The Grace Programming Language Draft Specification Version 0.8.2

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.

Many of the expressions and commands in Grace are actually defined in the Grace standard dialect (and hence may not be supported in all dialects). Because some of these constructs are used in the examples in this document, we urge you to have a copy of the documentation for the standard dialect at hand when reading this document. It should be available in the directory where this document was found.

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.

The context-free syntax of Grace is described by an EBNF grammar. Individual productions are included in this specification where appropriate. The complete grammar, with all productions in alphabetical order, appears at the end.

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+27E7
[[	⟦	U+27E6

3.2 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
```

3.3 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. In the grammar, <newline> denotes a newline.

Tab characters (U+0009) 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 String Escapes.

3.4 Layout

Statements are separated by one or more newlines; it is also permissible, but uncommon, to separate statements by semicolons. In the grammar, the non-terminal Ss is used to represent a *statement separator*.

Grammar

```
Ss ::= ";"
     | <newline>
     | Ss  ( ";" | <newline> )
```

Grace uses braces to indicate the boundaries of code blocks. Indentation (the number of leading spaces on a line) must be consistent with these boundaries: indentation must increase after an opening brace, and return to the prior level with (or after) the matching closing brace.

Here are the precise rules that govern layout.

1. A line containing just spaces, or spaces and a comment, is ignored as far as layout is concerned.
2. All changes in indentation must be by *two* or more spaces; a change of a single space is treated as an error.
3. If a line contains an unmatched opening brace, that line is said to open a code block. All lines up to the matching closing brace comprise the body of the code block, and must be indented more than the line containing the opening brace.
4. If the closing brace that closes the code block is the first non-space character on a line, then the indentation of the closing brace must be the same as that of the line containing the matching opening brace. Otherwise, the line containing the closing brace must be indented like all the other lines in the code block.
5. An increase in indentation that does *not* correspond to the start of a code block indicates a continuation line: the preceding line break is treated as a space and not as a statement separator, and the two physical lines are treated as a single logical line. Further physical lines at the same (or greater) indentation are treated as part of the same logical line. The continuation ends either when the indentation decreases, or when the continuation line contains an unmatched brace.
6. If a line ends with any kind of opening bracket — one of (, [, *llbracket*, or { — the following line break is treated as a space, and *not* as a statement separator.
7. If a line starts with any kind of closing bracket — one of),], *rrbracket*, or } — the preceding line break is treated as a space, and *not* as a statement separator.
8. Indentation may be reduced *only* when ending a code block, or after the end of a continued line. The indentation must return to that of the line that began the code block, or the continued line, respectively.

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 example defines x to be the result of the single request mumble ("3") fratz (7). Because the second and third lines are indented more than the first, they continue that line.

Example of if(_)then(_)else(_):

```
if (condition) then {
    doSomething
} else {
    doAnotherThing
}
```

The body of the block that comprises the then action is indented *more than* the line that contains the opening {; the closing } is at *the same* indentation as the the line that contains the opening {. Because there is no line break after the first }, the else(_) does not start a separate statement.

Alternative Layout for if(_)then(_)else(_):

```
if (condition)
    then { doSomething }
    else { doAnotherThing }
theFollowingStatement
```

Here, the whole if(_)then(_)else(_) is on a single logical line; the indentation indicates that the then and else lines are a continuation of the if line. This format is appropriate only when the code blocks are small.

Bad Layout for if(_)then(_)else(_):

```
if (condition)
then { doSomething }
else { doAnotherThing }
```

This layout shows three separate statements — an if(_), a then(_), and an else(_). It is *not* a valid way of formatting a single if(_)then(_)else(_) statement.

Bad Layout for blocks that answer blocks:

