The Grace Programming Language
Draft Specification Version 0.7.5

Andrew P. Black Kim B. Bruce James Noble

Contents

1 Introduction

This is a specification of the Grace Programming Language. This specification is notably incomplete, and everything is subject to change. In particular, this version does *not* address:

- static type system
- immutable data and pure methods.
- reflection
- assertions, data-structure invariants, pre- & post-conditions, and contracts
- concurrency
- libraries and dialects, including implementations of Number, and
- testing.

2 User Model

All designers in fact have user and use models consciously or subconsciously in mind as they work. Team design ... requires explicit models and assumptions.

Frederick P. Brooks, *The Design of Design.*

Grace has been designed with the following users in mind.

1. First year university students learning programming in CS1 and CS2 courses that use object-oriented programming.
   - The courses may be structured objects first, or procedures first.
   - The courses may be taught using dynamic types, static types, or both in combination (in either order).
   - Grace offers some (but not necessarily complete) support for "functional first" curricula, primarily for courses that proceed rapidly to procedural and object-oriented programming.
2. University students taking second year classes in programming, algorithms and data structures, concurrent programming, software craft, and software design.
3. Faculty and teaching assistants developing libraries, frameworks, examples, problems and solutions, for the above courses.
4. Programming language researchers needing a contemporary object-oriented programming language as a research vehicle.
5. Designers of other programming or scripting languages in search of a good example of contemporary OO language design.

3 Syntax

Much of the following text assumes the reader has a minimal grasp of computer terminology and a "feeling" for the structure of a program.

Jensen and Wirth, *Pascal: User Manual and Report.*

Grace programs are written in Unicode. Reserved words are written in the ASCII subset of Unicode.

3.1 Character Equivalencies

The following ASCII sequences are treated as equivalent to the corresponding Unicode characters everywhere except in strings.

ASCII	Unicode	Codepoint
>=	≥	U+2265
<=	≤	U+2264
!=	≠	U+2260
->	→	U+2192
]]	⟧	U+27E6
[[	⟦	U+27E7

3.2 Layout

Grace uses braces for grouping. Code layout must be consistent with grouping: indentation must increase by at least two spaces after a brace. Statements are terminated by line breaks when the following line has the same or lesser indentation than the indentation of the line containing the start of the current statement. Statements may optionally be terminated by semicolons.

Example code with punctuation

```
def x =
    mumble "3"
    fratz 7;
while {stream.hasNext} do {
    print(stream.read);
};
```

Example code without punctuation

```
def x =
    mumble "3"
    fratz 7
while {stream.hasNext} do {
    print(stream.read)
}
```

This defines x to be the result of the single request mumble ("3") fratz (7).

3.3 Comments

Comments start with a pair of slashes // and are terminated by the end of the line. Comments are *not* treated as white-space. Each comment is conceptually attached to the smallest immediately preceding syntactic unit, except that comments following a blank line are attached to the largest immediately following syntactic unit.

Example

```
// comment, to end of line
```

3.4 Identifiers and Operators

Identifiers must begin with a letter, which is followed by a sequence of zero or more letters, digits, prime (') or underscore (_) characters. Conventionally, type and pattern identifiers start with capital letters, while other identifiers start with lower-case letters.

A identifier comprising a single underscore _ acts as a placeholder: it can appear in declarations, but not in expressions. In declarations, _ is treated as a fresh identifier.

Operators are sequences of unicode mathematical operator symbols and the following ASCII operator characters:

! ? @ # % ^ & | ~ = + − ∗ / \ > < : . $

3.5 Reserved Tokens

Grace has the following reserved tokens:

alias as class def dialect exclude import inherit is method object outer prefix required return self Self trait type use var where

. ... := = ; { } [ ] ( ) : –> < >

3.6 Newlines, Tabs and Control Characters

Newline in Grace programs can be represented by the Unicode line feed (LF) character, by the Unicode carriage return (CR) character, or by the Unicode line separator (U+2028) character; a line feed that immediately follows a carriage return is ignored.

Tabs and all other non-printing control characters are syntax errors, even in a string literal. Escape sequences are provided to denote control characters in strings; see the Table of StringEscapes.

4 Built-in Objects

4.1 Done

Assignments, and methods without an explicit result, have the value done, of type Done. The type Done plays a role similar to *void* or *Unit* in other languages. The only requests understood by done are asString and asDebugString; in particular, done does not have an equality method.

4.2 Elipsis

The token ... is a valid expression, but evaluating it will lead to a runtime error. It is included in the language so that programmers can indicate that their code is incomplete.

4.3 Numbers

In Grace, numbers are objects. Grace supports a single type **Number**, which accommodates at least 64-bit precision floats. Implementations may support other classes of numbers, and may define types that extend Number; a full specification of numeric types is yet to be completed.

Grace has three forms of numerals (that is, literals that denote Number objects).

1. Decimal numerals, written as strings of digits.
2. Base-exponent numerals, always in decimal, which contain a decimal point, or an exponent, or both. Grace uses e as the exponent indicator. Base-exponent numerals may have a minus in front of the exponent.
3. Explicit radix numerals, written as a (decimal) number between 2 and 35 representing the radix, a leading x, and a string of digits, where the digits from 10 to 35 are represented by the letters A to Z, in either upper or lower case. A radix of 0 is taken to mean a radix of 16.

Examples

```
1
42
3.14159265
13.343e−12
414.45e3
16xF00F00
2x10110100
0xdeadbeef // Radix zero treated as 16
```

4.4 Booleans

The predefined constants **true** and **false** denote values of Grace's Boolean type. Boolean operators are written using && for and, || for or, and prefix ! for not.

Examples

```
p && q
toBe || toBe.not
```

In addition to && and || taking boolean arguments, they also accept parameterless blocks that return Boolean. This gives them "short circuit" (a.k.a. "non-commutative") semantics.

Examples

```
p && { q }
toBe || { ! toBe }
```

4.5 Strings

4.5.1 String Literals

String literals in Grace are written between double quotes, and must be confined to a single line. Strings literals support a range of escape characters such as "\n\t", and also escapes for Unicode; these are listed in the table below.

Individual characters are represented by Strings of length 1. Strings are immutable, so an implementation may intern them. Grace's standard library supports efficient incremental string construction.

Escape	Meaning	Unicode
\\	backslash	U+005C
\n	line-feed	U+000A
\t	tab	U+0009
\{	left brace	U+007B
\}	right brace	U+007D
\"	double quote	U+0022
\r	carriage return	U+000D
\l	line separator	U+2028
_	non-breaking space	U+00A0
\uhhhh	4-digit Unicode	U+hhhh
\Uhhhhhh	6-digit Unicode	U+hhhhhh

Examples

```
"Hello World!"
"\t"
```

```
"The End of the Line\n"
"A"
```

4.5.2 String Constructors

String Constructors are a generalization of String Literals that contain expressions enclosed in braces. The value of a String Constructor is obtained by first evaluating any expressions inside braces, requesting asString of the resulting object, and inserting the resulting string into the string literal in place of the brace expression.

Example

```
"Adding {a} to {b} gives {a+b}"
```

4.6 Lineups

A Lineup is a comma separated list of expressions surrounded by [and].

Examples

```
[ ]       //empty lineup
[ 1 ]
[ red, green, blue ]
```

When executed, a lineup returns an object that supports the Iterator interface, which includes the methods size, map, do(_), and iterator. Lineups are most frequently used to build collections, to control loops, and to pass collections of options to methods.

