRef~~proc Eval[FullNewExpression] (e: Environment): ObjOrRef
    [FullNewExpression ⇒ new FullNewSubexpression Arguments] do
        f: Object ← readReference(Eval[FullNewSubexpression](e));
        args: ArgumentList ← Eval[Arguments](e);
        return unaryDispatch(constructTable, null, f, args);
    [FullNewExpression ⇒ new FullSuperExpression Arguments] do
        f: ObjOptionalLimit ← readRefWithLimit(Eval[FullSuperExpression](e));
        args: ArgumentList ← Eval[Arguments](e);
        return unaryDispatch(constructTable, null, f, args)
end proc;

~~proc Eval[FullNewSubexpression] (e: DynamicEnv): ObjOrRef~~proc Eval[FullNewSubexpression] (e: Environment): ObjOrRef
    [FullNewSubexpression ⇒ PrimaryExpression] do return Eval[PrimaryExpression](e);
    ~~[FullNewSubexpression ⇒ QualifiedIdentifier] do return Eval[QualifiedIdentifier](e);~~[FullNewSubexpression ⇒ QualifiedIdentifier] do
        return VariableReference⟨e, Name[QualifiedIdentifier]⟩;
    [FullNewSubexpression ⇒ FullNewExpression] do return Eval[FullNewExpression](e);
    [FullNewSubexpression0 ⇒ FullNewSubexpression1 MemberOperator] do
        a: Object ← readReference(Eval[FullNewSubexpression1](e));
        return Eval[MemberOperator](e, a);
    [FullNewSubexpression ⇒ SuperExpression DotOperator] do
        a: ObjOptionalLimit ← readRefWithLimit(Eval[SuperExpression](e));
        return Eval[DotOperator](e, a)
end proc;

~~proc Eval[ShortNewExpression] (e: DynamicEnv): ObjOrRef~~proc Eval[ShortNewExpression] (e: Environment): ObjOrRef
    [ShortNewExpression ⇒ new ShortNewSubexpression] do
        f: Object ← readReference(Eval[ShortNewSubexpression](e));
        return unaryDispatch(constructTable, null, f, ArgumentList⟨[], {}⟩);
    [ShortNewExpression ⇒ new SuperExpression] do
        f: ObjOptionalLimit ← readRefWithLimit(Eval[SuperExpression](e));
        return unaryDispatch(constructTable, null, f, ArgumentList⟨[], {}⟩)
end proc;

~~Eval[ShortNewSubexpression]: DynamicEnv → ObjOrRef;~~Eval[ShortNewSubexpression]: Environment → ObjOrRef;
    Eval[ShortNewSubexpression ⇒ FullNewSubexpression] = Eval[FullNewSubexpression];
    Eval[ShortNewSubexpression ⇒ ShortNewExpression] = Eval[ShortNewExpression];
```

Member Operators

Syntax

MemberOperator ⇒
    DotOperator
  | **.** *ParenExpression*

DotOperator ⇒
    . *QualifiedIdentifier*
  | *Brackets*

Brackets ⇒
    [ ]
  | **[** *ListExpression*$^{\text{allowIn}}$ **]**
  | **[** *NamedArgumentList* **]**

Arguments ⇒
    ParenExpressions
  | **(** *NamedArgumentList* **)**

ParenExpressions ⇒
    ( )
  | *ParenListExpression*

NamedArgumentList ⇒
    LiteralField
  | *ListExpression*$^{\text{allowIn}}$ **,** *LiteralField*
  | *NamedArgumentList* **,** *LiteralField*

Validation

~~**proc** *Validate*[*MemberOperator*] (*v*: VALIDATIONENV)~~
~~[*MemberOperator* ⇒ *DotOperator*] **do** *Validate*[*DotOperator*](*v*);~~
~~[*MemberOperator* ⇒ **.** *ParenExpression*] **do** *Validate*[*ParenExpression*](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*DotOperator*] (*v*: VALIDATIONENV)~~
~~[*DotOperator* ⇒ **.** *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*v*);~~
~~[*DotOperator* ⇒ *Brackets*] **do** *Validate*[*Brackets*](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*Brackets*] (*v*: VALIDATIONENV)~~**proc** *Validate*[*MemberOperator*] (*c*: CONTEXT, *e*: STATICENV)
[*MemberOperator* ⇒ *DotOperator*] **do** *Validate*[*DotOperator*](*c*, *e*);
[*MemberOperator* ⇒ **.** *ParenExpression*] **do** *Validate*[*ParenExpression*](*c*, *e*)
end proc;

proc *Validate*[*DotOperator*] (*c*: CONTEXT, *e*: STATICENV)
[*DotOperator* ⇒ **.** *QualifiedIdentifier*] **do** *Validate*[*QualifiedIdentifier*](*c*, *e*);
[*DotOperator* ⇒ *Brackets*] **do** *Validate*[*Brackets*](*c*, *e*)
end proc;

proc *Validate*[*Brackets*] (*c*: CONTEXT, *e*: STATICENV)
[*Brackets* ⇒ **[ ]**] **do nothing**;
~~[*Brackets* ⇒ **[** *ListExpression*$^{\text{allowIn}}$ **]**] **do** *Validate*[*ListExpression*$^{\text{allowIn}}$](*v*);~~
~~[*Brackets* ⇒ **[** *NamedArgumentList* **]**] **do** *Validate*[*NamedArgumentList*](*v*)~~
~~**end proc**;~~

```
~~proc Validate[Arguments] (v: ValidationEnv)~~
    ~~[Arguments ⇒ ParenExpressions] do Validate[ParenExpressions](v);~~
    ~~[Arguments ⇒ ( NamedArgumentList )] do Validate[NamedArgumentList](v)~~
~~end proc~~;

    ~~proc Validate[ParenExpressions] (v: ValidationEnv)~~[Brackets ⇒ [ ListExpression^allowIn ]] do Validate[ListExpression^allowIn](c, e);
    [Brackets ⇒ [ NamedArgumentList ]] do Validate[NamedArgumentList](c, e)
end proc;

proc Validate[Arguments] (c: Context, e: StaticEnv)
    [Arguments ⇒ ParenExpressions] do Validate[ParenExpressions](c, e);
    [Arguments ⇒ ( NamedArgumentList )] do Validate[NamedArgumentList](c, e)
end proc;

proc Validate[ParenExpressions] (c: Context, e: StaticEnv)
    [ParenExpressions ⇒ ( )] do nothing;
    ~~[ParenExpressions ⇒ ParenListExpression] do Validate[ParenListExpression](v)~~
~~end proc~~;

~~proc Validate[NamedArgumentList] (v: ValidationEnv): String{}~~
    ~~[NamedArgumentList ⇒ LiteralField] do return Validate[LiteralField](v);~~[ParenExpressions ⇒ ParenListExpression]
        do Validate[ParenListExpression](c, e)
end proc;

proc Validate[NamedArgumentList] (c: Context, e: StaticEnv): String{}
    [NamedArgumentList ⇒ LiteralField] do return Validate[LiteralField](c, e);
    [NamedArgumentList ⇒ ListExpression^allowIn , LiteralField] do
        ~~Validate[ListExpression^allowIn](v);~~
        ~~return Validate[LiteralField](v);~~Validate[ListExpression^allowIn](c, e);
        return Validate[LiteralField](c, e);
    [NamedArgumentList_0 ⇒ NamedArgumentList_1 , LiteralField] do
        ~~names1: String{} ← Validate[NamedArgumentList_1](v);~~
        ~~names2: String{} ← Validate[LiteralField](v);~~names1: String{} ← Validate[NamedArgumentList_1](c, e);
        names2: String{} ← Validate[LiteralField](c, e);
        if names1 ∩ names2 ≠ {} then throw syntaxError end if;
        return names1 ∪ names2
end proc;
```

Evaluation

```
~~proc Eval[MemberOperator] (e: DynamicEnv, base: Object): ObjOrRef~~proc Eval[MemberOperator] (e: EnvironMEN
    T, base: Object): ObjOrRef
    [MemberOperator ⇒ DotOperator] do return Eval[DotOperator](e, base);
    [MemberOperator ⇒ . ParenExpression] do ????
end proc;

~~proc Eval[DotOperator] (e: DynamicEnv, base: ObjOptionalLimit): ObjOrRef~~proc Eval[DotOperator] (e: EnvironM
    ENT, base: ObjOptionalLimit): ObjOrRef
    [DotOperator ⇒ . QualifiedIdentifier] do
        ~~n: PartialName ← Name[QualifiedIdentifier](e);~~
        ~~return DotReference⟨base, n⟩;~~return DotReference⟨base, Name[QualifiedIdentifier]⟩;
    [DotOperator ⇒ Brackets] do
        args: ArgumentList ← Eval[Brackets](e);
        return BracketReference⟨base, args⟩
end proc;
```

~~**proc** *Eval*[*Brackets*] (*e*: DYNAMICENV): ARGUMENTLIST~~**proc** *Eval*[*Brackets*] (*e*: ENVIRONMENT): ARGUMENTLIST
  [*Brackets* ⇒ **[ ]**] **do return** ARGUMENTLIST⟨**[]**, {}⟩;
  [*Brackets* ⇒ **[** *ListExpression*$^{\text{allowIn}}$ **]**] **do**
    positional: OBJECT[] ← *EvalAsList*[*ListExpression*$^{\text{allowIn}}$](*e*);
    return ARGUMENTLIST⟨*positional*, {}⟩;
  [*Brackets* ⇒ **[** *NamedArgumentList* **]**] **do return** *Eval*[*NamedArgumentList*](*e*)
end proc;

~~**proc** *Eval*[*Arguments*] (*e*: DYNAMICENV): ARGUMENTLIST~~**proc** *Eval*[*Arguments*] (*e*: ENVIRONMENT): ARGUMENTLIST
  [*Arguments* ⇒ *ParenExpressions*] **do return** *Eval*[*ParenExpressions*](*e*);
  [*Arguments* ⇒ **(** *NamedArgumentList* **)**] **do return** *Eval*[*NamedArgumentList*](*e*)
end proc;

~~**proc** *Eval*[*ParenExpressions*] (*e*: DYNAMICENV): ARGUMENTLIST~~**proc** *Eval*[*ParenExpressions*] (*e*: ENVIRONMENT): ARGUMENTLIST
  [*ParenExpressions* ⇒ **( )**] **do return** ARGUMENTLIST⟨**[]**, {}⟩;
  [*ParenExpressions* ⇒ *ParenListExpression*] **do**
    positional: OBJECT[] ← *EvalAsList*[*ParenListExpression*](*e*);
    return ARGUMENTLIST⟨*positional*, {}⟩
end proc;

~~**proc** *Eval*[*NamedArgumentList*] (*e*: DYNAMICENV): ARGUMENTLIST~~**proc** *Eval*[*NamedArgumentList*] (*e*: ENVIRONMENT): ARGUMENTLIST
  [*NamedArgumentList* ⇒ *LiteralField*] **do**
    na: NAMEDARGUMENT ← *Eval*[*LiteralField*](*e*);
    return ARGUMENTLIST⟨**[]**, {*na*}⟩;
  [*NamedArgumentList* ⇒ *ListExpression*$^{\text{allowIn}}$ **,** *LiteralField*] **do**
    positional: OBJECT[] ← *EvalAsList*[*ListExpression*$^{\text{allowIn}}$](*e*);
    na: NAMEDARGUMENT ← *Eval*[*LiteralField*](*e*);
    return ARGUMENTLIST⟨*positional*, {*na*}⟩;
  [*NamedArgumentList*$_0$ ⇒ *NamedArgumentList*$_1$ **,** *LiteralField*] **do**
    args: ARGUMENTLIST ← *Eval*[*NamedArgumentList*$_1$](*e*);
    na: NAMEDARGUMENT ← *Eval*[*LiteralField*](*e*);
    if some *na2* ∈ *args*.named **satisfies** *na2*.name = *na*.name **then**
      throw argumentMismatchError
    end if;
    return ARGUMENTLIST⟨*args*.positional, *args*.named ∪ {*na*}⟩
end proc;

Unary Operators

Syntax

UnaryExpression ⇒
    PostfixExpression
  | **delete** *PostfixExpression*
  | **void** *UnaryExpression*
  | **typeof** *UnaryExpression*
  | **++** *PostfixExpressionOrSuper*
  | **--** *PostfixExpressionOrSuper*
  | **+** *UnaryExpressionOrSuper*
  | **-** *UnaryExpressionOrSuper*
  | **~** *UnaryExpressionOrSuper*
  | **!** *UnaryExpression*

UnaryExpressionOrSuper ⇒
UnaryExpression
| *SuperExpression*

Validation

~~**proc** *Validate*[*UnaryExpression*] (*v*: VALIDATIONENV)~~
~~[*UnaryExpression* ⇒ *PostfixExpression*] **do** *Validate*[*PostfixExpression*](*v*);~~
~~[*UnaryExpression* ⇒ **delete** *PostfixExpression*] **do** *Validate*[*PostfixExpression*](*v*);~~
~~[*UnaryExpression*$_0$ ⇒ **void** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*v*);~~
~~[*UnaryExpression*$_0$ ⇒ **typeof** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*v*);~~**proc** *Validate*[*UnaryExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*UnaryExpression* ⇒ *PostfixExpression*] **do** *Validate*[*PostfixExpression*](*c*, *e*);
[*UnaryExpression* ⇒ **delete** *PostfixExpression*] **do** *Validate*[*PostfixExpression*](*c*, *e*);
[*UnaryExpression*$_0$ ⇒ **void** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*c*, *e*);
[*UnaryExpression*$_0$ ⇒ **typeof** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*c*, *e*);
[*UnaryExpression* ⇒ **++** *PostfixExpressionOrSuper*] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*);~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*);
[*UnaryExpression* ⇒ **--** *PostfixExpressionOrSuper*] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*);~~
~~[*UnaryExpression* ⇒ **+** *UnaryExpressionOrSuper*] **do** *Validate*[*UnaryExpressionOrSuper*](*v*);~~
~~[*UnaryExpression* ⇒ **-** *UnaryExpressionOrSuper*] **do** *Validate*[*UnaryExpressionOrSuper*](*v*);~~
~~[*UnaryExpression* ⇒ **~** *UnaryExpressionOrSuper*] **do** *Validate*[*UnaryExpressionOrSuper*](*v*);~~
~~[*UnaryExpression*$_0$ ⇒ **!** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*v*)~~
~~**end proc**;~~

~~*Validate*[*UnaryExpressionOrSuper*]: VALIDATIONENV → ();~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*);
[*UnaryExpression* ⇒ **+** *UnaryExpressionOrSuper*] **do**
Validate[*UnaryExpressionOrSuper*](*c*, *e*);
[*UnaryExpression* ⇒ **-** *UnaryExpressionOrSuper*] **do**
Validate[*UnaryExpressionOrSuper*](*c*, *e*);
[*UnaryExpression* ⇒ **~** *UnaryExpressionOrSuper*] **do**
Validate[*UnaryExpressionOrSuper*](*c*, *e*);
[*UnaryExpression*$_0$ ⇒ **!** *UnaryExpression*$_1$] **do** *Validate*[*UnaryExpression*$_1$](*c*, *e*)
end proc;

Validate[*UnaryExpressionOrSuper*]: CONTEXT × STATICENV → ();
Validate[*UnaryExpressionOrSuper* ⇒ *UnaryExpression*] = *Validate*[*UnaryExpression*];
Validate[*UnaryExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*UnaryExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*UnaryExpression*] (*e*: ENVIRONMENT): OBJORREF
[*UnaryExpression* ⇒ *PostfixExpression*] **do return** *Eval*[*PostfixExpression*](*e*);
[*UnaryExpression* ⇒ **delete** *PostfixExpression*] **do**
return *deleteReference*(*Eval*[*PostfixExpression*](*e*));
[*UnaryExpression*$_0$ ⇒ **void** *UnaryExpression*$_1$] **do**
readReference(*Eval*[*UnaryExpression*$_1$](*e*));
return undefined;