```
def x = if (...) then { {
    print "true"
} } else { {
    print "false"
} }
```

This layout for an if(_)then(_)else() that answers a block can't work: it's not possible to close two blocks on the same line, because the second closing brace will violate rule 4.

3.5 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. In the grammar, <id> denotes an identifier.

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 ! ? @ # % ^ & | ~ = + − * / \ > < : . $, that are not reserved tokens. So, for example, +, ++ and .. are valid operators, but . is not, because it is reserved. In the grammar, <operator> denotes an operator.

3.6 Reserved Tokens

Grace has the following reserved tokens:

alias as class def dialect exclude import inherit interface **is method object**
once **outer prefix return self Self trait type** Unknown **use var where**
. ... := = ; { } [ ] ( ) : –> → [[ ]] ⟦ ⟧ //

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 Ellipsis

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.

Grammar

Ellipsis ::= "..."

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*. A decimal point, if present, must not be the first or last character of the numeral
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.

Grammar

Numeral ::= <decimalNumeral>
| <baseExponentNumeral>
| <explicitRadixNumeral>

Examples

```
1
42
3.14159265
0.25    // leading 0 is required
17.0    // the trailing 0 is required because of the decimal point
17      // same as the above
13.343e−12
414.45e3
16xF00F00
2x10110100
0xdeadbeef // Radix zero treated as 16
```

Note that there are no numerals for negative numbers; negative numbers can be generated by requesting the prefix − operator on a positive number.

Examples

```
−1
−2e4
```

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".

Grammar

```
Boolean ::= "true"
    | "false"
```

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; 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
\{	opening brace	U+007B

Escape	Meaning	Unicode
\}	closing 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 *interpolations*: expressions enclosed in braces. The value of a String Constructor is obtained by first evaluating any interpolations, requesting asString of the resulting object, and inserting the resulting string into the string literal in place of the interpolation.

Interpolations may contain String Literals, but not newlines or String Constructors. (Hence, interpolations may not contain nested interpolations.) In the grammar, a <stringSegment> denotes a sequence of characters that does not include unescaped ", newline, or {; it may contain the string escapes.

Example

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

4.5.3 Uninterpreted Strings

String literals can also be written between single guillemet quotation marks, ‹thus›. Between the ‹ and the ›, characters from the input become characters of the string without interpretation, and without any escapes (not even for ›). In the grammar, <uninterpretedString> denotes a sequence of *any* characters except ›.

Example

```
lexer.lex ‹// This is input for a test of the lexer.
// The input ends with a newline.
def s = "This is a String"
def n = 17
›
```

Grammar

```
String ::= StringLiteral
    | StringConstructor
    | UninterpretedString
StringConstructor ::= <dquote> <stringSegment>?  ( "{" Expression "}" <stringSegment>? )+  <dquote>
StringLiteral ::= <dquote> <stringSegment>?  <dquote>
```

4.6 Sequence Constructors

A Sequence Constructor is a comma-separated list of expressions surrounded by [and].

Examples

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

When executed, a sequence constructor returns an object of type Sequence. Sequences are immutable; they are most frequently used to initialize other collections, to control loops, and to pass options to methods.

Examples

```
set.withAll [ 1, 2, 4, 5 ]          // make a set
[ "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 }
]
```

Grammar

```
SequenceConstructor ::= "[" "]"
    | "[" Expression  ( "," Expression )*  "]"
```

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 assigned. Block parameters may optionally be annotated with types; omitted type annotations are treated as the type Unknown.

```
{ 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. It is a TypeError if an argument to apply(...) does not match the type annotation of the corresponding parameter.

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: Function1⟦Number, Number⟧ = { i:Number → sum := sum + i }
summingBlock.apply(4)       // sum now 4
summingBlock.apply(32)      // sum now 36
```

Grammar

```
Block ::= "{" BlockParameterList "→" Ss?  ( Statement  ( Ss Statement )*  )? "}"
    | "{"  ( Statement  ( Ss Statement )*  )? "}"
BlockParameterList ::= BlockParameter  ( "," BlockParameter )*
```

5 Declarations

Declarations may occur anywhere within a module, object, class, or trait. Constant and Variable 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 or temporary (but not a method or field) that has the same name as a lexically-enclosing field, method, parameter or temporary.

Grammar

```
Declaration ::= VarDeclaration
    | DefDeclaration
DefDeclaration ::= "def" Identifier TypeOption Annotations  ( "=" Expression )?
VarDeclaration ::= "var" Identifier TypeOption Annotations  ( ":=" Expression )?
TypeDeclaration ::= "type" Identifier TypeParameterList Annotations "=" TypeExpression
TypeOption ::= Empty
    | ":" TypeExpression
```

5.1 Constants

Constants are defined with the **def** keyword; they bind an identifier to the value of an initialising expression.

Constants may be optionally given a type: this type is checked when the constant is initialised. An omitted type annotation is treated as Unknown.

If the initialising expression is omitted, an annotation is required; in this case the declaration is a *marker declaration*. Constant declarations inside an object constructor create fields; others create temporary constants.

Examples