Examples

```
set [ 1, 2, 4, 5 ]          //make a set
sequence [ "a", "b", "c" ]   //make a sequence
["a", "e", "i", "o", "u"].do { x -> testletter(x) }
myWindow.addWidgets [
    title "Launch",
    text "Good Morning, Mrs President",
    button "OK" action { missiles.launch },
    button "Cancel" action { missiles.abort }
]
```

4.7 Blocks

Grace blocks are lambda expressions, with or without parameters. If a parameter list is present, the parameters are separated by commas and the list is separated from the body of the block by the -> symbol. Within the body of the block, the parameters cannot be assiged.

```
{ do.something }
{ i -> i + 1 }
{ i:Number -> i + 1 }
{ sum, next -> sum + next }
```

Blocks are lexically scoped, and can close over any visible field or parameter. The body of a block consists of a sequence of declarations and expressions; declarations are local to the block. An empty body is allowed, and is equivalent to done. A **return** statement inside a block returns from the enclosing method.

Blocks construct objects containing a method named apply, or apply(n), or apply(n, m), . . . , where the number of parameters to apply is the same as the number of parameters of the block. Requesting the apply(...) method evaluates the block; it is an error to provide the wrong number of arguments. If block parameters are declared with type annotations, it is a TypeError if the arguments do not conform to those types.

Examples

The looping construct

```
for (1..10) do {
    i -> print i
}
```

might be implemented as a method with a block parameter

```
method for (collection) do (block) {
    def stream = collection.iterator
    while {stream.hasNext} do {
        block.apply(stream.next)
    }
}
```

Here is another example:

```
var sum := 0
def summingBlock: Block1[[Number, Number]] =
    { i: Number ->  sum := sum + i }
summingBlock.apply(4)       // sum now 4
summingBlock.apply(32)      // sum now 36
```

5 Declarations

Declarations may occur anywhere within a module, object, class, or trait. Field declarations may also occur within a method or block body. Declarations are visible within the *whole* of their containing lexical scope. It is an error to declare any name more than once in a given lexical scope.

Grace has a single namespace for all identifiers: methods, parameters, constants, variables, classes, traits, and types. It is a *shadowing error* to declare a parameter

(but not a method or field) that has the same name as a lexically-enclosing field, method, or parameter.

5.1 Fields

Grace has two kinds of fields: constants and variables.

5.1.1 Constants

Constants are defined with the **def** keyword; they bind an identifier to the value of an initialising expression, and may optionally be given a type. This type is checked when the constant is initialised. Constants cannot be re-bound.

Examples

```
def x = 3 * 100 * 0.01
def x:Number = 3
def x:Number          // Syntax Error: x must be initialised
```

5.1.2 Variables

Variable are introduced with the **var** keyword. Variables can be re-bound to new values as often as desired, using an assignment. A variable declaration may optionally provide an initial value; if there is no initial value, the variable is *uninitialised* until it is assigned. Any attempt to access the value of an uninitialised variable is an error, which may be caught either at run time or at compile time. Variables may be optionally given a type: this type is checked when the variable is initialised and assigned.

Examples

```
var x:Rational := 3     // explicit type
var x:Rational          // x must be initialised before access
var x := 3              // x has type Unknown
var x                   // x has type Unknown, value is uninitialised
```

5.2 Methods

Methods are declared using the **method** keyword followed by a name. Methods define the action to be taken when the object containing the method receives a request with the given name. Because every method must be associated with an object, methods may not be declared directly inside other methods. The type of the object returned from the method may optionally be given after the symbol –>: this type is checked when the method returns. The body of the method is enclosed in braces.

```
method pi  { 3.141592634 }

method greet(user: Person) from(sender: Person) {
    print "{sender} sends his greetings, {user}."
}

method either (a) or (b) -> Done {
    if (random.nextBoolean)
        then {a.apply} else {b.apply}
}
```

5.2.1 Method Names

To improve readability, method names have several forms. For each form, we describe its appearance, and also a *canonical* form of the name which is used in dispatching method requests. A request "matches" a method if the canonical names are equal.

1. A method can be named by a single identifier, in which case the method has no parameters; in this case the *canonical* name of the method is the identifier.
2. A method can be named by a single identifier suffixed with :=; this form of name is conventionally used for writer methods, both user-written and automatically-generated, as exemplified by value:= below. Such methods *always* take a single parameter after the :=
3. A method can be named by one or more *parts*, where each *part* is an identifier followed by a parenthesized list of parameters. In this case the *canonical* name of the method is a sequence of parts, where each part comprises the identifier for that part followed by (_, ..., _), the number of underscores between the parentheses being the number of parameters of the part.
4. A method can be named by a sequence of operator symbols. Such an "operator method" can have no parameters, in which case the method is requested by a prefix operator expression. It can also have one parameter, in which case it is requested by a binary operator expression. The canonical name of a unary method is **prefix** followed by the operator symbols; the canonical name of a binary method is the sequence of operator symbols followed by (_)

Examples of single identifiers

```
method ping { print "PING!" }
method isEmpty { elements.size == 0 }
```

Examples of assignment methods

```
method value:= (n: Number) -> Done {
    print "value currently {value}, now assigned {n}"
    outer.value:= n
}
```

Examples of multi-part names

```
method drawLineFromOriginTo (destination)
method drawLineFrom (source) to (destination)
method max(v1, v2)
```

In the first two examples, the canonical names of the methods are drawLineFromOriginTo(_), and drawLineFrom(_)to(_). The latter comprises two parts: drawLineFrom(_) and to(_). In the third example, the canonical name of the method is max(_,_).

Examples of operator symbols

```
method + (other : Point) -> Point {
    (x + other.x) @ (y + other.y)
}

method prefix- -> Point
    { 0 - self }
```

As a consequence of the above rules, methods max(a, b, c) and max(a, b) have different canonical names and are therefore treated as distinct methods. In other words, Grace allows "overloading by arity". (Grace does *not* allow overloading by type).

5.2.2 Method parameters

Depending on their syntatic form, method declarations may include one or more paramter lists. Inside method bodies, method paramters are treated as **'def'**s: they may not be reassigned. Method parameters may optionally be annotated with types: the corresponding arguments will be checked against those types, either before execution, or just before the method body is executed.

5.2.3 Type Parameters

Methods may be declared with one or more type parameters, which are listed between **[[** and **]]** used as brackets. If present, type parameters must appear after the identifier of the first part of a multipart name.

If an operator method has a type parameter list, it must be separated from the sequence of operator symbols that names the method by a space.

The presence or absence of type parameters does not change the canonical name of the method.

Examples

```
method sumSq[[T]](a : T, b : T) -> T where T <: Numeric {
    (a * a) + (b * b)
}

method prefix- [[T]] -> Number
    { 0 - self }
```

5.2.4 Returning a Value from a Method

Methods may contain one or more **return** statements. If a **return** e statement is executed, the method terminates with the value of the expression e, while a **return** statement with no expression is equivalent to **return** done. If execution reaches the end of the method body without executing a **return**, the method terminates and returns the value of the last expression evaluated. An empty method body returns done.