```
    [UnaryExpression0 ⇒ typeof UnaryExpression1] do
        a: OBJECT ← readReference(Eval[UnaryExpression1](e));
        case a of
            UNDEFINED do return "undefined";
            NULL ∪ PROTOTYPE do return "object";
            BOOLEAN do return "boolean";
            FLOAT64 do return "number";
            STRING do return "string";
            NAMESPACE do return "namespace";
            COMPOUNDATTRIBUTE do return "attribute";
            CLASS ∪ METHODCLOSURE do return "function";
            INSTANCE do return a.typeofString
        end case;
    [UnaryExpression ⇒ ++ PostfixExpressionOrSuper] do
        r: OBJORREFOPTIONALLIMIT ← Eval[PostfixExpressionOrSuper](e);
        a: OBJOPTIONALLIMIT ← readRefWithLimit(r);
        b: OBJECT ← unaryDispatch(incrementTable, null, a, ARGUMENTLIST⟨[], {}⟩);
        writeReference(r, b);
        return b;
    [UnaryExpression ⇒ -- PostfixExpressionOrSuper] do
        r: OBJORREFOPTIONALLIMIT ← Eval[PostfixExpressionOrSuper](e);
        a: OBJOPTIONALLIMIT ← readRefWithLimit(r);
        b: OBJECT ← unaryDispatch(decrementTable, null, a, ARGUMENTLIST⟨[], {}⟩);
        writeReference(r, b);
        return b;
    [UnaryExpression ⇒ + UnaryExpressionOrSuper] do
        a: OBJOPTIONALLIMIT ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
        return unaryPlus(a);
    [UnaryExpression ⇒ - UnaryExpressionOrSuper] do
        a: OBJOPTIONALLIMIT ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
        return unaryDispatch(minusTable, null, a, ARGUMENTLIST⟨[], {}⟩);
    [UnaryExpression ⇒ ~ UnaryExpressionOrSuper] do
        a: OBJOPTIONALLIMIT ← readRefWithLimit(Eval[UnaryExpressionOrSuper](e));
        return unaryDispatch(bitwiseNotTable, null, a, ARGUMENTLIST⟨[], {}⟩);
    [UnaryExpression0 ⇒ ! UnaryExpression1] do
        a: OBJECT ← readReference(Eval[UnaryExpression1](e));
        return unaryNot(a)
end proc;

~~Eval[UnaryExpressionOrSuper]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~Eval[UnaryExpressionOrSuper]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
    Eval[UnaryExpressionOrSuper ⇒ UnaryExpression] = Eval[UnaryExpression];
    Eval[UnaryExpressionOrSuper ⇒ SuperExpression] = Eval[SuperExpression];
```

Multiplicative Operators

Syntax