```
def x = 3 * 100 * 0.01
def x:Number = 3
def x:Number          // Syntax Error: x must be initialised
def volatile is annotation  // marker declaration
```

5.2 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. An omitted type annotation is treated as Unknown.

Variable declarations inside an object constructor create fields; others create temporary variables.

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
```

5.3 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 that name. Because every method must be associated with an object, methods may not be declared directly inside other methods. The body of the method is delimited by braces.

Specifying the return type is optional; an omitted return type is treated as Unknown. When the method returns, its result is checked against this type.

If the MethodBody is omitted, an annotation is required; in this case the method declaration is a *marker declaration*.

```
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}
}

method changeSpeedBy(delta) is abstract     // marker declaration
```

Grammar

```
MethodDeclaration ::= "once"? "method" MethodHeader ReturnTypeOption Annotations MethodBody?
MethodHeader ::= AssignmentMethodHeader
    | ParameterizedMethodHeader
    | ParameterlessMethodHeader
    | BinaryMethodHeader
    | UnaryMethodHeader
ReturnTypeOption ::= Empty
    | "→" TypeExpression
MethodBody ::= "{"  ( Statement  ( Ss Statement )*  )? "}"
Statement ::= Expression
    | Declaration
    | Assignment
    | Return
```

```
    | Import
    | <error>
AssignmentMethodHeader ::= Identifier ":=" TypeParameterList SingleMethodParameter
ParameterizedMethodHeader ::= <id> TypeParameterList MethodParameterList  ( <id> MethodParameterList )*
ParameterlessMethodHeader ::= <id> TypeParameterList
BinaryMethodHeader ::= <operator> TypeParameterList SingleMethodParameter
UnaryMethodHeader ::= "prefix" <operator> TypeParameterList
MethodParameter ::= Identifier TypeOption
MethodParameterList ::= "(" MethodParameter  ( "," MethodParameter )*  ")"
TypeParameterList ::= Empty
    | "⟦" TypeParameter  ( "," TypeParameter )*  Where "⟧"
```

5.3.1 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; 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.

Grammar

```
Return ::= "return" Expression?
```

5.3.2 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 has the same name as a method if their canonical names are equal.

1. A method can be named by a single identifier, in which case the method has no parameters; the *canonical name* of the method is the identifier.
2. A method can be named by a single identifier suffixed with :=; such a method is called an assignment method, and is conventionally used for writer methods, both user-written and automatically-generated. Assignment methods *always* take a single parameter after the :=, and have a *canonical name* of the identifier followed by :=(_). It is an error to declare a variable and an assignment method with the same identifier in the same scope.
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 be a *unary operator*, which has no parameters, and which is requested by a prefix operator expression. It can also be a *binary method*, which has 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 sequence of 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 }
```

Example of an assignment method

```
method value:= (n: Number) → Done {
    print "value currently {v}, now assigned {n}"
    v := n
}
```

This declares a method with canonical name value:=(_); such a method cannot be declared in the same scope as a variable value.

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 distinct methods. In other words, Grace allows "overloading by arity". (Grace does *not* allow overloading by type).

5.3.3 Parameters

Depending on their syntactic form, method declarations may include one or more lists of parameters. Inside method bodies, parameters are treated as constants: they may not be reassigned. Parameters to a method may optionally be annotated with types: the corresponding arguments will be checked against those types, either before execution, or when the method is requested. An omitted type annotation is treated as Unknown.

5.3.4 Type Parameters

Methods (including classes and traits) may be declared with one or more type parameters. If present, type parameters are listed between ⟦ and ⟧ after the identifier that forms the first (or only) part of the method's name.

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

Example