5.3 Annotations

Any declaration, and any object constructor, may have a comma-separated list of annotations following the keyword **is** before its body or initialiser. Grace defines the following core annotations:

Annotation	Semantics
confidential	method may be requested only on self or outer — see Encapsulation

Annotation	Semantics
manifest	method must return a manifest object - Manifest Expressions
overrides	method must override another method - Overriding Methods
public	method may be requested from anywhere

Annotation	Semantics
	field can be read and written from any object - see Encapsulation
readable	field may be read from anywhere - see Encapsulation

Annotation	Semantics
writeable	variable may be assigned from anywhere - see Encapsulation

Additional annotations may be defined by dialect or libraries.

Examples

```
var x is readable, writeable := 3
def y: Number is public
method foo is confidential { }
method id[[T]] is required { }
```

5.4 Encapsulation

Grace has different default encapsulation rules for methods, types, and fields. The defaults can be changed by explicit annotations. The details are as follows.

5.4.1 Public

Public attributes can be requested by any client that has access to the object that defines them.

5.4.2 Confidential

Confidential attributes can be requested only on **self** or on some number of cascaded **outer**s, or in an implicit request (which must resolve to one of the former cases). Consequently, if m is defined in the object, class, or trait d, it is

accessible to *d*, to objects that inherit from *d*, and to objects lexically enclosed by *d*, but not to clients of *d*.

5.4.3 Methods, Classes, Traits and Types

By default, methods (which category includes classes and traits), and types, are public. If a method or type is annotated `is confidential`, it is confidential.

5.4.4 Fields

Variables and definitions (`var` and `def` declarations) immediately inside an object constructor create *fields* in that object.

A field declared as `var x` can be read using the request `x` and assigned to using the assignment request `x:=(...)`. A field declared as `def y` can be read using the request `y`, and cannot be assigned. By default, fields are *confidential*.

The default visibility can be changed using annotations. The annotation `readable` can be applied to a `def` or `var` declaration, and makes the accessor request available to any object. The annotation `writable` can be applied to a `var` declaration, and makes the assignment request available to any object. It is also possible to annotate a field declaration as `public`. In the case of a `def`, `public` is equivalent to (and preferred over) `readable`. In the case of a `var`, `public` is equivalent to `readable, writable`.

Fields and methods share the same namespace. The syntax for variable access is identical to that for requesting a reader method, while the syntax for variable assignment is identical to that for requesting an assignment method. This means that an object cannot have a field and a method with the same name, and cannot have a method `x:=(_)` as well as a `var` field named `x`.

Examples

```
object {
    def a = 1                  // Confidential access to a
    def b is public = 2        // Public access to b
    def c is readable = 2      // Public access to c
    var d := 3                 // Confidential access and assignment
    var e is readable          // Public access and confidential assignment
    var f is writable          // Confidential access, public assignment
    var g is public            // Public access and assignment
    var h is readable, writable    // Public access and assignment
}
```

5.4.5 No Private Attributes

Some other languages support "private attributes", which are available only to an object itself, and not to clients or inheritors. Grace does not have private

fields or methods; all can be accessed from subobjects. However, identifiers from outer scopes can be used to obtain an effect similar to privacy.

Examples

```
method newShipStartingAt(s:Point)endingAt(e:Point) {
    // returns a battleship object extending from s to e.  This object cannot
    // be asked its size, or its location, or how much floatation remains.
    assert ( (s.x == e.x) || (s.y == e.y) )
    def size = s.distanceTo(e)
    var floatation := size
    object {
        method isHitAt(shot:Vector2D) {
            if (shot.onLineFrom(s)to(e)) then {
                floatation := floatation −1
                if (floatation == 0) then { self.sink }
                true
            } else { false }
        }
        ...
    }
}
```

The object returned by newShipStartingAt()endingAt() can update the variable floatation in the surrounding scope, even though it is not accessible to anything inheriting from that object.

6 Objects, Classes, and Traits

Grace **object** constructors generate individual objects. Grace **class** declarations define methods that generate objects, all of which have the same structure.

The design of Grace's reuse mechanism is complete, but tentative. We need experience before confirming the design.

6.1 Objects

Object constructors are expressions that evaluate to an object with the given attributes. Each time an object constructor is executed, a new object is created. In addition to declarations of fields and methods, object constructors can also contain expressions (executable code at the top level), which are executed as a side-effect of evaluating the object constructor. All of the declared attributes of the object are in scope throughout the object constructor.

Examples

```
object {
    def colour:Colour = Colour.tabby
```

```
    def name:String = "Unnamed"
    var miceEaten := 0
    method eatMouse { miceEaten := miceEaten + 1 }
}
```

Like everything in Grace, object constructors are lexically scoped.

A name can be bound to an object constructor, like this:

```
def unnamedCat =  object {
    def colour : Colour = Colour.tabby
    def name : String = "Unnamed"
    var miceEaten := 0
    method eatMouse { miceEaten := miceEaten + 1 }
}
```

6.2 Class Declarations

A class is a method whose body is treated as an object constructor that is executed every time the class is invoked. The class returns the newly-created object. For example,

```
class catColoured(c) named (n) {
    def colour is public = c
    def name is public = n
    var miceEaten is readable := 0
    method eatMouse {miceEaten := miceEaten + 1}
    print "The cat {n} has been created."
}
```

is equivalent to

```
method catColoured(c) named (n) {
    object {
        inherit graceObject
        def colour is public = c
        def name is public = n
        var miceEaten is readable := 0
        method eatMouse {miceEaten := miceEaten + 1}
        print "The cat {n} has been created."
    }
}
```

This class might be used as follows:

```
def fergus = catColoured (colour.tortoiseshell) named "Fergus"
```

This creates an object with fields `colour` (set to `colour.tortoiseshell`), `name` (set to `"Fergus"`), and `miceEaten` (initialised to `0`), prints "The cat Fergus has been created", and binds the name `fergus` to this object.

6.3 Trait Objects and Trait Declarations

Trait objects are objects with certain properties. Specifically, a trait object is created by an object constructor that contains no field declarations and no executable code, that **use**s only other traits, and that **inherit**s nothing.

Aside from these restrictions, Grace's **trait** syntax and semantics is parallel to the class syntax. In particular, a **trait** defines a method that returns a trait object.

```
trait emptiness {
    method isEmpty { size == 0 }
    method nonEmpty { size != 0 }
    method ifEmptyDo (eAction) nonEmptyDo (nAction) {
            if (isEmpty) then { eAction.apply } else { do(nAction) }
    }
}
```

6.4 Type Parameters

Like methods, classes and traits may be declared to have type parameters. Requests on the class or trait may optionally be provided with type arguments.

Type parameters may be constrained with **where** clauses. The details have yet to be specified.

Example

```
class vectorOfSize(size)[[T]] {
    var contents := Array.size(size)
    method at(index : Number) -> T {return contents.at() }
    method at(index : Number) put(elem : T) { }
}

class sortedVectorOfSize(size)[[T]]
    where T <: Comparable[[T]] {
      ...
}
```

6.5 Reuse

Grace supports reuse in two ways: through **inherit** statements and through **use** statements. Object constructors (and classes) can contain one **inherit** statement, while traits cannot contain an **inherit** statement; object constructors, classes and traits can all contain one or more **use** statements.

Both **inherit** and **use** introduce the attributes of a reused object — called the *parent* — into the current object (the object under construction). There are two differences between **inherit** and **use** clauses:

1. the object reused by a **use** clause must be a trait object; and
2. **inherit** clauses include the methods from graceObject, while **use** clauses do not.