```
MultiplicativeExpression ⇒
    UnaryExpression
  | MultiplicativeExpressionOrSuper * UnaryExpressionOrSuper
  | MultiplicativeExpressionOrSuper / UnaryExpressionOrSuper
  | MultiplicativeExpressionOrSuper % UnaryExpressionOrSuper
```

MultiplicativeExpressionOrSuper ⇒
MultiplicativeExpression
| *SuperExpression*

Validation

proc *Validate*[*MultiplicativeExpression*] (*v*: VALIDATIONENV)
[*MultiplicativeExpression* ⇒ *UnaryExpression*] **do** *Validate*[*UnaryExpression*](*v*);**proc** *Validate*[*MultiplicativeExpression*] (*c*: CONTEXT, *e*: STATICENV)
[*MultiplicativeExpression* ⇒ *UnaryExpression*] **do** *Validate*[*UnaryExpression*](*c*, *e*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* ***** *UnaryExpressionOrSuper*] **do**
Validate[*MultiplicativeExpressionOrSuper*](*v*);
Validate[*UnaryExpressionOrSuper*](*v*);*Validate*[*MultiplicativeExpressionOrSuper*](*c*, *e*);
Validate[*UnaryExpressionOrSuper*](*c*, *e*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **/** *UnaryExpressionOrSuper*] **do**
Validate[*MultiplicativeExpressionOrSuper*](*v*);
Validate[*UnaryExpressionOrSuper*](*v*);*Validate*[*MultiplicativeExpressionOrSuper*](*c*, *e*);
Validate[*UnaryExpressionOrSuper*](*c*, *e*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **%** *UnaryExpressionOrSuper*] **do**
Validate[*MultiplicativeExpressionOrSuper*](*v*);
Validate[*UnaryExpressionOrSuper*](*v*)
end proc;

Validate[*MultiplicativeExpressionOrSuper*]: VALIDATIONENV → ();*Validate*[*MultiplicativeExpressionOrSuper*](*c*, *e*);
Validate[*UnaryExpressionOrSuper*](*c*, *e*)
end proc;

Validate[*MultiplicativeExpressionOrSuper*]: CONTEXT × STATICENV → ();
Validate[*MultiplicativeExpressionOrSuper* ⇒ *MultiplicativeExpression*] = *Validate*[*MultiplicativeExpression*];
Validate[*MultiplicativeExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

proc *Eval*[*MultiplicativeExpression*] (*e*: DYNAMICENV): OBJORREF**proc** *Eval*[*MultiplicativeExpression*] (*e*: ENVIRONMENT): OBJORREF
[*MultiplicativeExpression* ⇒ *UnaryExpression*] **do return** *Eval*[*UnaryExpression*](*e*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* ***** *UnaryExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
return *binaryDispatch*(*multiplyTable*, *a*, *b*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **/** *UnaryExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
return *binaryDispatch*(*divideTable*, *a*, *b*);
[*MultiplicativeExpression* ⇒ *MultiplicativeExpressionOrSuper* **%** *UnaryExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*UnaryExpressionOrSuper*](*e*));
return *binaryDispatch*(*remainderTable*, *a*, *b*)
end proc;

Eval[*MultiplicativeExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;*Eval*[*MultiplicativeExpressionOrSuper*]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
Eval[*MultiplicativeExpressionOrSuper* ⇒ *MultiplicativeExpression*] = *Eval*[*MultiplicativeExpression*];
Eval[*MultiplicativeExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Additive Operators

Syntax

AdditiveExpression ⇒
    MultiplicativeExpression
  | *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*
  | *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*

AdditiveExpressionOrSuper ⇒
    AdditiveExpression
  | *SuperExpression*

Validation

~~**proc** *Validate*[*AdditiveExpression*] (*v*: VALIDATIONENV)~~**proc** *Validate*[*AdditiveExpression*] (*c*: CONTEXT, *e*: STATICENV)
  [*AdditiveExpression* ⇒ *MultiplicativeExpression*] **do**
    ~~*Validate*[*MultiplicativeExpression*](*v*);~~*Validate*[*MultiplicativeExpression*](*c*, *e*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*] **do**
    ~~*Validate*[*AdditiveExpressionOrSuper*](*v*);~~
    ~~*Validate*[*MultiplicativeExpressionOrSuper*](*v*);~~*Validate*[*AdditiveExpressionOrSuper*](*c*, *e*);
    Validate[*MultiplicativeExpressionOrSuper*](*c*, *e*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*] **do**
    ~~*Validate*[*AdditiveExpressionOrSuper*](*v*);~~
    ~~*Validate*[*MultiplicativeExpressionOrSuper*](*v*)~~
~~**end proc**;~~

    ~~*Validate*[*AdditiveExpressionOrSuper*]: VALIDATIONENV → ();~~*Validate*[*AdditiveExpressionOrSuper*](*c*, *e*);
    Validate[*MultiplicativeExpressionOrSuper*](*c*, *e*)
end proc;

Validate[*AdditiveExpressionOrSuper*]: CONTEXT × STATICENV → ();
  Validate[*AdditiveExpressionOrSuper* ⇒ *AdditiveExpression*] = *Validate*[*AdditiveExpression*];
  Validate[*AdditiveExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*AdditiveExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*AdditiveExpression*] (*e*: ENVIRONMENT): OBJORREF
  [*AdditiveExpression* ⇒ *MultiplicativeExpression*] **do**
    return *Eval*[*MultiplicativeExpression*](*e*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **+** *MultiplicativeExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    return *binaryDispatch*(*addTable*, *a*, *b*);
  [*AdditiveExpression* ⇒ *AdditiveExpressionOrSuper* **-** *MultiplicativeExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*MultiplicativeExpressionOrSuper*](*e*));
    return *binaryDispatch*(*subtractTable*, *a*, *b*)
end proc;

~~*Eval*[*AdditiveExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~*Eval*[*AdditiveExpressionOrSuper*]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
  Eval[*AdditiveExpressionOrSuper* ⇒ *AdditiveExpression*] = *Eval*[*AdditiveExpression*];
  Eval[*AdditiveExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Bitwise Shift Operators

Syntax

ShiftExpression ⇒
    AdditiveExpression
  | *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*
  | *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*
  | *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*

ShiftExpressionOrSuper ⇒
    ShiftExpression
  | *SuperExpression*

Validation

~~**proc** *Validate*[*ShiftExpression*] (*v*: VALIDATIONENV)~~
~~[*ShiftExpression* ⇒ *AdditiveExpression*] **do** *Validate*[*AdditiveExpression*](*v*);~~**proc** *Validate*[*ShiftExpression*] (*c*: CONTEXT, *e*: STATICENV)
  [*ShiftExpression* ⇒ *AdditiveExpression*] **do** *Validate*[*AdditiveExpression*](*c*, *e*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*] **do**
    ~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~
    ~~*Validate*[*AdditiveExpressionOrSuper*](*v*);~~*Validate*[*ShiftExpressionOrSuper*](*c*, *e*);
    Validate[*AdditiveExpressionOrSuper*](*c*, *e*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*] **do**
    ~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~
    ~~*Validate*[*AdditiveExpressionOrSuper*](*v*);~~*Validate*[*ShiftExpressionOrSuper*](*c*, *e*);
    Validate[*AdditiveExpressionOrSuper*](*c*, *e*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*] **do**
    ~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~
    ~~*Validate*[*AdditiveExpressionOrSuper*](*v*)~~
~~**end proc**;~~

    ~~*Validate*[*ShiftExpressionOrSuper*]: VALIDATIONENV → ();~~*Validate*[*ShiftExpressionOrSuper*](*c*, *e*);
    Validate[*AdditiveExpressionOrSuper*](*c*, *e*)
end proc;

Validate[*ShiftExpressionOrSuper*]: CONTEXT × STATICENV → ();
  Validate[*ShiftExpressionOrSuper* ⇒ *ShiftExpression*] = *Validate*[*ShiftExpression*];
  Validate[*ShiftExpressionOrSuper* ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*ShiftExpression*] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*ShiftExpression*] (*e*: ENVIRONMENT): OBJORREF
  [*ShiftExpression* ⇒ *AdditiveExpression*] **do return** *Eval*[*AdditiveExpression*](*e*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **<<** *AdditiveExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    return *binaryDispatch*(*shiftLeftTable*, *a*, *b*);
  [*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>** *AdditiveExpressionOrSuper*] **do**
    a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
    b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
    return *binaryDispatch*(*shiftRightTable*, *a*, *b*);

[*ShiftExpression* ⇒ *ShiftExpressionOrSuper* **>>>** *AdditiveExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*AdditiveExpressionOrSuper*](*e*));
return *binaryDispatch*(*shiftRightUnsignedTable*, *a*, *b*)
end proc;

~~*Eval*[*ShiftExpressionOrSuper*]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~*Eval*[*ShiftExpressionOrSuper*]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
Eval[*ShiftExpressionOrSuper* ⇒ *ShiftExpression*] = *Eval*[*ShiftExpression*];
Eval[*ShiftExpressionOrSuper* ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Relational Operators

Syntax

RelationalExpression$^{\text{allowIn}}$ ⇒
ShiftExpression
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **<** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **>** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **<=** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{allowIn}}$ **>=** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{allowIn}}$ **is** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **as** *ShiftExpression*
| *RelationalExpression*$^{\text{allowIn}}$ **in** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{allowIn}}$ **instanceof** *ShiftExpression*

RelationalExpression$^{\text{noIn}}$ ⇒
ShiftExpression
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **<** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **>** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **<=** *ShiftExpressionOrSuper*
| *RelationalExpressionOrSuper*$^{\text{noIn}}$ **>=** *ShiftExpressionOrSuper*
| *RelationalExpression*$^{\text{noIn}}$ **is** *ShiftExpression*
| *RelationalExpression*$^{\text{noIn}}$ **as** *ShiftExpression*
| *RelationalExpression*$^{\text{noIn}}$ **instanceof** *ShiftExpression*

RelationalExpressionOrSuper$^{\beta}$ ⇒
RelationalExpression$^{\beta}$
| *SuperExpression*

Validation

~~**proc** *Validate*[*RelationalExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~
~~[*RelationalExpression*$^{\beta}$ ⇒ *ShiftExpression*] **do** *Validate*[*ShiftExpression*](*v*);~~**proc** *Validate*[*RelationalExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*RelationalExpression*$^{\beta}$ ⇒ *ShiftExpression*] **do** *Validate*[*ShiftExpression*](*c*, *e*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<** *ShiftExpressionOrSuper*] **do**
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*ShiftExpressionOrSuper*](*c*, *e*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>** *ShiftExpressionOrSuper*] **do**
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*ShiftExpressionOrSuper*](*c*, *e*);

[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<=** *ShiftExpressionOrSuper*] **do**
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*ShiftExpressionOrSuper*](*c*, *e*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>=** *ShiftExpressionOrSuper*] **do**
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*ShiftExpressionOrSuper*](*c*, *e*);
[*RelationalExpression*$^{\beta}_0$ ⇒ *RelationalExpression*$^{\beta}_1$ **is** *ShiftExpression*] **do**
~~*Validate*[*RelationalExpression*$^{\beta}_1$](*v*);~~
~~*Validate*[*ShiftExpression*](*v*);~~*Validate*[*RelationalExpression*$^{\beta}_1$](*c*, *e*);
Validate[*ShiftExpression*](*c*, *e*);
[*RelationalExpression*$^{\beta}_0$ ⇒ *RelationalExpression*$^{\beta}_1$ **as** *ShiftExpression*] **do**
~~*Validate*[*RelationalExpression*$^{\beta}_1$](*v*);~~
~~*Validate*[*ShiftExpression*](*v*);~~*Validate*[*RelationalExpression*$^{\beta}_1$](*c*, *e*);
Validate[*ShiftExpression*](*c*, *e*);
[*RelationalExpression*$^{\text{allowIn}}_0$ ⇒ *RelationalExpression*$^{\text{allowIn}}_1$ **in** *ShiftExpressionOrSuper*] **do**
~~*Validate*[*RelationalExpression*$^{\text{allowIn}}_1$](*v*);~~
~~*Validate*[*ShiftExpressionOrSuper*](*v*);~~*Validate*[*RelationalExpression*$^{\text{allowIn}}_1$](*c*, *e*);
Validate[*ShiftExpressionOrSuper*](*c*, *e*);
[*RelationalExpression*$^{\beta}_0$ ⇒ *RelationalExpression*$^{\beta}_1$ **instanceof** *ShiftExpression*] **do**
~~*Validate*[*RelationalExpression*$^{\beta}_1$](*v*);~~
~~*Validate*[*ShiftExpression*](*v*)~~
~~**end proc**;~~

~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$]: VALIDATIONENV → ();~~*Validate*[*RelationalExpression*$^{\beta}_1$](*c*, *e*);
Validate[*ShiftExpression*](*c*, *e*)
end proc;

Validate[*RelationalExpressionOrSuper*$^{\beta}$]: CONTEXT × STATICENV → ();
Validate[*RelationalExpressionOrSuper*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$] = *Validate*[*RelationalExpression*$^{\beta}$];
Validate[*RelationalExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*RelationalExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*RelationalExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
[*RelationalExpression*$^{\beta}$ ⇒ *ShiftExpression*] **do return** *Eval*[*ShiftExpression*](*e*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessTable*, *a*, *b*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessTable*, *b*, *a*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **<=** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessOrEqualTable*, *a*, *b*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpressionOrSuper*$^{\beta}$ **>=** *ShiftExpressionOrSuper*] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*ShiftExpressionOrSuper*](*e*));
return *binaryDispatch*(*lessOrEqualTable*, *b*, *a*);
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$ **is** *ShiftExpression*] **do** ????;

[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$ **as** *ShiftExpression*] **do** ????;
[*RelationalExpression*$^{\text{allowIn}}$ ⇒ *RelationalExpression*$^{\text{allowIn}}$ **in** *ShiftExpressionOrSuper*] **do**
????;
[*RelationalExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$ **instanceof** *ShiftExpression*] **do** ????
end proc;

~~*Eval*[*RelationalExpressionOrSuper*$^{\beta}$]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~*Eval*[*RelationalExpressionOrSuper*$^{\beta}$]:
ENVIRONMENT → OBJORREFOPTIONALLIMIT;
Eval[*RelationalExpressionOrSuper*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$] = *Eval*[*RelationalExpression*$^{\beta}$];
Eval[*RelationalExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Equality Operators

Syntax

EqualityExpression$^{\beta}$ ⇒
RelationalExpression$^{\beta}$
| *EqualityExpressionOrSuper*$^{\beta}$ **==** *RelationalExpressionOrSuper*$^{\beta}$
| *EqualityExpressionOrSuper*$^{\beta}$ **!=** *RelationalExpressionOrSuper*$^{\beta}$
| *EqualityExpressionOrSuper*$^{\beta}$ **===** *RelationalExpressionOrSuper*$^{\beta}$
| *EqualityExpressionOrSuper*$^{\beta}$ **!==** *RelationalExpressionOrSuper*$^{\beta}$

EqualityExpressionOrSuper$^{\beta}$ ⇒
EqualityExpression$^{\beta}$
| *SuperExpression*

Validation

~~**proc** *Validate*[*EqualityExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~
~~[*EqualityExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$] **do** *Validate*[*RelationalExpression*$^{\beta}$](*v*);~~**proc** *Validate*[*EqualityExpression*$^{\beta}$]
(*c*: CONTEXT, *e*: STATICENV)
[*EqualityExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$] **do**
Validate[*RelationalExpression*$^{\beta}$](*c*, *e*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **==** *RelationalExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **!=** *RelationalExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **===** *RelationalExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*);~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **!==** *RelationalExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*RelationalExpressionOrSuper*$^{\beta}$](*v*)~~
~~**end proc**;~~

~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$]: VALIDATIONENV → ();~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*RelationalExpressionOrSuper*$^{\beta}$](*c*, *e*)
end proc;

Validate[*EqualityExpressionOrSuper*$^{\beta}$]: CONTEXT × STATICENV → ();
Validate[*EqualityExpressionOrSuper*$^{\beta}$ ⇒ *EqualityExpression*$^{\beta}$] = *Validate*[*EqualityExpression*$^{\beta}$];
Validate[*EqualityExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*EqualityExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*EqualityExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
[*EqualityExpression*$^{\beta}$ ⇒ *RelationalExpression*$^{\beta}$] **do**
return *Eval*[*RelationalExpression*$^{\beta}$](*e*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **==** *RelationalExpressionOrSuper*$^{\beta}$] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
return *binaryDispatch*(*equalTable*, *a*, *b*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **!=** *RelationalExpressionOrSuper*$^{\beta}$] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
return *unaryNot*(*binaryDispatch*(*equalTable*, *a*, *b*));