```
method indexOf⟦W⟧ (pattern:String) ifAbsent (absent:Function0⟦W⟧) → Number | W {
    // returns the leftmost index at which pattern appears in self;
    // applies absent if it is not there.
    ...
}

method list⟦T⟧ {
    object {
        method asString { "the list factory" }
        method empty –> List⟦T⟧ { list⟦T⟧ [] }
        method with (elem) –> List⟦T⟧ { list⟦T⟧ [elem] }
```

```
        method withAll (elems) -> List⟦T⟧ { list⟦T⟧ (elems) }
    }
}
```

In the first example, the ifAbsent block can return an arbitrary object. If this object has type W, then the result of the indexOf(_)ifAbsent(_) method will have type Number | W. The second example illustrates a method list with a single type parameter T, which is used as a type argument within the body of the object that it returns.

Type parameters may be constrained with **where clauses**. The reserved word **where** follows the final type parameter; if there is more than one where condition, the conditions are separated by commas.

Example

```
method sumSq⟦T where T <* Numeric⟧(a:T, b:T) → T  {
    (a * a) + (b * b)
}
```

The type relation in a where condition can be one of <:, :>,<*, or *>. <: indicates subtyping, while <* indicates matching. Matching is like subtyping, except that where **Self** appears in one argument, it must also appear in the other. :> and *> are the inverses of <: and <*.

Grammar

```
TypeParameterList ::= Empty
    | "⟦" TypeParameter  ( "," TypeParameter )*  Where "⟧"
TypeParameter ::= Identifier
Where ::= Empty
    | "where" WhereCondition  ( "," WhereCondition )*
WhereCondition ::= <id> <typeRelation> Type
```

5.3.5 Once Methods

A **once method** is declared by prefixing a method declaration with the reserved word once. Such a method completes execution at most once on each object with a given set of arguments: the first time that the object receives the corresponding request. The return value is memoized, and subsequent requests of the method with equal arguments will return the memoized value without re-executing the method. To make the memoization possible, the arguments must conform to EqualityObject, that is, they must have ==(_) and hash methods.

If a once method does *not* return a value, e.g., because it raises an exception on its first execution, then no value is memoized, and execution of the method will start again if it is requested anew. This process will repeat until the once method returns normally, at which point the return value will be memoized, and subsequent executions with equal argument lists will return the memoized value. Once methods can be used to represent lazily-initialized scalar constants, pure functions, and for any method whose result will not change once it has been calculated.

A parameterless once method differs from a constant field (declared with **def**) in that the latter is initialized as part of the process of creating its containing object, and is consequently *uninitialized* during part of that process. Hence, parameterless once methods are convenient for defining constants that may be used during initialization. They are also useful for defining a group of interdependent constants, because the programmer need not worry about the initialization order. Moreover, unlike a **def**, a once **method** is a method, and can appear in a trait.

Examples

```
def o = object {
    def nums = 1..100
    once method sum {
```

```
        nums.fold {a, b → a + b} startingWith 0
    }
}

once method fib(n) {
    // computes the nth Fibonacci number
    if (n ≤ 2) then { 1 }
               else { fib(n−1) + fib(n−2) }
}
```

The simple recursive definition of fib would take exponential time without the once. Because of the memoization provided by once, the above code takes linear time.

5.4 Annotations

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

Some annotations, like required, abstract and annotation, indicate that the declaration is a *marker declaration*, that is, a declaration without an initialiser or a method body.

Grace defines the following core annotations:

Annotation	Semantics
confidential	method may be requested only on self or outer — see Encapsulation
abstract	a marker declaration for method that must be provided when this component is reused
required	a marker declaration for method that is assumed to exist, but not provided, by the current class
override	method must override another method - see Overriding Methods
public	a public method may be requested from anywhere; a public variable field may be read and written from anywhere; a public field may be read from anywhere - see Encapsulation
readable	field may be read from anywhere - see Encapsulation
writeable	variable field may be written from anywhere - see Encapsulation
annotation	a marker declaration for a def or method that will be used as an annotation

Additional annotations can be defined by marker declarations annotated with **is** annotation. Annotations are identifiers, i.e., static labels, not runtime values. For example, a **def** declaration is public because it is annotated with the identifier public. It is not possible to make another identifier, say secret, mean the same thing as public by writing