An **inherit** or **use** clause contains a Manifest Expression that creates a new object, such as a request on a class or trait. This means that the request cannot depend on a **self**, implicitly or explicitly. This means that programs cannot inherit or use any trait or class that can potentially be overridden. Note that the arguments to Manifest Expression need not themselves be manifest.

If it is necessary for the current object to access an overridden attribute of a parent, the overridden attribute can be given an additional name by attaching an **alias** clause to the inherit or use statement: **alias** n(_) = m(_) creates a new confidential *alias* n(_) for the attribute m(_). It is a *object composition error* to alias an attribute to its own name. Attributes of the parent that are not wanted can be excluded using an **exclude** clause: excluded attributes are replaced by a confidential, **required** method. It is an *object composition error* to alias or exclude attributes that are not present in the class or trait being inherited.

6.5.1 Object Combination and Initialisation

When executed, an object constructor (or trait or class declaration) first creates a new object with no attributes, and binds it to **self**.

Second, the attributes of the superobject (created by the **inherit** clause, possibly modified by **alias** and **exclude**) are installed in the new object.

Third, the methods of all traits (created by **use** clauses, possibly modified by **alias** and **exclude**, and excluding those methods inherited unchanged from graceObject) are combined. It is an *object composition error* for there to be multiple definitions of a method. This combination of methods is then installed in the new object: methods in the trait combination override declarations in the superobject.

Fourth, attributes create by local declarations are installed in the new object: local declarations override declarations from both superobject and traits, except that it is an *object composition error* for an alias to be overridden by a local declaration.

Fifth, types are evaluated and bound to their declarations. Types cannot depend on runtime values; if they depend on the type of a constant (because the constant is treated as a Singleton type, then that constant, if overridden in a subclass, can be overridden only by a new object with the same type.

Finally, field initializers and executable statements are executed, starting with the most superior inherited superobject, and finishing with the initializers of local fields, and local statements. (Note that *used* objects must be traits, and therefore contain no executable code.) Initialisers for all **def**s and **var**s, and code in the bodies of parents, are executed once in the order they are written, even

for **def**s or **var**s that are excluded from the new object, or aliased to one or more new names. During initialisation, **self** is bound to the new object being created, even while executing code and initialisers of parents.

As a consequence of these rules, a new object can change the initialization of its parents, by overriding the method requested on self by the parents' initialisers.

6.5.2 Required Methods

Methods may be declared to be *required* by giving them the body **required** . (Required methods are similar to what some other languages call abstract methods.) This indicates that a real method body must be supplied before that method can be requested. Required methods do not conflict with other methods; a required local method overrides an inherited method in the normal way. Requesting a required method will generate a runtime error.

6.5.3 Overriding Methods

A new declaration in the current object overrides a declaration from a parent. Methods may be annotated with override. A method so annotated must override a method from its parent with the same name and arity. The override annotation is optional: local methods override parents' methods with or without the override annotation. Dialects may require the annotation.

Examples

The example below shows how a class can use a method to override an accessor method for an inherited variable.

```
class pedigreeCatColoured (aColour) named (aName) {
    inherit catColoured (aColour) named (aName)
    var prizes := 0
    method miceEaten is override { 0 }
          // a pedigree cat would never be so coarse
    method miceEaten:= (n:Number) -> Number is override { return }
          // ignore attempts to debase it
}
```

Traits are designed to be used as fine-grained components of reuse:

```
trait feline {
    method independent { "I did it my way" }
    method move {
        if (isOut) then {
            comeIn
        } else {
            goOut
        }
    }
```

```
}

trait canine {
    method loyal { "I'm your best friend" }
    method move {
        if (you.location != self.location) then {
            self.position := you.heel
        }
    }
}
```

In this context, the following object has a trait conflict:

```
object {
    use feline alias catMove = move
    use canine alias dogMove = move
}
```

because the move attribute is defined in two separate traits. In contrast, the following definition is legal:

```
def nyssa = object {
    use feline alias catMove = move
    use canine alias dogMove = move
    method move {
        if (random.choice) then {
            catMove
        } else {
            dogMove
        }
    }
}
```

Here, the conflict is resolved by overriding with a local move method. This method accesses the overridden methods from the parent traits using the aliases catMove and dogMove; as a result, nyssa will move either like a dog or a cat, depending on a random variable.

6.5.4 Default Methods

All objects implement a number of *default methods* by inheriting from the graceObject trait. Programmers can of course override some of these implementations, or write alternative implementations of these methods *ab initio*. The type Object defines a type containing all the public default methods:

Method	Purpose
isMe (other: Object) –> Boolean	a *confidential* method that returns true if other is the same object as self
≠ (other: Object) –> Boolean	the inverse of ==

Method	Purpose
asString –> String	a string describing self
asDebugString –> String	a string describing the internals of self
:: (other:Object) –> Binding	a Binding object with self as key and other as value

Notice that graceObject implements ≠ but not ==. This is to help ensure that, when an object chooses to implement ==, ≠ is also available, and is the inverse of ==.

7 Method Requests

Grace is a pure object-oriented language. All computation proceeds by *requesting* an object — the target of the request — to execute a method with a particular *name*. The response of the target is to execute the method, and to answer the return value of the method.

Grace distinguishes the act of *requesting* a method (what Smalltalk calls "sending a message"), and *executing* that method. Requesting a method happens outside the target object, and involves only a reference to the target, the method name, and possibly some arguments. In contrast, executing the method involves the code of the method, which is internal to the target.

7.1 Self

The reserved word **self** refers to the current object. Inside a method, self always refers to the target of the method-request that caused the method to execute. Elsewhere, **self** refers to the object being constructed by the lexically-innermost module, object constructor, class or trait surrounding the word **self** Hence, the expression **self**.x requests x on the current object.

The reserved word **Self** refers to the type of the current object.

Examples

```
self
self.value
self.bar(1,2,6)
self.doThis(3) timesTo("foo")
self + 1
! self
```

7.2 Outer

The reserved word **outer** refers to the object lexically enclosing the current object. The expression **outer**.x requests x on the object lexically enclosing **self**.

Examples

```
outer
outer.outer.outer.outer
outer.value
outer.bar(1,2,6)
outer.outer.doThis(3) timesTo("foo")
outer + 1
! outer
```

7.3 Named Requests

A named method request comprises a *receiver*, followed by a dot **.**, followed by a method name, wherein the parameters have been replaced by expressions that evaluate to the method's arguments. Note that a request without arguments does not contain any parentheses.

The *reciever* is an expression, which when evaluated designates the *target* of the request. The name of a method, which determines the position of the argument lists within that name, is chosen when the method is declared (See Methods). When reading a request of a multi-part method name, you should continue accumulating words and argument lists as far to the right as possible.

Examples

```
self.clear
self.drawLineFrom (p1) to (p2)
self.drawLineFrom (origin) length (9) angle (pi/6)
self.movePenTo (x, y)
self.movePenTo (p)
```

7.3.1 Delimited Arguments

Parenthesis may be omitted where they would enclose a single argument that is a numeral, string, lineup, or block.