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **===** *RelationalExpressionOrSuper*$^{\beta}$] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
return *binaryDispatch*(*strictEqualTable*, *a*, *b*);
[*EqualityExpression*$^{\beta}$ ⇒ *EqualityExpressionOrSuper*$^{\beta}$ **!==** *RelationalExpressionOrSuper*$^{\beta}$] **do**
a: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^{\beta}$](*e*));
b: OBJOPTIONALLIMIT ← *readRefWithLimit*(*Eval*[*RelationalExpressionOrSuper*$^{\beta}$](*e*));
return *unaryNot*(*binaryDispatch*(*strictEqualTable*, *a*, *b*))
end proc;

~~*Eval*[*EqualityExpressionOrSuper*$^{\beta}$]: DYNAMICENV → OBJORREFOPTIONALLIMIT;~~*Eval*[*EqualityExpressionOrSuper*$^{\beta}$]: ENVIRONMENT → OBJORREFOPTIONALLIMIT;
Eval[*EqualityExpressionOrSuper*$^{\beta}$ ⇒ *EqualityExpression*$^{\beta}$] = *Eval*[*EqualityExpression*$^{\beta}$];
Eval[*EqualityExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Binary Bitwise Operators

Syntax

BitwiseAndExpression$^{\beta}$ ⇒
EqualityExpression$^{\beta}$
| *BitwiseAndExpressionOrSuper*$^{\beta}$ **&** *EqualityExpressionOrSuper*$^{\beta}$

BitwiseXorExpression$^{\beta}$ ⇒
BitwiseAndExpression$^{\beta}$
| *BitwiseXorExpressionOrSuper*$^{\beta}$ **^** *BitwiseAndExpressionOrSuper*$^{\beta}$

BitwiseOrExpression$^{\beta}$ ⇒
BitwiseXorExpression$^{\beta}$
| *BitwiseOrExpressionOrSuper*$^{\beta}$ **|** *BitwiseXorExpressionOrSuper*$^{\beta}$

BitwiseAndExpressionOrSuper$^{\beta}$ ⇒
BitwiseAndExpression$^{\beta}$
| *SuperExpression*

BitwiseXorExpressionOrSuper$^{\beta}$ ⇒
BitwiseXorExpression$^{\beta}$
| *SuperExpression*

BitwiseOrExpressionOrSuper$^{\beta}$ ⇒
BitwiseOrExpression$^{\beta}$
| *SuperExpression*

Validation

~~**proc** *Validate*[*BitwiseAndExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~
~~[*BitwiseAndExpression*$^{\beta}$ ⇒ *EqualityExpression*$^{\beta}$] **do** *Validate*[*EqualityExpression*$^{\beta}$](*v*);~~**proc** *Validate*[*BitwiseAndExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*BitwiseAndExpression*$^{\beta}$ ⇒ *EqualityExpression*$^{\beta}$] **do**
Validate[*EqualityExpression*$^{\beta}$](*c*, *e*);
[*BitwiseAndExpression*$^{\beta}$ ⇒ *BitwiseAndExpressionOrSuper*$^{\beta}$ **&** *EqualityExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*BitwiseAndExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*EqualityExpressionOrSuper*$^{\beta}$](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*BitwiseXorExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~*Validate*[*BitwiseAndExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*EqualityExpressionOrSuper*$^{\beta}$](*c*, *e*)
end proc;

proc *Validate*[*BitwiseXorExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*BitwiseXorExpression*$^{\beta}$ ⇒ *BitwiseAndExpression*$^{\beta}$] **do**
~~*Validate*[*BitwiseAndExpression*$^{\beta}$](*v*);~~*Validate*[*BitwiseAndExpression*$^{\beta}$](*c*, *e*);
[*BitwiseXorExpression*$^{\beta}$ ⇒ *BitwiseXorExpressionOrSuper*$^{\beta}$ **^** *BitwiseAndExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*BitwiseXorExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*BitwiseAndExpressionOrSuper*$^{\beta}$](*v*)~~
~~**end proc**;~~

~~**proc** *Validate*[*BitwiseOrExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~*Validate*[*BitwiseXorExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*BitwiseAndExpressionOrSuper*$^{\beta}$](*c*, *e*)
end proc;

proc *Validate*[*BitwiseOrExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*BitwiseOrExpression*$^{\beta}$ ⇒ *BitwiseXorExpression*$^{\beta}$] **do**
~~*Validate*[*BitwiseXorExpression*$^{\beta}$](*v*);~~*Validate*[*BitwiseXorExpression*$^{\beta}$](*c*, *e*);
[*BitwiseOrExpression*$^{\beta}$ ⇒ *BitwiseOrExpressionOrSuper*$^{\beta}$ **|** *BitwiseXorExpressionOrSuper*$^{\beta}$] **do**
~~*Validate*[*BitwiseOrExpressionOrSuper*$^{\beta}$](*v*);~~
~~*Validate*[*BitwiseXorExpressionOrSuper*$^{\beta}$](*v*)~~
~~**end proc**;~~

~~*Validate*[*BitwiseAndExpressionOrSuper*$^{\beta}$]: VALIDATIONENV → ();~~*Validate*[*BitwiseOrExpressionOrSuper*$^{\beta}$](*c*, *e*);
Validate[*BitwiseXorExpressionOrSuper*$^{\beta}$](*c*, *e*)
end proc;

Validate[*BitwiseAndExpressionOrSuper*$^{\beta}$]: CONTEXT × STATICENV → ();
Validate[*BitwiseAndExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseAndExpression*$^{\beta}$] = *Validate*[*BitwiseAndExpression*$^{\beta}$];
Validate[*BitwiseAndExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

~~*Validate*[*BitwiseXorExpressionOrSuper*$^{\beta}$]: VALIDATIONENV → ();~~*Validate*[*BitwiseXorExpressionOrSuper*$^{\beta}$]: CONTEXT × STATICENV → ();
Validate[*BitwiseXorExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseXorExpression*$^{\beta}$] = *Validate*[*BitwiseXorExpression*$^{\beta}$];
Validate[*BitwiseXorExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

~~*Validate*[*BitwiseOrExpressionOrSuper*$^{\beta}$]: VALIDATIONENV → ();~~*Validate*[*BitwiseOrExpressionOrSuper*$^{\beta}$]: CONTEXT × STATICENV → ();
Validate[*BitwiseOrExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseOrExpression*$^{\beta}$] = *Validate*[*BitwiseOrExpression*$^{\beta}$];
Validate[*BitwiseOrExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Validate*[*SuperExpression*];

Evaluation

~~**proc** *Eval*[*BitwiseAndExpression*$^{\beta}$] (*e*: DynamicEnv): ObjOrRef~~**proc** *Eval*[*BitwiseAndExpression*$^{\beta}$] (*e*: Environment): ObjOrRef
  [*BitwiseAndExpression*$^{\beta}$ ⇒ *EqualityExpression*$^{\beta}$] **do**
    return *Eval*[*EqualityExpression*$^{\beta}$](*e*);
  [*BitwiseAndExpression*$^{\beta}$ ⇒ *BitwiseAndExpressionOrSuper*$^{\beta}$ **&** *EqualityExpressionOrSuper*$^{\beta}$] **do**
    a: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*BitwiseAndExpressionOrSuper*$^{\beta}$](*e*));
    b: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*EqualityExpressionOrSuper*$^{\beta}$](*e*));
    return *binaryDispatch*(*bitwiseAndTable*, *a*, *b*)
end proc;

~~**proc** *Eval*[*BitwiseXorExpression*$^{\beta}$] (*e*: DynamicEnv): ObjOrRef~~**proc** *Eval*[*BitwiseXorExpression*$^{\beta}$] (*e*: Environment): ObjOrRef
  [*BitwiseXorExpression*$^{\beta}$ ⇒ *BitwiseAndExpression*$^{\beta}$] **do**
    return *Eval*[*BitwiseAndExpression*$^{\beta}$](*e*);
  [*BitwiseXorExpression*$^{\beta}$ ⇒ *BitwiseXorExpressionOrSuper*$^{\beta}$ **^** *BitwiseAndExpressionOrSuper*$^{\beta}$] **do**
    a: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*BitwiseXorExpressionOrSuper*$^{\beta}$](*e*));
    b: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*BitwiseAndExpressionOrSuper*$^{\beta}$](*e*));
    return *binaryDispatch*(*bitwiseXorTable*, *a*, *b*)
end proc;

~~**proc** *Eval*[*BitwiseOrExpression*$^{\beta}$] (*e*: DynamicEnv): ObjOrRef~~**proc** *Eval*[*BitwiseOrExpression*$^{\beta}$] (*e*: Environment): ObjOrRef
  [*BitwiseOrExpression*$^{\beta}$ ⇒ *BitwiseXorExpression*$^{\beta}$] **do**
    return *Eval*[*BitwiseXorExpression*$^{\beta}$](*e*);
  [*BitwiseOrExpression*$^{\beta}$ ⇒ *BitwiseOrExpressionOrSuper*$^{\beta}$ **|** *BitwiseXorExpressionOrSuper*$^{\beta}$] **do**
    a: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*BitwiseOrExpressionOrSuper*$^{\beta}$](*e*));
    b: ObjOptionalLimit ← *readRefWithLimit*(*Eval*[*BitwiseXorExpressionOrSuper*$^{\beta}$](*e*));
    return *binaryDispatch*(*bitwiseOrTable*, *a*, *b*)
end proc;

~~*Eval*[*BitwiseAndExpressionOrSuper*$^{\beta}$]: DynamicEnv → ObjOrRefOptionalLimit;~~*Eval*[*BitwiseAndExpressionOrSuper*$^{\beta}$]: Environment → ObjOrRefOptionalLimit;
  Eval[*BitwiseAndExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseAndExpression*$^{\beta}$] = *Eval*[*BitwiseAndExpression*$^{\beta}$];
  Eval[*BitwiseAndExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

~~*Eval*[*BitwiseXorExpressionOrSuper*$^{\beta}$]: DynamicEnv → ObjOrRefOptionalLimit;~~*Eval*[*BitwiseXorExpressionOrSuper*$^{\beta}$]: Environment → ObjOrRefOptionalLimit;
  Eval[*BitwiseXorExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseXorExpression*$^{\beta}$] = *Eval*[*BitwiseXorExpression*$^{\beta}$];
  Eval[*BitwiseXorExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

~~*Eval*[*BitwiseOrExpressionOrSuper*$^{\beta}$]: DynamicEnv → ObjOrRefOptionalLimit;~~*Eval*[*BitwiseOrExpressionOrSuper*$^{\beta}$]: Environment → ObjOrRefOptionalLimit;
  Eval[*BitwiseOrExpressionOrSuper*$^{\beta}$ ⇒ *BitwiseOrExpression*$^{\beta}$] = *Eval*[*BitwiseOrExpression*$^{\beta}$];
  Eval[*BitwiseOrExpressionOrSuper*$^{\beta}$ ⇒ *SuperExpression*] = *Eval*[*SuperExpression*];

Binary Logical Operators

Syntax

LogicalAndExpression$^{\beta}$ ⇒
    BitwiseOrExpression$^{\beta}$
  | *LogicalAndExpression*$^{\beta}$ **&&** *BitwiseOrExpression*$^{\beta}$

LogicalXorExpression$^{\beta}$ ⇒
    LogicalAndExpression$^{\beta}$
  | *LogicalXorExpression*$^{\beta}$ **^^** *LogicalAndExpression*$^{\beta}$

LogicalOrExpression$^{\beta}$ ⇒
    LogicalXorExpression$^{\beta}$
  | *LogicalOrExpression*$^{\beta}$ **||** *LogicalXorExpression*$^{\beta}$

Validation

proc *Validate*[*LogicalAndExpression*$^{\beta}$] (*v*: VALIDATIONENV)
[*LogicalAndExpression*$^{\beta}$ ⇒ *BitwiseOrExpression*$^{\beta}$] **do** *Validate*[*BitwiseOrExpression*$^{\beta}$](*v*);**proc** *Validate*[*LogicalAndExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
  [*LogicalAndExpression*$^{\beta}$ ⇒ *BitwiseOrExpression*$^{\beta}$] **do**
    Validate[*BitwiseOrExpression*$^{\beta}$](*c*, *e*);
  [*LogicalAndExpression*$^{\beta}_{0}$ ⇒ *LogicalAndExpression*$^{\beta}_{1}$ **&&** *BitwiseOrExpression*$^{\beta}$] **do**
    Validate[*LogicalAndExpression*$^{\beta}_{1}$](*v*);
    Validate[*BitwiseOrExpression*$^{\beta}$](*v*)
end proc;

    proc *Validate*[*LogicalXorExpression*$^{\beta}$] (*v*: VALIDATIONENV)*Validate*[*LogicalAndExpression*$^{\beta}_{1}$](*c*, *e*);
    Validate[*BitwiseOrExpression*$^{\beta}$](*c*, *e*)
end proc;

proc *Validate*[*LogicalXorExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
  [*LogicalXorExpression*$^{\beta}$ ⇒ *LogicalAndExpression*$^{\beta}$] **do**
    Validate[*LogicalAndExpression*$^{\beta}$](*v*);*Validate*[*LogicalAndExpression*$^{\beta}$](*c*, *e*);
  [*LogicalXorExpression*$^{\beta}_{0}$ ⇒ *LogicalXorExpression*$^{\beta}_{1}$ **^^** *LogicalAndExpression*$^{\beta}$] **do**
    Validate[*LogicalXorExpression*$^{\beta}_{1}$](*v*);
    Validate[*LogicalAndExpression*$^{\beta}$](*v*)
end proc;

    proc *Validate*[*LogicalOrExpression*$^{\beta}$] (*v*: VALIDATIONENV)*Validate*[*LogicalXorExpression*$^{\beta}_{1}$](*c*, *e*);
    Validate[*LogicalAndExpression*$^{\beta}$](*c*, *e*)
end proc;

proc *Validate*[*LogicalOrExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
  [*LogicalOrExpression*$^{\beta}$ ⇒ *LogicalXorExpression*$^{\beta}$] **do**
    Validate[*LogicalXorExpression*$^{\beta}$](*v*);*Validate*[*LogicalXorExpression*$^{\beta}$](*c*, *e*);
  [*LogicalOrExpression*$^{\beta}_{0}$ ⇒ *LogicalOrExpression*$^{\beta}_{1}$ **||** *LogicalXorExpression*$^{\beta}$] **do**
    Validate[*LogicalOrExpression*$^{\beta}_{1}$](*v*);
    Validate[*LogicalXorExpression*$^{\beta}$](*v*)*Validate*[*LogicalOrExpression*$^{\beta}_{1}$](*c*, *e*);
    Validate[*LogicalXorExpression*$^{\beta}$](*c*, *e*)
end proc;

Evaluation

proc *Eval*[*LogicalAndExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF**proc** *Eval*[*LogicalAndExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
  [*LogicalAndExpression*$^{\beta}$ ⇒ *BitwiseOrExpression*$^{\beta}$] **do**
    return *Eval*[*BitwiseOrExpression*$^{\beta}$](*e*);
  [*LogicalAndExpression*$^{\beta}_{0}$ ⇒ *LogicalAndExpression*$^{\beta}_{1}$ **&&** *BitwiseOrExpression*$^{\beta}$] **do**
    a: OBJECT ← *readReference*(*Eval*[*LogicalAndExpression*$^{\beta}_{1}$](*e*));
    if *toBoolean*(*a*) **then return** *readReference*(*Eval*[*BitwiseOrExpression*$^{\beta}$](*e*))
    else return *a*
    end if
end proc;

~~proc~~ ~~*Eval*[*LogicalXorExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*LogicalXorExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
[*LogicalXorExpression*$^{\beta}$ ⇒ *LogicalAndExpression*$^{\beta}$] **do**
return *Eval*[*LogicalAndExpression*$^{\beta}$](*e*);
[*LogicalXorExpression*$^{\beta}{}_0$ ⇒ *LogicalXorExpression*$^{\beta}{}_1$ **^^** *LogicalAndExpression*$^{\beta}$] **do**
a: OBJECT ← *readReference*(*Eval*[*LogicalXorExpression*$^{\beta}{}_1$](*e*));
b: OBJECT ← *readReference*(*Eval*[*LogicalAndExpression*$^{\beta}$](*e*));
ab: BOOLEAN ← *toBoolean*(*a*);
bb: BOOLEAN ← *toBoolean*(*b*);
return *ab* **xor** *bb*
end proc;

~~proc~~ ~~*Eval*[*LogicalOrExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*LogicalOrExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
[*LogicalOrExpression*$^{\beta}$ ⇒ *LogicalXorExpression*$^{\beta}$] **do**
return *Eval*[*LogicalXorExpression*$^{\beta}$](*e*);
[*LogicalOrExpression*$^{\beta}{}_0$ ⇒ *LogicalOrExpression*$^{\beta}{}_1$ **||** *LogicalXorExpression*$^{\beta}$] **do**
a: OBJECT ← *readReference*(*Eval*[*LogicalOrExpression*$^{\beta}{}_1$](*e*));
if *toBoolean*(*a*) **then return** *a*
else return *readReference*(*Eval*[*LogicalXorExpression*$^{\beta}$](*e*))
end if
end proc;

Conditional Operator

Syntax

ConditionalExpression$^{\beta}$ ⇒
LogicalOrExpression$^{\beta}$
| *LogicalOrExpression*$^{\beta}$ **?** *AssignmentExpression*$^{\beta}$ **:** *AssignmentExpression*$^{\beta}$

NonAssignmentExpression$^{\beta}$ ⇒
LogicalOrExpression$^{\beta}$
| *LogicalOrExpression*$^{\beta}$ **?** *NonAssignmentExpression*$^{\beta}$ **:** *NonAssignmentExpression*$^{\beta}$

Validation

~~proc~~ ~~*Validate*[*ConditionalExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~**proc** *Validate*[*ConditionalExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$] **do**
~~*Validate*[*LogicalOrExpression*$^{\beta}$](*v*);~~*Validate*[*LogicalOrExpression*$^{\beta}$](*c*, *e*);
[*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$ **?** *AssignmentExpression*$^{\beta}{}_1$ **:** *AssignmentExpression*$^{\beta}{}_2$] **do**
~~*Validate*[*LogicalOrExpression*$^{\beta}$](*v*);~~
~~*Validate*[*AssignmentExpression*$^{\beta}{}_1$](*v*);~~
~~*Validate*[*AssignmentExpression*$^{\beta}{}_2$](*v*)~~*Validate*[*LogicalOrExpression*$^{\beta}$](*c*, *e*);
Validate[*AssignmentExpression*$^{\beta}{}_1$](*c*, *e*);
Validate[*AssignmentExpression*$^{\beta}{}_2$](*c*, *e*)
end proc;

Evaluation

~~proc~~ ~~*Eval*[*ConditionalExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*ConditionalExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
[*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$] **do**
return *Eval*[*LogicalOrExpression*$^{\beta}$](*e*);

[*ConditionalExpression*$^{\beta}$ ⇒ *LogicalOrExpression*$^{\beta}$ **?** *AssignmentExpression*$^{\beta}{}_{1}$ **:** *AssignmentExpression*$^{\beta}{}_{2}$] **do**
if *toBoolean*(*readReference*(*Eval*[*LogicalOrExpression*$^{\beta}$](*e*))) **then**
return *Eval*[*AssignmentExpression*$^{\beta}{}_{1}$](*e*)
else return *Eval*[*AssignmentExpression*$^{\beta}{}_{2}$](*e*)
end if
end proc;

Assignment Operators

Syntax

AssignmentExpression$^{\beta}$ ⇒
ConditionalExpression$^{\beta}$
| *PostfixExpression* **=** *AssignmentExpression*$^{\beta}$
| *PostfixExpressionOrSuper CompoundAssignment AssignmentExpression*$^{\beta}$
| *PostfixExpressionOrSuper CompoundAssignment SuperExpression*
| *PostfixExpression LogicalAssignment AssignmentExpression*$^{\beta}$

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

LogicalAssignment ⇒
&&=
| **^^=**
| **||=**

Validation

~~**proc** *Validate*[*AssignmentExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~**proc** *Validate*[*AssignmentExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
[*AssignmentExpression*$^{\beta}$ ⇒ *ConditionalExpression*$^{\beta}$] **do**
~~*Validate*[*ConditionalExpression*$^{\beta}$](*v*);~~*Validate*[*ConditionalExpression*$^{\beta}$](*c*, *e*);
[*AssignmentExpression*$^{\beta}{}_{0}$ ⇒ *PostfixExpression* **=** *AssignmentExpression*$^{\beta}{}_{1}$] **do**
~~*Validate*[*PostfixExpression*](*v*);~~
~~*Validate*[*AssignmentExpression*$^{\beta}{}_{1}$](*v*);~~*Validate*[*PostfixExpression*](*c*, *e*);
Validate[*AssignmentExpression*$^{\beta}{}_{1}$](*c*, *e*);
[*AssignmentExpression*$^{\beta}{}_{0}$ ⇒ *PostfixExpressionOrSuper CompoundAssignment AssignmentExpression*$^{\beta}{}_{1}$] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*);~~
~~*Validate*[*AssignmentExpression*$^{\beta}{}_{1}$](*v*);~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*);
Validate[*AssignmentExpression*$^{\beta}{}_{1}$](*c*, *e*);
[*AssignmentExpression*$^{\beta}$ ⇒ *PostfixExpressionOrSuper CompoundAssignment SuperExpression*] **do**
~~*Validate*[*PostfixExpressionOrSuper*](*v*);~~
~~*Validate*[*SuperExpression*](*v*);~~*Validate*[*PostfixExpressionOrSuper*](*c*, *e*);
Validate[*SuperExpression*](*c*, *e*);