```
def secret is annotation = public
```

Examples

```
var x is readable, writeable := 3
def maxSpeed:Number is public = 80
method foo is confidential  { "the method body" }
method id⟦T⟧ is required          // no method body
def annotation is annotation      // no initialiser
```

Grammar

```
Annotations ::= Empty
    | "is" AnnotationLabel  ( "," AnnotationLabel )*
AnnotationLabel ::= <id>
    | <id> TypeArguments AnnotationArgList  ( <id> AnnotationArgList )*
    | <id> TypeArguments
    | <id> AnnotationArgList  ( <id> AnnotationArgList )*
AnnotationArgList ::= "(" Expression  ( "," Expression )+  ")"
    | Numeral
    | String
    | SequenceConstructor
    | SpecialTerm
    | "(" Expression ")"
```

5.5 Encapsulation

Grace has different default encapsulation rules for methods, types, and fields; the defaults can be changed by explicit annotations. Grace defines two levels of visibility: public and confidential.

5.5.1 Public

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

5.5.2 Confidential

Confidential attributes can be requested only on **self**, or on an **outer** sequence, 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 reuse (i.e., **inherit** or **use**) d, and to objects lexically enclosed either by d itself, or by objects that reuse d.

5.5.3 Methods, Classes, Traits and Types

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

5.5.4 Fields

Variable (**var**) and constant (**def**) declarations immediately inside an object constructor create *fields* in that object. By default, fields are *confidential*.

A field declared as **var** x can be read using the request x and assigned to using the assignment x := .... A field declared as **def** y can be read using the request y, and cannot be assigned.

The default visibility can be changed using annotations. The annotation readable, applied to a **def** or **var** declaration, makes the accessor request available to any object. The annotation writable, applied to a **var** declaration, 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
}
```

5.5.5 No Private Attributes

Some other languages support "private attributes", which are available only to an object itself (and not its reusers). Grace does not have private fields or methods; all attributes can be accessed from reusers. However, identifiers from outer scopes can be used to obtain an effect similar to privacy.

Example simulating private fields

```
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:Point) {
            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. Notice also how the coordinates of the ship s and e are also inaccessible.

6 Objects, Classes, and Traits

A Grace **object** constructor generates an individual object. A Grace **class** declaration defines a method that generates a fresh object each time it executes; all of these objects have the same structure.

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

6.1 Objects

Object constructors are expressions that evaluate to an object with the attributes defined in the constructor. Each time an object constructor is executed, a fresh object is created. In addition to declarations of types, fields and methods, object constructors can also contain expressions (that is, 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.

Grammar

```
ObjectConstructor ::= "object" Annotations ObjectBody
ObjectBody ::= "{"  ( ObjectItem  ( Ss ObjectItem )*  )? "}"
ObjectItem ::= Statement
    | MethodDeclaration
    | TypeDeclaration
    | ClassDeclaration
    | TraitDeclaration
    | UseStatement
    | InheritStatement
Statement ::= Expression
    | Declaration
    | Assignment
    | Return
    | Import
    | <error>
```

Examples

```
object {
    def colour:Colour = colours.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 the object created by object constructor, like this:

```
def unnamedCat = object {
     def colour:Colour is public = colours.tabby
     def name:String is public = "Unnamed"
     var miceEaten is readable := 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 freshly-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 {
        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 (colours.tortoiseshell) named "Fergus"
```

This creates an object with fields colour (set to colours.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.

Grammar

```
ClassDeclaration ::= "class" MethodHeader ReturnTypeOption Annotations ObjectBody?
```

If the MethodBody is omitted, an annotation is required; in this case the class declaration is a *marker declaration*.

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 directly contains no field declarations, **inherit** statements, or executable code. A trait can **use** other traits. Methods in a trait can capture variables in the lexical scope of the trait, so that they can have what is effectively private state, as illustrated in the Section on private attributes. Note that a trait object can contain types, traits, and classes; these classes *can* contain field declarations, and can inherit.

Aside from these restrictions, Grace's **trait** syntax and semantics is parallel to the class syntax. In particular, the reserved word **trait** defines a *method* that returns a trait object. Hence, in the following example, emptiness1 and emptiness2 are both methods, and both create and return equivalent traits objects.