Examples

```
self.drawLineFrom (p1) to (p2)
self.drawLineFrom (origin) length 9 angle (pi/6)
print "Hello World"
while {x < 10} then {
    print [a, x, b]
    x := x + 1
}
```

7.3.2 Implicit Requests

If the receiver of a named method request using the name m is **self** or **outer** it may be left implicit, *i.e.*, the **self** or **outer** and the dot may both be omitted. Implicit requests are interpreted as a **self** request, or as an **outer** request, or as an **outer**.**outer**. ... request with the appropriate number of **outer**s.

When resolving an implicit request, the usual rules of lexical scoping apply, so a definition of m in the current scope will take precedence over any definitions in enclosing scopes. However, if m is defined in the current scope by inheritance or trait use, rather than directly, and *also* defined directly in an enclosing scope, then an implicit request of m is ambiguous and is an error.

Implicit requests are always resolved lexically, that is, within the scope in which they are written, and not within the scope of object (class, or trait) that may inherit the method containing them.

Examples of Implicit Requests

```
print "Hello world"
size
canvas
```

Example of Implicit Request Resolution

```
method foo { print "outer" }

class app {
  method barf { foo }
}

class bar {
  inherit app
  method foo { print "bar" }
}

class baz {
  inherit bar
  method barf { foo }   // ambiguous — could be self.foo or outer.foo
}


app.barf  // prints "outer"
bar.barf  // prints "outer"
```

7.4 Assignment Requests

An assignment request is a variable followed by :=, or a request of a method whose name ends with :=. In both cases the := is followed by a single argument,

which need not be surrounded by parentheses. Spaces are optional before and after the :=.

Examples

```
x := 3
y:=2
widget.active := true
```

Assignment methods conventionally return done.

7.5 Binary Operator Requests

Binary operators are methods whose names comprise one or more operator characters, provided that the operator is not reserved by the Grace language. Binary operators have a receiver and one argument; the receiver must be explicit. So, for example, +, ++ and .. are valid operator symbols, but . is not, because it is reserved.

Most Grace operators have the same precedence: it is a syntax error for two distinct operator symbols to appear in an expression without parenthesis to indicate order of evaluation. The same operator symbol can be requested more than once without parenthesis; such expressions are evaluated left-to-right.

Four binary operators do have precedence defined between them: / and * bind more tightly than + and −.

Examples

```
1 + 2 + 3              // evaluates to 6
1 + (2 * 3)            // evaluates to 7
(1 + 2) * 3            // evaluates to 9
1 + 2 * 3              // evaluates to 7
1 +*+ 4 −*− 4            // precedence error
```

Examples

Named method requests without arguments bind more tightly than operator requests.

Grace	Parsed as
1 + 2.i	1 + (2.i)
(a * a) + (b * b).sqrt	(a * a) + ((b *b).sqrt)
((a * a) + (b * b)).sqrt	((a * a) + (b *b)).sqrt
a * a + b * b	(a * a) + (b *b)
a + b + c	(a + b) + c
a − b − c	(a − b) − c

7.6 Unary Prefix Operator Requests

Grace supports unary methods named by operator symbols that precede the explicit receiver. (Since binary operator methods must have an explicit receiver, there is no syntactic ambiguity.)

Prefix operators bind less tightly than named method requests, and more tightly than binary operator requests.

Examples

```
−3 + 4
(−b).squared
−(b.squared)
− b.squared      // parses as −(b.squared)

status.ok := !engine.isOnFire && wings.areAttached && isOnCourse
```

7.7 Precedence of Method Requests

Grace programs are formally defined by the language's Grammar. The grammar gives the following precedence levels; lower numbers bind more tightly.

1. Numerals and constructors for strings, objects, iterables, blocks, and types; parenthesized expressions.
2. Requests of named methods. Multi-part requests accumulate name-parts and arguments as far to the right as possible.
3. Prefix operators
4. "Multiplicative" operators ∗ and /: associate left to right.
5. "Additive" operators + and −: associate left to right.
6. "Other" operators, whose binding must be given explicitly using parenthesis.
7. Assignments and method requests that use := as a suffix to a method name.

7.8 Requesting Methods with Type Parameters

Methods that have type parameters may be requested with or without explicit type arguments. If type arguments are supplied there must be the same number of arguments as there are parameters. If type arguments are omitted, they are assumed to be type Unknown.

Examples

```
sumSq[[Number]](10.i64, 20.i64)

sumSq(10.i64, 20.i64)
```

7.9 Manifest Expressions

The parents in **inherit** <parent> and **use** <parent> clauses must be *manifest*. This means that Grace must be able to determine the *shape* of the object that is being inherited on a module-by-module basis. In particular,

1. the meaning of the parent expressions must not be subject to overriding, and
2. the result of the parent expression must be a fresh object whose shape is statically determinable.

8 Pattern Matching

Pattern matching is based on Pattern objects that respond to the match(subject) request by returning a MatchResult, which is either **false** if the match fails, or a SuccessfulMatch[[R]] object which behaves like **true** but also supports a result request. All type objects are Patterns; in addition, libraries supply non-type Patterns, and programmers are free to implement their own Patterns.

Example

Suppose that the type Point is defined by:

```
type Point = {
  x -> Number
  y -> Number
}
```

and implemented by this class:

```
class x(x':Number) y(y':Number) -> Point {
  method x { x' }
  method y { y' }
}
```

we can write

```
def cp = x(10) y(20)

Point.match(cp)          // SuccessfulMatch, behaves like true
Point.match(cp).result   // cp
Point.match(42)          // false
```

8.1 Matching Blocks

Blocks with a single parameter are called *matching blocks*. Matching blocks also conform to type Pattern, and can be evaluated by requesting match(_) as well as apply(_). When apply(_) would raise a type error because the block's argument

would not conform to its parameter type, match(_) returns false; when apply(_) would return a result r, match(_) returns a SuccessfulMatch object whose result is r.

If the parameter declaration of a matching block takes the form _:pattern, then the _: can be omitted, provided that pattern is is parenthesized, or is a string literal or a numeral. This rule (the *delimited argument rule*) means that the pattern can't be mistaken for a declaration of a parameter to the block.

8.2 Self-Matching Objects

The objects created by String Literals and Numerals are patterns that match strings and numbers that are equal to the literal.

Examples

Matching blocks and self-matching objects can be conveniently used in the match(_)case(_)... family of methods.

```
method fib(n : Number) -> Number {
    match (n)
        case { 0 -> 0 }
        case { 1 -> 1 }
        case { _ -> fib(n-1) + fib(n-2) }
}
```

The first two blocks use self-matching objects; the first is short for { _:0 -> 0 }.

The last block has no pattern (or, if you prefer, has the pattern Unknown, which matches any object). Such a block always matches.

If match(_)case(_)... does not find a match, it raises a non-exhaustive match exception.

```
{ 0 -> "Zero" }
    // match against a Sting Literal

{ s:String -> print(s) }
    // type match, binding s – identical to block with typed parameter

{ (pi) -> print("Pi = " ++ pi) }
    // match against the value of an expression – requires parenthesis

{ a -> print("did not match") }
    // match against empty type annotation; matches anything, and binds to `a`
```

9 Exceptions

Grace supports exceptions, which can be raised and caught. Exceptions are categorized into a hierarchy of ExceptionKinds. At the site where an exceptional

situation is detected, an exception is raised by requesting the raise method on an ExceptionKind object, with a string argument explaining the problem.