```
    [AssignmentExpression^β0 ⇒ PostfixExpression LogicalAssignment AssignmentExpression^β1] do
        ~~Validate[PostfixExpression](v);~~
        ~~Validate[AssignmentExpression^β1](v)~~Validate[PostfixExpression](c, e);
        Validate[AssignmentExpression^β1](c, e)
    end proc;
```

Evaluation

```
    ~~proc Eval[AssignmentExpression^β] (e: DynamicEnv): ObjOrRef~~proc Eval[AssignmentExpression^β] (e: Environment)
            : ObjOrRef
        [AssignmentExpression^β ⇒ ConditionalExpression^β] do
            return Eval[ConditionalExpression^β](e);
        [AssignmentExpression^β0 ⇒ PostfixExpression = AssignmentExpression^β1] do
            r: ObjOrRef ← Eval[PostfixExpression](e);
            a: Object ← readReference(Eval[AssignmentExpression^β1](e));
            writeReference(r, a);
            return a;
        [AssignmentExpression^β0 ⇒ PostfixExpressionOrSuper CompoundAssignment AssignmentExpression^β1] do
            return evalAssignmentOp(Table[CompoundAssignment], Eval[PostfixExpressionOrSuper],
                    Eval[AssignmentExpression^β1], e);
        [AssignmentExpression^β ⇒ PostfixExpressionOrSuper CompoundAssignment SuperExpression] do
            return evalAssignmentOp(Table[CompoundAssignment], Eval[PostfixExpressionOrSuper], Eval[SuperExpression],
                e);
        [AssignmentExpression^β ⇒ PostfixExpression LogicalAssignment AssignmentExpression^β] do
            ????
    end proc;

    Table[CompoundAssignment]: BinaryMethod{};
        Table[CompoundAssignment ⇒ *=] = multiplyTable;
        Table[CompoundAssignment ⇒ /=] = divideTable;
        Table[CompoundAssignment ⇒ %=] = remainderTable;
        Table[CompoundAssignment ⇒ +=] = addTable;
        Table[CompoundAssignment ⇒ -=] = subtractTable;
        Table[CompoundAssignment ⇒ <<=] = shiftLeftTable;
        Table[CompoundAssignment ⇒ >>=] = shiftRightTable;
        Table[CompoundAssignment ⇒ >>>=] = shiftRightUnsignedTable;
        Table[CompoundAssignment ⇒ &=] = bitwiseAndTable;
        Table[CompoundAssignment ⇒ ^=] = bitwiseXorTable;
        Table[CompoundAssignment ⇒ |=] = bitwiseOrTable;