Examples

```
trait emptiness1 {
    method size is required
    method isEmpty { size == 0 }
    method nonEmpty { size ≠ 0 }
    method ifEmptyDo (eAction) nonEmptyDo (nAction) {
            if (isEmpty) then { eAction.apply } else { do(nAction) }
    }
}

method emptiness2 {
    object {
        method size is required
        method isEmpty { size == 0 }
        method nonEmpty { size ≠ 0 }
        method ifEmptyDo (eAction) nonEmptyDo (nAction) {
                if (isEmpty) then { eAction.apply } else { do(nAction) }
        }
    }
}
```

The advantage of emptiness1 is that it makes the programmer's intention clearer. Moreover, if a field or an inherit statement were inadvertently added to emptiness1, the implementation would immediately complain. With the second form, the error would be found only when emptiness2 is **use**d.

6.4 Type Parameters

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

Example

```
class vectorOfSize(size)⟦T⟧ {
    var contents := Array.size(size)
    method at(index: Number) → T { return contents.at(index) }
    method at(index: Number) put(elem: T) { ... }
}

class sortedVector⟦T⟧
    where T <∗ Comparable⟦T⟧ {
      ...
}
```

6.5 Reuse

Grace supports reuse in two ways: through **inherit** statements and through **use** statements. Object constructors (including classes and modules) can contain one **inherit** statement, while traits cannot contain an **inherit** statement; object constructors, classes, modules and traits can all contain one or more **use** statements. As a special case, an object with no **inherit** statements is treated as though it inherited from graceObject (although this implicit inheritance does not disqualify the object from being a trait).

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 methods in the parent that originated in graceObject, while **use** clauses do not.

The expression parent in an **inherit** parent and **use** parent clause must be a Manifest Expression that returns a Fresh Object; usually this will be a request on a class or trait. The expression parent cannot depend on **self**, implicitly or explicitly, because **self** does not exist until after the reuse statement containing parent has been evaluated.

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** new(_) = old(_) creates a new confidential *alias* new(_) for the attribute old(_). Attributes of the parent that are not wanted can be excluded using an **exclude** clause, which consists of the reserved word **exclude** followed by the canonical name of an attribute. It is an *object composition error* to alias or exclude attributes that are not present in the object being inherited, or to alias an attribute to its own name.

The set of attributes that is introduced by the reuse statement is determined as follows.

- The set contains all the attributes of the parent, except for those that appear in an **exclude** clause (and, in the case of a **use** statement, except for the attributes obtained from graceObject).
- The set contains all of the new attribute names introduced by the **alias** clauses. Each of these names is bound to the attribute to which the old method name is bound in the parent.

The order of the alias and exclude clauses is irrelevant.

Attributes introduced by an alias clause are treated as being introduced by the object under construction, and thus do *not* conflict with (and may therefore override) attributes obtained by reuse. They *do* conflict with attributes declared in the object under construction.

The method names in alias and exclude clauses have the same syntax as method names in method declarations and interface literals; this means that they can contain both parameter names and type annotations. Such names and annotations may be useful as documentation, but do not affect the meaning of the program.

Examples

```
trait t1 {
    method x(size:Number) { ... }
    method y(name:String) { ... }
}

class c1 {
    use t1 alias w(_) = y(_) exclude x(_)
    method v { ... }
}
```

Objects generated by `c1` have attributes `v`, `w(_)` and `y(_)`, but not `x(_)`

```
trait t1 {
    method x(size:Number) { ... }
    method y(name:String) { ... }
}

class c1 {
    use t1 alias w(name) = y(name) exclude x(_)
    method w(kind) { ... }
}
```

This is a trait composition error, because `c1` gets a method with canonical name `w(_)` from two places: an alias clause, and a method definition.

```
trait t1 {
    method x { ... }
    method y { ... }
}

class c1 {
    use t1 alias x = y
    method y is override { ... }
    method w { ... }
}
```

This is also a trait composition error, because `x` is defined twice: in `t1`, and in the alias clause. This can be corrected by excluding `x` from `t1`.

```
trait compare {
    method lessThanOrEqual(a, b) { a.name ≤ b.name }
}

trait moreCompare {
    use compare alias greaterThanOrEqual(x, y) = lessThanOrEqual(y, x)
}
```