Raising an exception does two things: it creates an exception object of the specified kind, and terminates the execution of the expression containing the raise request; it is not possible to restart or resume that execution, although reflection (and thus debuggers) should have access to the stack at the point the exception is thrown. Execution continues when the exception is *caught.*

Examples

```
BoundsError.raise "index {ix} not in range 1..{n}"
UserException.raise "impossible happened"
```

9.1 Catching Exceptions

An exception in expression can be caught by a dynamically-enclosing

```
try(expression)
    catch (block 1)
    ...
    catch (block n)
    finally (finalBlock)
```

in which the block i are pattern-matching blocks. More precisely, if an exception is raised during the evaluation of the try block expression, the catch blocks block 1, block 2, . . . , block n, are attempted, in order, until one of them matches the exception. If none of them matches, then the process of matching the exception continues in the dynamically-surrounding try(_) catch(_) ... catch(_) finally(_). The finalBlock is always executed before control leaves the try(_) catch (_) ... catch(_) finally(_) construct, whether or not an exception is raised, and whether or not one of the catch blocks returns.

Finally clauses can return early, either by executing a **return**, or by raising an exception. In such a situation, any prior **return** or raised exception is silently dropped.

Example

```
try {
    def f = io.open("data.store", "r")
} catch {
    e: NoSuchFile -> print "No Such File"
} catch {
    e: PermissionError -> print "Permission denied"
} catch {
    _: Exception -> print "Unidentified Error"
    system.exit
} finally {
    f.close
}
```

10 Types

Grace uses structural typing @Modula3 @malayeri08 @whiteoak08. Types primarily describe the requests that objects can answer. Fields do not directly influence types, except that a field that is public, readable or writable is treated as the appropriate method or methods.

Type names introduced by type declarations are treated as expressions that denote *type objects*. All type objects are also patterns, so they can be used in pattern matching, typically to perform dynamic type tests. Because type declarations cannot be changed by overriding, the value of a type expression can always be determined before the program is executed; this means that types can be checked statically.

10.1 Predeclared Types

A number of types are declared in the standard prelude and included in most dialects, including None, Done, Boolean, Object, Number, String, Block0, Block1, Block2, Fun, Iterator, Pattern, Exception, and ExceptionKind. Some particular types are treated specially:

10.1.1 Type None

Type None is completely empty; it has no methods.

10.1.2 Type Object

The type Object includes methods to which most objects respond — the Default Methods declared in graceObject. Some objects, notably done, do not conform to Object.

```
type Object = {
    != (other: Object) -> Boolean        // the inverse of ==
    asString -> String                 // a string for use by the client
    asDebugString -> String              // a string for use by the implementor
    :: (other:Object) -> Binding         // a binding with self as the key
}
```

Notice that isMe, although present in graceObject, is not present in type Object, because it is *confidential*. Also notice that neither graceObject nor type Object include ==.

10.1.3 Type Self

The type **Self** represents the public interface of the current object. Self is prohibited as the annotation on parameters, but can be used to annotate results.

10.1.4 Type Unknown

Unknown is not actually a type, although it is treated as a type by the type checker. It is similar to the type label "Dynamic" in C#. Unknown can be written explicitly as a type annotation; moreover, if a declaration is not annotated, then the type of the declared name is *implicitly* Unknown. In addition, omitted type arguments are replaced by Unknown.

Type-checking against Unknown will always succeed: any object matches type Unknown, and type Unknown conforms to all other types.

Examples

```
var x: Unknown := 5   //who knows what the type is?
var x := 5            //same here, but Unknown is implicit
x := "five"           //who cares
x.gilad               //almost certainly raises NoSuchMethod

method id(x) { x }    //argument and return types both implicitly unknown
method id(x: Unknown) -> Unknown { x }  // same thing, explicitly
```

10.2 Interface Types

Types define the interface of objects by detailing their public methods, and the types of the parameters and results of those methods. Types can also contain definitions of other types to describe types nested inside objects.

The various Cat object and class descriptions (see Objects, Classes, and Traits) would create objects that conform to an interface type such as the following. Notice that the public methods implicitly inherited from Object are implicitly included in all types.

```
interface {
    colour -> Colour
    name -> String
    miceEaten -> Number
    miceEaten:= (n : Number) -> Done
}
```

For commonality with method declarations, parameters are normally named in type declarations. These names are useful when writing specifications of the methods. If a parameter name is omitted, it must be replaced by an underscore. The type of a parameter or result may be omitted, in which case the type is Unknown.

10.3 Type Declarations

Types, including parameterized types, may be named in type declarations. By convention, the names of types start with an uppercase letter. A simple type

literal consists of the keyword interface followed by an open curly brace, a sequence of method signatures, and a closed curly brace. The interface keyword may be omitted from the right-hand-side of a type declaration when the right-hand-side is a simple type literal. Type declarations may not be overridden.

Examples

```
type MyCatType = interface {  // the word interface may be omitted
    color -> Colour
    name -> String
}
// I care only about names and colours

type MyParametricType[[A,B]] =
    interface {
        at (_:A) put (_:B) -> Boolean
        cleanup(_:B)
    } where A <: Hashable,  B <: DisposableReference
```

10.4 Type Conformance

The key relation between types is **conformance**. We write B <: A to mean B conforms to A; that is, that B has all of the methods of A, and perhaps additional methods (and that the corresponding methods have conforming signatures). This can also be read as "B is a subtype of A", or "A is a supertype of B".

We now define the conformance relation more rigorously. This section draws heavily on the wording of the Modula-3 report @Modula3.

If B <: A, then every object of type B is also an object of type A. The converse does not apply.

If A and B are interfaces, then B <: A iff for every method m in A, there is a corresponding method m (with the same canonical name) in B such that

- If the method m in A has signature "m(P1,...,Pk)n(Pk+1,...,Pn)... -> R, and m in B has signature "m(Q1,...,Qk)n(Qk+1,...,Qn)... -> S", then
  - parameter types must be contravariant: Pi <: Qi
  - results types must be covariant: S <: R

The relationship used in **where** clauses to constrain type parameters of classes and methods has yet to be specified.

10.5 Composite types

Grace offers a number of operators to build up composite types.

10.5.1 Variant Types

The expression T1 | T2 | ... | Tn signifies an untagged, retained variant type. When a *variable* or *method* is annotated with a variant type, that variable may be bound to, or that method may return, an object of any one of the component types T1, T2, . . . , Tn. No *objects* actually have variant types, only expressions. The type of an object referred to by a variant variable (as determined by the type annotations in its declaration) can be examined using that object's reified type information.

The only methods in the static type of a receiver with a variant type are methods present in all members of the variant.

Variant types are *not* equivalent to the object type that describes all common methods. This is so that the exhaustiveness of match/case statements can be determined statically. Thus the rules for conformance are more restrictive:

```
S <: (S | T)
T <: (S | T)
(S' <: S) & (T' <: T)  ==>  (S' | T') <: (S | T)
```

Example

To illustrate the limitations on conformance of variant types, suppose

```
type S = {m: A -> B, n: C -> D}
type T = {m: A -> B, k: E -> F}
type U = {m: A -> B}
```

Then U fails to conform to S | T even though U contains all methods contained in both S and T.