    proc evalAssignmentOp(table: BinaryMethod{}, ~~leftEval: DynamicEnv → ObjOrRefOptionalLimit,~~
            ~~rightEval: DynamicEnv → ObjOrRefOptionalLimit,~~
            ~~e: DynamicEnv):~~leftEval: Environment → ObjOrRefOptionalLimit,
            rightEval: Environment → ObjOrRefOptionalLimit, e: Environment): ObjOrRef
        rLeft: ObjOrRefOptionalLimit ← leftEval(e);
        oLeft: ObjOptionalLimit ← readRefWithLimit(rLeft);
        oRight: ObjOptionalLimit ← readRefWithLimit(rightEval(e));
        result: Object ← binaryDispatch(table, oLeft, oRight);
        writeReference(rLeft, result);
        return result
    end proc;
```

Comma Expressions

Syntax

ListExpression$^{\beta}$ ⇒
    AssignmentExpression$^{\beta}$
  | *ListExpression*$^{\beta}$, *AssignmentExpression*$^{\beta}$

OptionalExpression ⇒
    ListExpression^{allowIn}
  | «empty»

Validation

~~**proc** *Validate*[*ListExpression*$^{\beta}$] (*v*: VALIDATIONENV)~~
~~[*ListExpression*$^{\beta}$ ⇒ *AssignmentExpression*$^{\beta}$] **do** *Validate*[*AssignmentExpression*$^{\beta}$](*v*);~~**proc** *Validate*[*ListExpression*$^{\beta}$] (*c*: CONTEXT, *e*: STATICENV)
  [*ListExpression*$^{\beta}$ ⇒ *AssignmentExpression*$^{\beta}$] **do** *Validate*[*AssignmentExpression*$^{\beta}$](*c*, *e*);
  [*ListExpression*$^{\beta}_0$ ⇒ *ListExpression*$^{\beta}_1$, *AssignmentExpression*$^{\beta}$] **do**
    ~~*Validate*[*ListExpression*$^{\beta}_1$](*v*);~~
    ~~*Validate*[*AssignmentExpression*$^{\beta}$](*v*)~~*Validate*[*ListExpression*$^{\beta}_1$](*c*, *e*);
    Validate[*AssignmentExpression*$^{\beta}$](*c*, *e*)
end proc;

Evaluation

~~**proc** *Eval*[*ListExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJORREF~~**proc** *Eval*[*ListExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJORREF
  [*ListExpression*$^{\beta}$ ⇒ *AssignmentExpression*$^{\beta}$] **do return** *Eval*[*AssignmentExpression*$^{\beta}$](*e*);
  [*ListExpression*$^{\beta}_0$ ⇒ *ListExpression*$^{\beta}_1$, *AssignmentExpression*$^{\beta}$] **do**
    readReference(*Eval*[*ListExpression*$^{\beta}_1$](*e*));
    return *readReference*(*Eval*[*AssignmentExpression*$^{\beta}$](*e*))
end proc;

~~**proc** *EvalAsList*[*ListExpression*$^{\beta}$] (*e*: DYNAMICENV): OBJECT[]~~**proc** *EvalAsList*[*ListExpression*$^{\beta}$] (*e*: ENVIRONMENT): OBJECT[]
  [*ListExpression*$^{\beta}$ ⇒ *AssignmentExpression*$^{\beta}$] **do**
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^{\beta}$](*e*));
    return [*elt*];
  [*ListExpression*$^{\beta}_0$ ⇒ *ListExpression*$^{\beta}_1$, *AssignmentExpression*$^{\beta}$] **do**
    elts: OBJECT[] ← *EvalAsList*[*ListExpression*$^{\beta}_1$](*e*);
    elt: OBJECT ← *readReference*(*Eval*[*AssignmentExpression*$^{\beta}$](*e*));
    return *elts* ⊕ [*elt*]
end proc;

Type Expressions

Syntax

TypeExpression$^{\beta}$ ⇒ *NonAssignmentExpression*$^{\beta}$

Statements

Syntax

ω ∈ {abbrev, noShortIf, full}

*Statement*ω ⇒
    EmptyStatement
  | *ExpressionStatement Semicolon*ω
  | *SuperStatement Semicolon*ω
  | *AnnotatedBlock*
  | *LabeledStatement*ω
  | *IfStatement*ω
  | *SwitchStatement*
  | *DoStatement Semicolon*ω
  | *WhileStatement*ω
  | *ForStatement*ω
  | *WithStatement*ω
  | *ContinueStatement Semicolon*ω
  | *BreakStatement Semicolon*ω
  | *ReturnStatement Semicolon*ω
  | *ThrowStatement Semicolon*ω
  | *TryStatement*

*Substatement*ω ⇒
    *Statement*ω
  | *SimpleVariableDefinition Semicolon*ω

*Semicolon*abbrev ⇒
    ;
  | **VirtualSemicolon**
  | «empty»

*Semicolon*noShortIf ⇒
    ;
  | **VirtualSemicolon**
  | «empty»

*Semicolon*full ⇒
    ;
  | **VirtualSemicolon**

Validation

proc ~~*Validate*[*Statement*ω] (*v*: VALIDATIONENV)~~**proc** *Validate*[*Statement*ω] (*c*: CONTEXT, *e*: STATICENV, *a*: ATTRIBUTENOT FALSE, *sl*: LABEL{})
  [*Statement*ω ⇒ *EmptyStatement*] **do nothing**;
  ~~[*Statement*ω ⇒ *ExpressionStatement Semicolon*ω] **do** *Validate*[*ExpressionStatement*](*v*);~~
  ~~[*Statement*ω ⇒ *SuperStatement Semicolon*ω] **do** ????;~~
  ~~[*Statement*ω ⇒ *AnnotatedBlock*] **do** *Validate*[*AnnotatedBlock*](*v*);~~
  ~~[*Statement*ω ⇒ *LabeledStatement*ω] **do** *Validate*[*LabeledStatement*ω](*v*);~~
  ~~[*Statement*ω ⇒ *IfStatement*ω] **do** *Validate*[*IfStatement*ω](*v*);~~
  ~~[*Statement*ω ⇒ *SwitchStatement*] **do** ????;~~
  ~~[*Statement*ω ⇒ *DoStatement Semicolon*ω] **do** ????;~~
  ~~[*Statement*ω ⇒ *WhileStatement*ω] **do** ????;~~
  ~~[*Statement*ω ⇒ *ForStatement*ω] **do** ????;~~
  ~~[*Statement*ω ⇒ *WithStatement*ω] **do** ????;~~
  ~~[*Statement*ω ⇒ *ContinueStatement Semicolon*ω] **do** *Validate*[*ContinueStatement*](*v*);~~
  ~~[*Statement*ω ⇒ *BreakStatement Semicolon*ω] **do** *Validate*[*BreakStatement*](*v*);~~
  ~~[*Statement*ω ⇒ *ReturnStatement Semicolon*ω] **do** *Validate*[*ReturnStatement*](*v*);~~
  ~~[*Statement*ω ⇒ *ThrowStatement Semicolon*ω] **do** ????;~~

~~[*Statement*$^{\omega}$ ⇒ *TryStatement*] **do** ????~~
~~**end proc**;~~

~~**proc** *Validate*[*Substatement*$^{\omega}$] (*v*: VALIDATIONENV)~~
~~[*Substatement*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do** *Validate*[*Statement*$^{\omega}$](*v*);~~[*Statement*$^{\omega}$ ⇒ *ExpressionStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*ExpressionStatement*](*c*, *e*)
else throw syntaxError
end if;
[*Statement*$^{\omega}$ ⇒ *SuperStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*SuperStatement*](*c*, *e*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *AnnotatedBlock*] **do** *Validate*[*AnnotatedBlock*](*c*, *e*, *a*);
[*Statement*$^{\omega}$ ⇒ *LabeledStatement*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*LabeledStatement*$^{\omega}$](*c*, *e*, *sl*)
else throw syntaxError
end if;
[*Statement*$^{\omega}$ ⇒ *IfStatement*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*IfStatement*$^{\omega}$](*c*, *e*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *SwitchStatement*] **do**
if *a* = **true then** ???? **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *DoStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*DoStatement*](*c*, *e*, *sl*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *WhileStatement*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*WhileStatement*$^{\omega}$](*c*, *e*, *sl*)
else throw syntaxError
end if;
[*Statement*$^{\omega}$ ⇒ *ForStatement*$^{\omega}$] **do if** *a* = **true then** ???? **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *WithStatement*$^{\omega}$] **do**
if *a* = **true then** ???? **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *ContinueStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*ContinueStatement*](*c*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *BreakStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*BreakStatement*](*c*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *ReturnStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*ReturnStatement*](*c*, *e*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *ThrowStatement Semicolon*$^{\omega}$] **do**
if *a* = **true then** *Validate*[*ThrowStatement*](*c*, *e*) **else throw syntaxError end if**;
[*Statement*$^{\omega}$ ⇒ *TryStatement*] **do if** *a* = **true then** ???? **else throw syntaxError end if**
end proc;

proc *Validate*[*Substatement*$^{\omega}$] (*c*: CONTEXT, *e*: STATICENV, *sl*: LABEL{})
[*Substatement*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do** *Validate*[*Statement*$^{\omega}$](*c*, *e*, **true**, *sl*);
[*Substatement*$^{\omega}$ ⇒ *SimpleVariableDefinition Semicolon*$^{\omega}$] **do** ????
end proc;

Evaluation

proc *Eval*[*Statement*$^{\omega}$] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
[*Statement*$^{\omega}$ ⇒ *EmptyStatement*] **do return** *d*;
[*Statement*$^{\omega}$ ⇒ *ExpressionStatement Semicolon*$^{\omega}$] **do**
return *Eval*[*ExpressionStatement*](*e*);
~~[*Statement*$^{\omega}$ ⇒ *SuperStatement Semicolon*$^{\omega}$] **do** ????;~~[*Statement*$^{\omega}$ ⇒ *SuperStatement Semicolon*$^{\omega}$] **do return** *Eval*[*SuperStatement*](*e*);
[*Statement*$^{\omega}$ ⇒ *AnnotatedBlock*] **do return** *Eval*[*AnnotatedBlock*](*e*, *d*);

[*Statement*$^{\omega}$ ⇒ *LabeledStatement*$^{\omega}$] **do return** *Eval*[*LabeledStatement*$^{\omega}$](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *IfStatement*$^{\omega}$] **do return** *Eval*[*IfStatement*$^{\omega}$](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *SwitchStatement*] **do** ????;
~~[*Statement*$^{\omega}$ ⇒ *DoStatement Semicolon*$^{\omega}$] **do** ????;~~
~~[*Statement*$^{\omega}$ ⇒ *WhileStatement*$^{\omega}$] **do** ????;~~[*Statement*$^{\omega}$ ⇒ *DoStatement Semicolon*$^{\omega}$] **do return** *Eval*[*DoStatement*](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *WhileStatement*$^{\omega}$] **do return** *Eval*[*WhileStatement*$^{\omega}$](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *ForStatement*$^{\omega}$] **do** ????;
[*Statement*$^{\omega}$ ⇒ *WithStatement*$^{\omega}$] **do** ????;
[*Statement*$^{\omega}$ ⇒ *ContinueStatement Semicolon*$^{\omega}$] **do return** *Eval*[*ContinueStatement*](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *BreakStatement Semicolon*$^{\omega}$] **do return** *Eval*[*BreakStatement*](*e*, *d*);
[*Statement*$^{\omega}$ ⇒ *ReturnStatement Semicolon*$^{\omega}$] **do return** *Eval*[*ReturnStatement*](*e*);
~~[*Statement*$^{\omega}$ ⇒ *ThrowStatement Semicolon*$^{\omega}$] **do** ????;~~[*Statement*$^{\omega}$ ⇒ *ThrowStatement Semicolon*$^{\omega}$] **do return** *Eval*[*ThrowStatement*](*e*);
[*Statement*$^{\omega}$ ⇒ *TryStatement*] **do** ????
end proc;

proc *Eval*[*Substatement*$^{\omega}$] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
[*Substatement*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do return** *Eval*[*Statement*$^{\omega}$](*e*, *d*);
[*Substatement*$^{\omega}$ ⇒ *SimpleVariableDefinition Semicolon*$^{\omega}$] **do** ????
end proc;

Empty Statement

Syntax

EmptyStatement ⇒ **;**

Expression Statement

Syntax

ExpressionStatement ⇒ [lookahead∉ {**function**, **{**}] *ListExpression*$^{\text{allowIn}}$

Validation

proc *Validate*[*ExpressionStatement* ⇒ [lookahead∉ {**function**, **{**}] *ListExpression*$^{\text{allowIn}}$]
~~(*v*: VALIDATIONENV)~~(*c*: CONTEXT, *e*: STATICENV)
~~*Validate*[*ListExpression*$^{\text{allowIn}}$]~~(*v*)*Validate*[*ListExpression*$^{\text{allowIn}}$](*c*, *e*)
end proc;

Evaluation

proc *Eval*[*ExpressionStatement* ⇒ [lookahead∉ {**function**, **{**}] *ListExpression*$^{\text{allowIn}}$] (*e*: DYNAMICENV): OBJECT
return *readReference*(*Eval*[*ListExpression*$^{\text{allowIn}}$](*e*))
end proc;

Super Statement

Syntax

SuperStatement ⇒ **super** *Arguments*

Validation

proc *Validate*[*SuperStatement* ⇒ **super** *Arguments*] (*c*: CONTEXT, *e*: STATICENV)
????
end proc;

Evaluation

proc *Eval*[*SuperStatement* ⇒ **super** *Arguments*] (*e*: DYNAMICENV): OBJECT
????
end proc;

Block Statement

Syntax

AnnotatedBlock ⇒ *Attributes Block*

Block ⇒ **{** *Directives* **}**

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

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

Validation

~~*Validate*[*AnnotatedBlock* ⇒ *Attributes Block*]: VALIDATIONENV → () = *Validate*[*Block*];~~

~~*Validate*[*Block* ⇒ **{** *Directives* **}**]: VALIDATIONENV → () = *Validate*[*Directives*];~~

~~**proc** *Validate*[*Directives*] (*v*: VALIDATIONENV)~~*Attribute*[*AnnotatedBlock*]: ATTRIBUTE;

proc *Validate*[*AnnotatedBlock* ⇒ *Attributes Block*] (*c*: CONTEXT, *e*: STATICENV, *a*: ATTRIBUTENOTFALSE)
Validate[*Attributes*](*c*, *e*);
a2: ATTRIBUTE ← *Eval*[*Attributes*](*e*);
a3: ATTRIBUTE ← *combineAttributes*(*a*, *a2*);
Attribute[*AnnotatedBlock*] ← *a3*;
if *a3* ≠ **false then** *Validate*[*Block*](*c*, *e*, *a3*) **end if**
end proc;

Validate[*Block* ⇒ **{** *Directives* **}**]: CONTEXT × STATICENV × ATTRIBUTENOTFALSE → () = *Validate*[*Directives*];

proc *Validate*[*Directives*] (*c*: CONTEXT, *e*: STATICENV, *a*: ATTRIBUTENOTFALSE)
[*Directives* ⇒ «empty»] **do nothing**;
[*Directives* ⇒ *DirectivesPrefix Directive*[abbrev]] **do**
~~*Validate*[*DirectivesPrefix*](*v*);~~
~~*Validate*[*Directive*[abbrev]](*v*)~~*c2*: CONTEXT ← *Validate*[*DirectivesPrefix*](*c*, *e*, *a*);
Validate[*Directive*[abbrev]](*c2*, *e*, *a*)
end proc;

~~**proc** *Validate*[*DirectivesPrefix*] (*v*: VALIDATIONENV)~~
~~[*DirectivesPrefix* ⇒ «empty»] **do nothing**;~~**proc** *Validate*[*DirectivesPrefix*] (*c*: CONTEXT, *e*: STATICENV, *a*: ATTRIBUTENOTFALSE): CONTEXT
[*DirectivesPrefix* ⇒ «empty»] **do return** *c*;

[*DirectivesPrefix*$_0$ ⇒ *DirectivesPrefix*$_1$ *Directive*^{full}] **do**
~~*Validate*[~~*~~DirectivesPrefix~~*$_1$~~]~~(*v*);~~
~~*Validate*[~~*~~Directive~~*^{full}~~]~~(*v*)~~*c2*: CONTEXT ← *Validate*[*DirectivesPrefix*$_1$](*c*, *e*, *a*);
return *Validate*[*Directive*^{full}](*c2*, *e*, *a*)
end proc;

Evaluation

~~*Eval*[*AnnotatedBlock* ⇒ *Attributes Block*]: DYNAMICENV × OBJECT → OBJECT = *Eval*[*Block*];~~**proc** *Eval*[*AnnotatedBlock* ⇒ *Attributes Block*] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
if *Attribute*[*AnnotatedBlock*] = **false then return** *d*
else return *Eval*[*Block*](*e*, *d*)
end if
end proc;

Eval[*Block* ⇒ **{** *Directives* **}**]: DYNAMICENV × OBJECT → OBJECT = *Eval*[*Directives*];

proc *Eval*[*Directives*] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
[*Directives* ⇒ «empty»] **do return** *d*;
[*Directives* ⇒ *DirectivesPrefix Directive*^{abbrev}] **do**
o: OBJECT ← *Eval*[*DirectivesPrefix*](*e*, *d*);
return *Eval*[*Directive*^{abbrev}](*e*, *o*)
end proc;

proc *Eval*[*DirectivesPrefix*] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
[*DirectivesPrefix* ⇒ «empty»] **do return** *d*;
[*DirectivesPrefix*$_0$ ⇒ *DirectivesPrefix*$_1$ *Directive*^{full}] **do**
o: OBJECT ← *Eval*[*DirectivesPrefix*$_1$](*e*, *d*);
return *Eval*[*Directive*^{full}](*e*, *o*)
end proc;

Labeled Statements

Syntax

LabeledStatement$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$

Validation

~~**proc** *Validate*[*LabeledStatement*$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$] (*v*: VALIDATIONENV)~~
~~*Validate*[*Substatement*$^{\omega}$](*addLabel*(*v*, *Name*[*Identifier*]))~~**proc** *Validate*[*LabeledStatement*$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$] (*c*: CONTEXT, *e*: STATICENV, *sl*: LABEL{}) *name*: STRING ← *Name*[*Identifier*];
c2: CONTEXT ← *addBreakLabel*(*c*, *name*);
Validate[*Substatement*$^{\omega}$](*c2*, *e*, *sl* ∪ {*name*})
end proc;

Evaluation

proc *Eval*[*LabeledStatement*$^{\omega}$ ⇒ *Identifier* **:** *Substatement*$^{\omega}$] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
try return *Eval*[*Substatement*$^{\omega}$](*e*, *d*)
catch *x*: SEMANTICEXCEPTION **do**
if *x* ∈ GOBREAK **and** *x*.label = *Name*[*Identifier*] **then return** *x*.value
else throw *x*
end if
end try
end proc;

If Statement

Syntax

IfStatement^{abbrev} ⇒
    if *ParenListExpression Substatement*^{abbrev}
  | **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{abbrev}

IfStatement^{full} ⇒
    if *ParenListExpression Substatement*^{full}
  | **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{full}

IfStatement^{noShortIf} ⇒ **if** *ParenListExpression Substatement*^{noShortIf} **else** *Substatement*^{noShortIf}

Validation

~~**proc** *Validate*[*IfStatement*$^{\omega}$] (*v*: VALIDATIONENV)~~**proc** *Validate*[*IfStatement*$^{\omega}$] (*c*: CONTEXT, *e*: STATICENV)
  [*IfStatement*^{abbrev} ⇒ **if** *ParenListExpression Substatement*^{abbrev}] **do**
    ~~*Validate*[*ParenListExpression*](*v*);~~
    ~~*Validate*[*Substatement*^{abbrev}](*v*);~~*Validate*[*ParenListExpression*](*c*, *e*);
    Validate[*Substatement*^{abbrev}](*c*, *e*, {});
  [*IfStatement*^{full} ⇒ **if** *ParenListExpression Substatement*^{full}] **do**
    ~~*Validate*[*ParenListExpression*](*v*);~~
    ~~*Validate*[*Substatement*^{full}](*v*);~~*Validate*[*ParenListExpression*](*c*, *e*);
    Validate[*Substatement*^{full}](*c*, *e*, {});
  [*IfStatement*$^{\omega}$ ⇒ **if** *ParenListExpression Substatement*$^{noShortIf}_{1}$ **else** *Substatement*$^{\omega}_{2}$] **do**
    ~~*Validate*[*ParenListExpression*](*v*);~~
    ~~*Validate*[*Substatement*$^{noShortIf}_{1}$](*v*);~~
    ~~*Validate*[*Substatement*$^{\omega}_{2}$](*v*)~~*Validate*[*ParenListExpression*](*c*, *e*);
    Validate[*Substatement*$^{noShortIf}_{1}$](*c*, *e*, {});
    Validate[*Substatement*$^{\omega}_{2}$](*c*, *e*, {})
end proc;

Evaluation

proc *Eval*[*IfStatement*$^{\omega}$] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
  [*IfStatement*^{abbrev} ⇒ **if** *ParenListExpression Substatement*^{abbrev}] **do**
    if *toBoolean*(*readReference*(*Eval*[*ParenListExpression*](*e*))) **then**
      return *Eval*[*Substatement*^{abbrev}](*e*, *d*)
    else return *d*
    end if;
  [*IfStatement*^{full} ⇒ **if** *ParenListExpression Substatement*^{full}] **do**
    if *toBoolean*(*readReference*(*Eval*[*ParenListExpression*](*e*))) **then**
      return *Eval*[*Substatement*^{full}](*e*, *d*)
    else return *d*
    end if;
  [*IfStatement*$^{\omega}$ ⇒ **if** *ParenListExpression Substatement*$^{noShortIf}_{1}$ **else** *Substatement*$^{\omega}_{2}$] **do**
    if *toBoolean*(*readReference*(*Eval*[*ParenListExpression*](*e*))) **then**
      return *Eval*[*Substatement*$^{noShortIf}_{1}$](*e*, *d*)
    else return *Eval*[*Substatement*$^{\omega}_{2}$](*e*, *d*)
    end if
end proc;

Switch Statement

Syntax

SwitchStatement ⇒ **`switch`** *ParenListExpression* **`{`** *CaseStatements* **`}`**

CaseStatements ⇒
«empty»
| *CaseLabel*
| *CaseLabel CaseStatementsPrefix CaseStatement*^{abbrev}

CaseStatementsPrefix ⇒
«empty»
| *CaseStatementsPrefix CaseStatement*^{full}

CaseStatement$^{\omega}$ ⇒
Substatement$^{\omega}$
| *CaseLabel*

CaseLabel ⇒
`case` *ListExpression*^{allowIn} **`:`**
| **`default :`**

Do-While Statement

Syntax

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

Validation

Labels[*DoStatement*]: LABEL{};

proc *Validate*[*DoStatement* ⇒ **do** *Substatement*^{abbrev} **`while`** *ParenListExpression*]
(*c*: CONTEXT, *e*: STATICENV, *sl*: LABEL{})
continueLabels: LABEL{} ← *sl* ∪ {**default**};
Labels[*DoStatement*] ← *continueLabels*;
c2: CONTEXT ← *addBreakLabel*(*c*, **default**);
c3: CONTEXT ← *addContinueLabels*(*c2*, *continueLabels*);
Validate[*Substatement*^{abbrev}](*c3*, *e*, {});
Validate[*ParenListExpression*](*c*, *e*)
end proc;

Evaluation