10.5.2 Intersection Types

An object conforms to an Intersection type, written T1 & T2 & ... & Tn, if and only if that object conforms to all of the component types. The main uses of intersection types is for augmenting types with new operations, and as bounds on **where** clauses.

```
(S & T) <: S
(S & T) <: T
U <: S; U <: T; <==> U <: (S & T)
```

Examples

```
type List[[T]] = Sequence[[T]] & interface {
    add(_:T) -> List[[T]]
    remove(_:T) -> List[[T]]
}

class happy[[T]](param: T) -> Done
  where T <: (Comparable[[T]] & Printable & Happyable) {
```

```
        ...
}
```

10.5.3 Union Types

Structural union types (sum types), written `1 + 2 + ... + Tn`, are the dual of intersection types. A union type `T1 + T2` has the interface common to `T1` and `T2`. Thus, a type `U` conforms to `T1 + T2` if it has a method that conforms to each of the methods common to `T1` and `T2`. Union types are included for completeness: variant types subsume most uses.

```
S <: (S + T)
T <: (S + T)
```

10.5.4 Type Subtraction

A type subtraction, written `T1 − T2` has the interface of `T1` without any of the methods in `T2`. The signatures of the methods in `T2` are irrelevant.

10.5.5 Nested Types

Type definitions may be nested inside other expressions, for example, they may be defined inside object, class, method, and other type definitions, and typically accessed via Manifest Requests. This allows types to be declared and imported from other modules.

10.6 Type Assertions

When parameters, fields, and method results are annoatated with types, the programmer can be confident that objects bound to those parameters and fields, and returned from those methods, do indeed have the specified types, insofar as Grace has the required type information. The checks necessary to implement this guarantee may be performed statically or dynamically.

When implementing the type check, types specified as `Unknown` will always conform. So, if a variable is annotated with type

```
interface {
    add(Number) -> Collection[[Number]]
    removeLast -> Number
}
```

an object with type

```
interface {
    add(Unknown) -> Collection[[Unknown]]
    removeLast -> Unknown
    size -> Number
}
```

will pass the type test. Of course, the presence of Unknown in the type of the object means that a subsequent type error may still occur. For example, the code of the add(_) method might actually depend on being given a String argument, or the collection returned from add(_) might contain Booleans.

The same type check can be requested explicitly by using the operators <:, :> and == between types.

Examples

```
assert (B <: A) description "B does not conform to A"
assert (B <: type { foo(_) } ) description "B has no foo(_) method"
assert (B <: type {foo(_:C) -> D} ) description "B doesn't have a method foo(_:C)
    ->D"
assert (B == (A | C)) description "B is neither an A or a C"
```

11 Modules and Dialects

Grace programs can be divided into multiple modules. A module is typically used to define library functionality.

11.1 Modules

A module is typically defined in a implementation-dependent fashion, typically by creating a file containing Grace code. The text of the file is treated as the body of an object constructor, so it may contain both declarations and executable code. When a module is loaded, this object constructor is *executed*, resulting in a *module object*.

11.2 Importing Modules

Modules may begin with one or more **import** *moduleName* **as** *nickname* statements. *moduleName* is a string literal that identifies the module to be imported in an implementation-dependent manner; for example, *moduleName* may be a file path. *nickname* is the Grace identifier used to refer to the imported module object in the importing module. The nickname is confidential by default, but can be annotated as public.

Because importing a module creates a module object, public declarations at the top level of imported modules are accessed by requesting a method on the

module's nickname. Confidential declarations are not visible to the importing module.

11.3 Executing a Module

Grace programs are executed by asking the execution environment to run a particular module, which may be thought of as the "main" module. Grace will load and initialise all transitively imported modules in depth-first order, thus executing the "main" module *last*, after all its dependencies are loaded. Each imported module is loaded just once, the first time it is reached: importing the same *moduleName* multiple times results in the same module object. Circular module dependencies are errors.

Examples

cat.grace module:

```
import "animals" as a
print "initialising cat module"
class cat {
    inherit a.mammal
    method species { "cat" }
}
print "cat module done"
```

animals.grace module:

```
print "initialising animals module"
class mammal {
    method asString { "I am a {species}" }
    method species { "mammal" }
}
print "animals module done"
```

will print:

```
initialising animals module
animals module done
initialising cat module
cat module done
```

11.4 Dialects

Grace dialects support language levels for teaching, and domain-specific "little" languages. A module may begin with a dialect statement **dialect** "name", where the **dialect** keyword is followed by a string literal.

The effect of the dialect statement is to import the dialect like any other module, but then arrange that the dialect's module object lexically encloses the object

defined by the module. This means that Implicit Requests in the module can resolve to the definitions in the dialect.

Many features built in to other programming languages are obtained from dialects in Grace: this includes all preexisting type declarations, classes, traits, control structures, and even the 'graceObject' trait that defines the default methods.

Modules that do not declare a 'dialect' implicitly belong to the standardGrace dialect.

In addition to declarations, a dialect can also define a *checker* that examines the parse tree or syntax tree of any module written in the dialect, and generates errors. This enables a dialect to restrict the language of its modules to a subset of the full Grace language.

Examples

The bcpl.grace module declares an unless(_)do(_) control structure that is like if, but backwards.

bcpl.grace module:

```
method do (block: Block0) unless (test: Boolean)  {
    if (test.not) then (block)
}
```

A module written in this dialect can use that control structure as if it were built in:

example.grace module:

```
dialect "bcpl"
...
do { average := sum / count } unless (count == 0)
```

11.5 Module and Dialect Scopes

The **module scope** of a Grace module contains all declarations at the top level of the module, including the nicknames introduced by **import** declarations.

Surrounding the module scope is the **dialect scope**, which contains all public declarations at the top level of the module providing the dialect. That is, the public names at the top level of the dialect are treated as being in a scope surrounding that of any module written in that dialect.

Lexical lookup stops at the dialect scope: it does not extend to the scope surrounding the dialect (which would contain any other dialects used to implement the current dialect).

This allows dialects to import modules, and to be defined via other (module-defining) dialects, without those other definitions polluting the language defined by the dialect.

12 Pragmatics

The distribution medium for Grace programs, objects, and libraries is Grace source code.

Grace source files should have the file extension .grace. If, for any bizarre reason a trigraph extension is required, it should be .grc

Grace files may start with one or more lines beginning with #: these lines are ignored by the language, but may be interpreted as directives by an implementation.

12.1 Garbage Collection

Grace implementations should be garbage collected. GC may occur at any backwards branch, at any method request, and at any point where an object is constructed. Grace does not support finalization.

12.2 Concurrency

The core Grace specification does not describe a concurrent language. Various concurrency models may be provided as dialects. The details remain to be sepecified.

13 Acknowledgements

We thank Michael Homer and Tim Jones for working on early implementations of Grace, and Josh Bloch, Cay Horstmann, Michael Kölling, Doug Lea, Ewan Tempero, Laurence Tratt, and the participants at the Grace Design Workshops and IFIP WG2.16 on Programming Language Design meetings for discussions about the design of Grace.

14 Grammar

The following PEG defines the context-free syntax of Grace. Productions are arranged in alphabetical order.

```
addExpression ::= rep1sep(multExpression, addOp)
aliasClause ::= aliasId ~ methodHeader ~ equals ~ methodHeader ~ semicolon
argumentHeader ::= identifier ~ methodParams
argumentsInParens ::= lParen ~ rep1sep(drop(opt(ws)) ~ expression, comma) ~
    rParen
assignmentMethodHeader ::= identifier ~ assign ~ genericParams ~
```

```
    oneMethodParam
assignmentTail ::= assign ~ expression
basicTypeExpression ::= nakedTypeLiteral | literal | pathTypeExpression |
    parenTypeExpression
blockLiteral ::= lBrace ~ opt(ws) ~ opt(genericParams ~ opt(matchBinding) ~
    blockParams ~ opt(ws)~ arrow) ~ innerCodeSequence ~ rBrace
blockParams ::= repsep( identifier ~ opt(colon ~ typeExpression), comma)
classOrTraitDeclaration ::= (classId | traitId) ~ classHeader ~
    methodReturnType ~ whereClause ~ lBrace ~ rep(reuseClause) ~ codeSequence
    ~ rBrace
codeSequence ::= repdel((declaration | statement | empty), semicolon)
colon ::= both(symbol ":", not(assign))
comma ::= symbol(",")
declaration ::= varDeclaration | defDeclaration | classOrTraitDeclaration |
    typeDeclaration | methodDeclaration
defDeclaration ::= defId ~ identifier ~  opt(colon ~ typeExpression) ~ equals
    ~ expression
delimitedArgument ::= argumentsInParens | blockLiteral | stringLiteral
excludeClause ::= excludeId ~ methodHeader ~ semicolon
expression ::= opExpression
firstArgumentHeader ::= identifier ~ genericParams ~ methodParams
firstRequestArgumentClause ::= identifier ~ genericActuals ~ opt(ws) ~
    delimitedArgument
genericActuals ::= opt(lGeneric ~ opt(ws) ~ rep1sep(opt(ws) ~ typeExpression ~
    opt(ws), opt(ws) ~ comma ~ opt(ws)) ~ opt(ws) ~ rGeneric)
genericParams ::= opt(lGeneric ~ rep1sep(identifier, comma) ~ rGeneric)
hashLine ::= (symbol "#") ~ rep(anyChar | space) ~ (newLine | end)
identifier ::= guard(identifierString, { s -> ! parse(s) with(
    reservedIdentifier ~ end ) })
implicitSelfRequest ::= requestWithArgs |  rep1sep(unaryRequest,dot)
importStatement ::= importId ~ stringLiteral ~ asId ~ identifier ~ semicolon
innerCodeSequence ::= repdel((innerDeclaration | statement | empty),
    semicolon)
innerDeclaration ::= varDeclaration | defDeclaration | classOrTraitDeclaration
    | typeDeclaration
lBrack ::= both(symbol "[", not(lGeneric))
lineupLiteral ::= lBrack ~ repsep( expression, comma ) ~ rBrack
listOfOuters ::= rep1sep(outerId, dot)
literal ::= stringLiteral | selfLiteral | blockLiteral | numberLiteral |
    objectLiteral | lineupLiteral | typeLiteral
matchBinding ::= (stringLiteral | numberLiteral | (lParen ~ identifier ~
    rParen)) ~ opt(colon ~ nonEmptyTypeExpression)
matchingBlockTail ::= lParen ~ rep1sep(matchBinding, comma)  ~ rParen
methodDeclaration ::= methodId ~ methodHeader ~ methodReturnType ~
      whereClause
    ~ lBrace ~ innerCodeSequence ~ rBrace
methodHeader ::= assignmentMethodHeader | methodWithArgsHeader |
    unaryMethodHeader | operatorMethodHeader | prefixMethodHeader
methodParams ::= lParen ~ rep1sep( identifier ~ opt(colon ~ opt(ws) ~
    typeExpression), comma) ~ rParen
```

```
methodReturnType ::= opt(arrow ~ nonEmptyTypeExpression )
methodWithArgsHeader ::= firstArgumentHeader ~ repsep(argumentHeader,opt(ws))
moduleHeader ::= rep(hashLine) ~ rep(importStatement | reuseClause)
multExpression ::= rep1sep(prefixExpression, multOp)
nakedTypeLiteral ::= lBrace ~ opt(ws) ~ repdel(methodHeader ~
    methodReturnType, (semicolon | whereClause)) ~ opt(ws) ~ rBrace
nonEmptyTypeExpression ::= opt(ws) ~ typeOpExpression ~ opt(ws)
objectLiteral ::= objectId ~ lBrace ~ rep(reuseClause) ~ codeSequence ~ rBrace
      oneMethodParam ::= lParen ~ identifier ~ opt(colon ~ typeExpression) ~ rParen
operator ::= otherOp | reservedOp
operatorMethodHeader ::= otherOp ~ genericParams ~ oneMethodParam
otherOp ::= guard(trim(rep1(operatorChar)), { s -> ! parse(s) with( reservedOp
    ~ end ) })
parenExpression ::= lParen ~ rep1sep(drop(opt(ws)) ~ expression, semicolon) ~
    rParen
parenTypeExpression ::= lParen ~ typeExpression ~ rParen
pathTypeExpression ::= opt(listOfOuters ~ dot) ~ rep1sep((identifier ~
    genericActuals),dot)
prefixExpression ::= (opt(otherOp) ~ selectorExpression) | (otherOp ~
    listOfOuters)
prefixMethodHeader ::= opt(ws) ~ token("prefix") ~ otherOp ~ genericParams
primaryExpression ::= literal | listOfOuters | implicitSelfRequest |
    parenExpression
program ::= codeSequence ~ rep(ws) ~ end
rBrack ::= both(symbol "]", not(rGeneric))
requestArgumentClause ::= identifier ~ opt(ws) ~ delimitedArgument
requestWithArgs ::= firstRequestArgumentClause ~
    repsep(requestArgumentClause,opt(ws))
reservedIdentifier ::= selfLiteral | aliasId | asId | classId | defId |
    dialectId | excludeId | importId | inheritId | isId | methodId |
    objectId | outerId | prefixId | requiredId | returnId | traitId |
    typeId | useId | varId | whereId
reuseClause ::= (inheritId | useId) ~ expression ~ semicolon ~
    rep(reuseModifiers)
reuseModifiers ::= excludeClause | aliasClause
selector ::= (dot ~ unaryRequest) | (dot ~ requestWithArgs)
selectorExpression ::= primaryExpression ~ rep(selector)
semicolon ::= (symbol(";") ~ opt(newLine)) | (opt(ws) ~ lineBreak("left" |
    "same") ~ opt(ws))
statement ::= returnStatement | (expression ~ opt(assignmentTail))
stringChar ::= (drop(backslash) ~ escapeChar) | anyChar | space
stringLiteral ::= opt(ws) ~ doubleQuote ~ rep( stringChar ) ~ doubleQuote ~
    opt(ws)
typeDeclaration ::= typeId ~ identifier ~ genericParams ~ equals ~
    nonEmptyTypeExpression ~ (semicolon | whereClause)
typeExpression ::= (opt(ws) ~ typeOpExpression ~ opt(ws)) | opt(ws)
typeLiteral ::= typeId ~ opt(ws) ~ nakedTypeLiteral
typeOp ::= opsymbol("|") | opsymbol("&") | opsymbol("+")
typePredicate ::= expression
unaryMethodHeader ::= identifier ~ genericParams
```

```
unaryRequest ::= trim(identifier) ~ genericActuals ~ not(delimitedArgument)
varDeclaration ::= varId ~ identifier ~  opt(colon ~ typeExpression) ~
    opt(assign ~ expression)
whereClause ::= repdel(whereId ~ typePredicate, semicolon)
```