```
proc Eval[DoStatement ⇒ do Substatement^abbrev while ParenListExpression] (e: DYNAMICENV, d: OBJECT): OBJECT
    try
        d1: OBJECT ← d;
        while true do
            try d1 ← Eval[Substatement^abbrev](e, d1)
            catch x: SEMANTICEXCEPTION do
                if x ∈ GOCONTINUE and x.label ∈ Labels[DoStatement] then d1 ← x.value
                else throw x
                end if
            end try;
            if not toBoolean(readReference(Eval[ParenListExpression](e))) then return d1
            end if
        end while
    catch x: SEMANTICEXCEPTION do
        if x ∈ GOBREAK and x.label = default then return x.value else throw x end if
    end try
end proc;
```

While Statement

Syntax

WhileStatement$^{\omega}$ ⇒ **while** *ParenListExpression Substatement*$^{\omega}$

Validation

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

proc Validate[WhileStatement^ω ⇒ while ParenListExpression Substatement^ω]
        (c: CONTEXT, e: STATICENV, sl: LABEL{})
    Validate[ParenListExpression](c, e);
    continueLabels: LABEL{} ← sl ∪ {default};
    Labels[WhileStatement^ω] ← continueLabels;
    c2: CONTEXT ← addBreakLabel(c, default);
    c3: CONTEXT ← addContinueLabels(c2, continueLabels);
    Validate[Substatement^ω](c3, e, {})
end proc;
```

Evaluation

```
proc Eval[WhileStatement^ω ⇒ while ParenListExpression Substatement^ω] (e: DYNAMICENV, d: OBJECT): OBJECT
    try
        d1: OBJECT ← d;
        while toBoolean(readReference(Eval[ParenListExpression](e))) do
            try d1 ← Eval[Substatement^ω](e, d1)
            catch x: SEMANTICEXCEPTION do
                if x ∈ GOCONTINUE and x.label ∈ Labels[WhileStatement^ω] then
                    d1 ← x.value
                else throw x
                end if
            end try
        end while;
        return d1
    catch x: SEMANTICEXCEPTION do
        if x ∈ GOBREAK and x.label = default then return x.value else throw x end if
    end try
end proc;
```

For Statements

Syntax

ForStatement^ω ⇒
    for (*ForInitialiser* **;** *OptionalExpression* **;** *OptionalExpression* **)** *Substatement*^ω
  | **for (** *ForInBinding* **in** *ListExpression*^allowIn **)** *Substatement*^ω

ForInitialiser ⇒
    «empty»
  | *ListExpression*^noIn
  | *Attributes VariableDefinitionKind VariableBindingList*^noIn

ForInBinding ⇒
    PostfixExpression
  | *Attributes VariableDefinitionKind VariableBinding*^noIn

With Statement

Syntax

WithStatement^ω ⇒ **with** *ParenListExpression Substatement*^ω

Continue and Break Statements

Syntax

ContinueStatement ⇒
    continue
  | **continue** [no line break] *Identifier*

BreakStatement ⇒
    break
  | **break** [no line break] *Identifier*

Validation

~~**proc** *Validate*[*ContinueStatement*] (*v*: VALIDATIONENV)~~
~~[*ContinueStatement* ⇒ **continue**] **do** ????;~~
~~[*ContinueStatement* ⇒ **continue** [no line break] *Identifier*] **do** ????~~
~~**end proc**;~~

~~**proc** *Validate*[*BreakStatement*] (*v*: VALIDATIONENV)~~
~~[*BreakStatement* ⇒ **break**] **do** ????;~~
~~[*BreakStatement* ⇒ **break** [no line break] *Identifier*] **do** ????~~**proc** *Validate*[*ContinueStatement*] (*c*: CONTEXT)
[*ContinueStatement* ⇒ **continue**] **do**
if default ∉ *c*.continueLabels **then throw syntaxError end if**;
[*ContinueStatement* ⇒ **continue** [no line break] *Identifier*] **do**
if *Name*[*Identifier*] ∉ *c*.continueLabels **then throw syntaxError end if**
end proc;

proc *Validate*[*BreakStatement*] (*c*: CONTEXT)
[*BreakStatement* ⇒ **break**] **do**
if default ∉ *c*.breakLabels **then throw syntaxError end if**;
[*BreakStatement* ⇒ **break** [no line break] *Identifier*] **do**
if *Name*[*Identifier*] ∉ *c*.breakLabels **then throw syntaxError end if**
end proc;

Evaluation

proc *Eval*[*ContinueStatement*] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
~~[*ContinueStatement* ⇒ **continue**] **do throw** GOCONTINUE⟨*d*, “”⟩;~~[*ContinueStatement* ⇒ **continue**] **do throw** GOCONTINUE⟨*d*, **default**⟩;
[*ContinueStatement* ⇒ **continue** [no line break] *Identifier*] **do**
throw GOCONTINUE⟨*d*, *Name*[*Identifier*]⟩
end proc;

proc *Eval*[*BreakStatement*] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
~~[*BreakStatement* ⇒ **break**] **do throw** GOBREAK⟨*d*, “”⟩;~~[*BreakStatement* ⇒ **break**] **do throw** GOBREAK⟨*d*, **default**⟩;
[*BreakStatement* ⇒ **break** [no line break] *Identifier*] **do**
throw GOBREAK⟨*d*, *Name*[*Identifier*]⟩
end proc;

Return Statement

Syntax

ReturnStatement ⇒
return
| **return** [no line break] *ListExpression*^{allowIn}

Validation

~~**proc** *Validate*[*ReturnStatement*] (*v*: VALIDATIONENV)~~
~~[*ReturnStatement* ⇒ **return**] **do if not** *v*.canReturn **then throw syntaxError end if**;~~**proc** *Validate*[*ReturnStatement*] (*c*: CONTEXT, *e*: STATICENV)
[*ReturnStatement* ⇒ **return**] **do if not** *c*.insideFunction **then throw syntaxError end if**;

[*ReturnStatement* ⇒ **return** [no line break] *ListExpression*$^{\text{allowIn}}$] **do**
~~**if not** *v*.canReturn **then throw syntaxError end if**;~~
~~*Validate*[*ListExpression*$^{\text{allowIn}}$](*v*)~~**if not** *c*.insideFunction **then throw syntaxError end if**;
Validate[*ListExpression*$^{\text{allowIn}}$](*c*, *e*)
end proc;

Evaluation

proc *Eval*[*ReturnStatement*] (*e*: DYNAMICENV): OBJECT
[*ReturnStatement* ⇒ **return**] **do throw** GORETURN⟨**undefined**⟩;
[*ReturnStatement* ⇒ **return** [no line break] *ListExpression*$^{\text{allowIn}}$] **do**
throw GORETURN⟨*readReference*(*Eval*[*ListExpression*$^{\text{allowIn}}$](*e*))⟩
end proc;

Throw Statement

Syntax

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

Validation

Validate[*ThrowStatement* ⇒ **throw** [no line break] *ListExpression*$^{\text{allowIn}}$]: CONTEXT × STATICENV → ()
= *Validate*[*ListExpression*$^{\text{allowIn}}$];

Evaluation

proc *Eval*[*ThrowStatement* ⇒ **throw** [no line break] *ListExpression*$^{\text{allowIn}}$] (*e*: DYNAMICENV): OBJECT
throw GOTHROW⟨*readReference*(*Eval*[*ListExpression*$^{\text{allowIn}}$](*e*))⟩
end proc;

Try Statement

Syntax

TryStatement ⇒
try *Block CatchClauses*
| **try** *Block FinallyClause*
| **try** *Block CatchClauses FinallyClause*

CatchClauses ⇒
CatchClause
| *CatchClauses CatchClause*

CatchClause ⇒ **catch** **(** *Parameter* **)** *Block*

FinallyClause ⇒ **finally** *Block*

Directives

Syntax

Directive$^{\omega}$ ⇒
  Statement$^{\omega}$
| *AnnotatableDirective*$^{\omega}$
| *Attribute* [no line break] *Attributes AnnotatableDirective*$^{\omega}$
| *PackageDefinition*
| *IncludeDirective Semicolon*$^{\omega}$
| *Pragma Semicolon*$^{\omega}$

AnnotatableDirective$^{\omega}$ ⇒
  ExportDefinition Semicolon$^{\omega}$
| *VariableDefinition Semicolon*$^{\omega}$
| *FunctionDefinition*$^{\omega}$
| *ClassDefinition*$^{\omega}$
| *NamespaceDefinition Semicolon*$^{\omega}$
| *InterfaceDefinition*$^{\omega}$
| *UseDirective Semicolon*$^{\omega}$
| *ImportDirective Semicolon*$^{\omega}$

Validation

~~**proc** *Validate*[*Directive*$^{\omega}$] (*v*: VALIDATIONENV)~~
~~[*Directive*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do** *Validate*[*Statement*$^{\omega}$](*v*);~~**proc** *Validate*[*Directive*$^{\omega}$] (*c*: CONTEXT, *e*: STATICENV, *a*: ATTRIBUTENOTFALSE): CONTEXT
  [*Directive*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do** *Validate*[*Statement*$^{\omega}$](*c*, *e*, *a*, {}); **return** *c*;
  [*Directive*$^{\omega}$ ⇒ *AnnotatableDirective*$^{\omega}$] **do** ????;
  [*Directive*$^{\omega}$ ⇒ *Attribute* [no line break] *Attributes AnnotatableDirective*$^{\omega}$] **do** ????;
  ~~[*Directive*$^{\omega}$ ⇒ *PackageDefinition*] **do** ????;~~
  ~~[*Directive*$^{\omega}$ ⇒ *IncludeDirective Semicolon*$^{\omega}$] **do** ????;~~[*Directive*$^{\omega}$ ⇒ *PackageDefinition*] **do**
    if *a* = **true then** ???? **else throw syntaxError end if**;
  [*Directive*$^{\omega}$ ⇒ *IncludeDirective Semicolon*$^{\omega}$] **do**
    if *a* = **true then** ???? **else throw syntaxError end if**;
  [*Directive*$^{\omega}$ ⇒ *Pragma Semicolon*$^{\omega}$] **do** ????
end proc;

Evaluation

proc *Eval*[*Directive*$^{\omega}$] (*e*: DYNAMICENV, *d*: OBJECT): OBJECT
  [*Directive*$^{\omega}$ ⇒ *Statement*$^{\omega}$] **do return** *Eval*[*Statement*$^{\omega}$](*e*, *d*);
  [*Directive*$^{\omega}$ ⇒ *AnnotatableDirective*$^{\omega}$] **do** ????;
  [*Directive*$^{\omega}$ ⇒ *Attribute* [no line break] *Attributes AnnotatableDirective*$^{\omega}$] **do** ????;
  [*Directive*$^{\omega}$ ⇒ *PackageDefinition*] **do** ????;
  [*Directive*$^{\omega}$ ⇒ *IncludeDirective Semicolon*$^{\omega}$] **do** ????;
  [*Directive*$^{\omega}$ ⇒ *Pragma Semicolon*$^{\omega}$] **do** ????
end proc;

Attributes

Syntax

Attributes ⇒
  «empty»
| *Attribute* [no line break] *Attributes*

Attribute ⇒
  AttributeExpression
| **abstract**
| **final**
| **private**
| **public**
| **static**
| **true**
| **false**

Validation

proc *Validate*[*Attributes*] (*c*: CONTEXT, *e*: STATICENV)
  [*Attributes* ⇒ «empty»] **do nothing**;
  [*Attributes*$_0$ ⇒ *Attribute* [no line break] *Attributes*$_1$] **do**
    Validate[*Attribute*](*c*, *e*);
    Validate[*Attributes*$_1$](*c*, *e*)
end proc;

PrivateNamespace[*Attribute*]: NAMESPACE;

proc *Validate*[*Attribute*] (*c*: CONTEXT, *e*: STATICENV)
  [*Attribute* ⇒ *AttributeExpression*] **do** *Validate*[*AttributeExpression*](*c*, *e*);
  [*Attribute* ⇒ **abstract**] **do nothing**;
  [*Attribute* ⇒ **final**] **do nothing**;
  [*Attribute* ⇒ **private**] **do**
    p: NAMESPACEOPT ← *c*.privateNamespace;
    if *p* = **null then throw syntaxError end if**;
    PrivateNamespace[*Attribute*] ← *p*;
  [*Attribute* ⇒ **public**] **do nothing**;
  [*Attribute* ⇒ **static**] **do nothing**;
  [*Attribute* ⇒ **true**] **do nothing**;
  [*Attribute* ⇒ **false**] **do nothing**
end proc;

Evaluation

proc *Eval*[*Attributes*] (*e*: ENVIRONMENT): ATTRIBUTE
  [*Attributes* ⇒ «empty»] **do return true**;
  [*Attributes*$_0$ ⇒ *Attribute* [no line break] *Attributes*$_1$] **do**
    a: ATTRIBUTE ← *Eval*[*Attribute*](*e*);
    if *a* = **false then return false end if**;
    b: ATTRIBUTE ← *Eval*[*Attributes*$_1$](*e*);
    return *combineAttributes*(*a*, *b*)
end proc;

```
proc Eval[Attribute] (e: ENVIRONMENT): ATTRIBUTE
    [Attribute ⇒ AttributeExpression] do
        a: OBJECT ← readReference(Eval[AttributeExpression](e));
        if a ∉ ATTRIBUTE then throw typeError end if;
        return a;
    [Attribute ⇒ abstract] do
        return COMPOUNDATTRIBUTE⟨{}, false, null, false, false, abstract, null, false, false⟩;
    [Attribute ⇒ final] do
        return COMPOUNDATTRIBUTE⟨{}, false, null, false, false, final, null, false, false⟩;
    [Attribute ⇒ private] do return PrivateNamespace[Attribute];
    [Attribute ⇒ public] do return publicNamespace;
    [Attribute ⇒ static] do
        return COMPOUNDATTRIBUTE⟨{}, false, null, false, false, static, null, false, false⟩;
    [Attribute ⇒ true] do return true;
    [Attribute ⇒ false] do return false
end proc;
```

Built-in Operators

Unary Operators

```
proc plusObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    return toNumber(a)
end proc;

proc minusObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    return float64Negate(toNumber(a))
end proc;

proc bitwiseNotObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    i: INTEGER ← toInt32(toNumber(a));
    return realToFloat64(bitwiseXor(i, –1))
end proc;

proc incrementObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    x: OBJECT ← unaryPlus(a);
    return binaryDispatch(addTable, x, 1.0)
end proc;

proc decrementObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    x: OBJECT ← unaryPlus(a);
    return binaryDispatch(subtractTable, x, 1.0)
end proc;

proc callObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    case a of
        ~~UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ ATTRIBUTE ∪ PROTOTYPE do~~UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ PROTOTYPE do
            throw typeError;
        CLASS ∪ INSTANCE do return a.call(this, args);
        METHODCLOSURE do return callObject(a.this, a.method.f, args)
    end case
end proc;
```

```
proc constructObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    case a of
        UNDEFINED ∪ NULL ∪ BOOLEAN ∪ FLOAT64 ∪ STRING ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪
                METHODCLOSURE ∪ PROTOTYPE do
            throw typeError;
        CLASS ∪ INSTANCE do return a.construct(this, args)
    end case
end proc;

proc bracketReadObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
    name: STRING ← toString(args.positional[0]);
    return readQualifiedProperty(a, QUALIFIEDNAME⟨publicNamespace, name⟩, true)
end proc;

proc bracketWriteObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    if |args.positional| ≠ 2 or args.named ≠ {} then throw argumentMismatchError end if;
    newValue: OBJECT ← args.positional[0];
    name: STRING ← toString(args.positional[1]);
    writeQualifiedProperty(a, QUALIFIEDNAME⟨publicNamespace, name⟩, true, newValue);
    return undefined
end proc;

proc bracketDeleteObject(this: OBJECT, a: OBJECT, args: ARGUMENTLIST): OBJECT
    if |args.positional| ≠ 1 or args.named ≠ {} then throw argumentMismatchError end if;
    name: STRING ← toString(args.positional[0]);
    return deleteQualifiedProperty(a, name, publicNamespace, true)
end proc;

plusTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, plusObject⟩};

minusTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, minusObject⟩};

bitwiseNotTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, bitwiseNotObject⟩};

incrementTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, incrementObject⟩};

decrementTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, decrementObject⟩};

callTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, callObject⟩};

constructTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, constructObject⟩};

bracketReadTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, bracketReadObject⟩};

bracketWriteTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, bracketWriteObject⟩};

bracketDeleteTable: UNARYMETHOD{} ← {UNARYMETHOD⟨objectClass, bracketDeleteObject⟩};
```

Binary Operators

```
proc addObjects(a: OBJECT, b: OBJECT): OBJECT
    ap: OBJECT ← toPrimitive(a, null);
    bp: OBJECT ← toPrimitive(b, null);
    if ap ∈ STRING or bp ∈ STRING then return toString(ap) ⊕ toString(bp)
    else return float64Add(toNumber(ap), toNumber(bp))
    end if
end proc;
```

```
proc subtractObjects(a: OBJECT, b: OBJECT): OBJECT
    return float64Subtract(toNumber(a), toNumber(b))
end proc;

proc multiplyObjects(a: OBJECT, b: OBJECT): OBJECT
    return float64Multiply(toNumber(a), toNumber(b))
end proc;

proc divideObjects(a: OBJECT, b: OBJECT): OBJECT
    return float64Divide(toNumber(a), toNumber(b))
end proc;

proc remainderObjects(a: OBJECT, b: OBJECT): OBJECT
    return float64Remainder(toNumber(a), toNumber(b))
end proc;

proc lessObjects(a: OBJECT, b: OBJECT): OBJECT
    ap: OBJECT ← toPrimitive(a, null);
    bp: OBJECT ← toPrimitive(b, null);
    if ap ∈ STRING and bp ∈ STRING then return ap < bp
    else return float64Compare(toNumber(ap), toNumber(bp)) = less
    end if
end proc;

proc lessOrEqualObjects(a: OBJECT, b: OBJECT): OBJECT
    ap: OBJECT ← toPrimitive(a, null);
    bp: OBJECT ← toPrimitive(b, null);
    if ap ∈ STRING and bp ∈ STRING then return ap ≤ bp
    else return float64Compare(toNumber(ap), toNumber(bp)) ∈ {less, equal}
    end if
end proc;
```

```
proc equalObjects(a: OBJECT, b: OBJECT): OBJECT
    case a of
        UNDEFINED ∪ NULL do return b ∈ UNDEFINED ∪ NULL;
        BOOLEAN do
            if b ∈ BOOLEAN then return a = b
            else return equalObjects(toNumber(a), b)
            end if;
        FLOAT64 do
            bp: OBJECT ← toPrimitive(b, null);
            case bp of
                UNDEFINED ∪ NULL ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪
                        PROTOTYPE ∪ INSTANCE do
                    return false;
                BOOLEAN ∪ STRING ∪ FLOAT64 do
                    return float64Compare(a, toNumber(bp)) = equal
            end case;
        STRING do
            bp: OBJECT ← toPrimitive(b, null);
            case bp of
                UNDEFINED ∪ NULL ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪
                        PROTOTYPE ∪ INSTANCE do
                    return false;
                BOOLEAN ∪ FLOAT64 do
                    return float64Compare(toNumber(a), toNumber(bp)) = equal;
                STRING do return a = bp
            end case;
        NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do
            case b of
                UNDEFINED ∪ NULL do return false;
                ~~NAMESPACE ∪ ATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do~~NAMESPACE ∪
                        COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪ PROTOTYPE ∪ INSTANCE do
                    return strictEqualObjects(a, b);
                BOOLEAN ∪ FLOAT64 ∪ STRING do
                    ap: OBJECT ← toPrimitive(a, null);
                    case ap of
                        UNDEFINED ∪ NULL ∪ NAMESPACE ∪ COMPOUNDATTRIBUTE ∪ CLASS ∪ METHODCLOSURE ∪
                                PROTOTYPE ∪ INSTANCE do
                            return false;
                        BOOLEAN ∪ FLOAT64 ∪ STRING do return equalObjects(ap, b)
                    end case
            end case
    end case
end proc;

proc strictEqualObjects(a: OBJECT, b: OBJECT): OBJECT
    if a ∈ FLOAT64 and b ∈ FLOAT64 then return float64Compare(a, b) = equal
    else return a = b
    end if
end proc;

proc shiftLeftObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toUInt32(toNumber(a));
    count: INTEGER ← bitwiseAnd(toUInt32(toNumber(b)), 0x1F);
    return realToFloat64(uInt32ToInt32(bitwiseAnd(bitwiseShift(i, count), 0xFFFFFFFF)))
end proc;
```

```
proc shiftRightObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toInt32(toNumber(a));
    count: INTEGER ← bitwiseAnd(toUInt32(toNumber(b)), 0x1F);
    return realToFloat64(bitwiseShift(i, –count))
end proc;

proc shiftRightUnsignedObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toUInt32(toNumber(a));
    count: INTEGER ← bitwiseAnd(toUInt32(toNumber(b)), 0x1F);
    return realToFloat64(bitwiseShift(i, –count))
end proc;

proc bitwiseAndObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toInt32(toNumber(a));
    j: INTEGER ← toInt32(toNumber(b));
    return realToFloat64(bitwiseAnd(i, j))
end proc;

proc bitwiseXorObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toInt32(toNumber(a));
    j: INTEGER ← toInt32(toNumber(b));
    return realToFloat64(bitwiseXor(i, j))
end proc;

proc bitwiseOrObjects(a: OBJECT, b: OBJECT): OBJECT
    i: INTEGER ← toInt32(toNumber(a));
    j: INTEGER ← toInt32(toNumber(b));
    return realToFloat64(bitwiseOr(i, j))
end proc;

addTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, addObjects⟩};

subtractTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, subtractObjects⟩};

multiplyTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, multiplyObjects⟩};

divideTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, divideObjects⟩};

remainderTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, remainderObjects⟩};

lessTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, lessObjects⟩};

lessOrEqualTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, lessOrEqualObjects⟩};

equalTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, equalObjects⟩};

strictEqualTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, strictEqualObjects⟩};

shiftLeftTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, shiftLeftObjects⟩};

shiftRightTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, shiftRightObjects⟩};

shiftRightUnsignedTable: BINARYMETHOD{}
        ← {BINARYMETHOD⟨objectClass, objectClass, shiftRightUnsignedObjects⟩};

bitwiseAndTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, bitwiseAndObjects⟩};

bitwiseXorTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, bitwiseXorObjects⟩};

bitwiseOrTable: BINARYMETHOD{} ← {BINARYMETHOD⟨objectClass, objectClass, bitwiseOrObjects⟩};
```

Built-in Namespaces

Built-in Units

EvalProgram[*Program* ⇒ *Directives*]: OBJECT
begin
~~*Validate*[*Directives*](*initialValidationEnv*);~~*Validate*[*Directives*](*initialContext*, *initialStaticEnv*, **true**);
return *Eval*[*Directives*](*initialDynamicEnv*, **undefined**)
end;

Index

Nonterminals

Tags

Semantic Domains

Globals