In this example, the trait `moreCompare` has two methods, `greaterThanOrEqual(_,_)` and `lessThanOrEqual(_,_)`, *but these method are identical*, and therefore the names are misleading. If the intension is to make `greaterThanOrEqual(_,_)` perform the inverse comparison, this should have been written as

```
trait moreCompare {
    use compare
    method greaterThanOrEqual(x, y) is confidential {
        lessThanOrEqual(y, x)
    }
}
```

Grammar

```
InheritStatement ::= "inherit" Expression ( ReuseModifier )*
UseStatement ::= "use" Expression ( ReuseModifier )*
ReuseModifier ::= ExcludeClause
    | AliasClause
ExcludeClause ::= "exclude" MethodHeader
AliasClause ::= "alias" MethodHeader "=" MethodHeader
```

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. Fields (**def**s and varss) thus installed are uninitialised.

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. The term "local declarations" comprises declarations of methods, types and fields (both **def**s and varss).

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 a method requested on self by the parents' initialisers.

6.5.2 Abstract Methods

Methods may be declared to be *abstract* by annotating the method header with abstract, and omitting the method body. Abstract methods do not override normal methods. Requesting an abstract method will generate an error.

6.5.3 Required Methods

Methods may be declared to be *required* by annotating them as required, and omitting the method body. This indicates that a normal method with a body must be supplied. Required methods do *not* conflict with other methods. In particular, a required local method does not override a method from a parent; instead the parent is said to *supply* the requirement. Similarly, a method required by a used trait can be supplied by another used trait without any conflict. Requesting a required method that has not been supplied will generate an error.

6.5.4 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 canonical name. 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.

6.5.5 Overriding Types

If a type declared in the current object has the same name as a type declared in a parent, the two types must be *identical*.

6.5.6 Default Methods

All objects implement a number of *default methods* by inheriting from graceObject. Programmers can override these implementations with alternative implementations. Type Object contains just the public default methods.

Method	Return value
isMe (other:Object) → Boolean; *confidential*	true if other is the same object as self
myIdentityHash → Number; *confidential*	a hash code characteristic of this object
asString → String	a string describing self
asDebugString → String	a string describing the internals of self

Notice that graceObject implements neither == nor *neq*. In the *standard* dialect, the trait equality is available to help in their implementation.

```
trait equality {
    method == (other) is required
    method hash is required
            // should obey invariant (a == b) => (a.hash == b.hash)
    method ≠ (other)  { (self == other).not }
    method :: (obj) { binding.key (self) value (obj) }
}
```

As the is required indicates, an object using this trait must provide an == method, and a corresponding hash method. One way to define these methods is by combining the equality and hash on the results of all the observer methods; another is to use the trait identityEquality, which defines == as object identity and hash as identity hash.

```
trait identityEquality {
    use equality
    method == (other) { self.isMe(other) }
    method hash { self.myIdentityHash }
}
```

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.

Grammar

```
Request ::= ImplicitRequest
    | SelfRequest
    | OuterRequest
    | DottedRequest
ImplicitRequest ::= RequestPartsWithArguments
SelfRequest ::= "self" <dot> RequestPart
OuterRequest ::= ( "outer" <dot> )+  RequestPart
DottedRequest ::= Term <dot> RequestPart
RequestPart ::= RequestPartNoArguments
    | RequestPartsWithArguments
RequestPartNoArguments ::= <id>
RequestPartsWithArguments ::= <id> TypeArguments ArgumentList  ( <id> ArgumentList )*
    | <id> TypeArguments
    | <id> ArgumentList  ( <id> ArgumentList )*
```

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. Because of inheritance and trait use, this may not be the definition of x that appears in the current object constructor.

The reserved word **Self** (capitalised) may appear in an object or in an interface. In an object, it refers to the type of the object **self**; in an interface, it refers to the type of which that interface is a part. Because interfaces can be combined using & and |, the meaning of **Self**, like that of **self**, depends on the context in which it is evaluated.

Examples

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

type Copyable = interface { copy → Self }
type Key = interface { unlock(_) → Done }
type CopyableKey = Copyable & Key      // the result of the copy method
// in a CopyableKey is also a CopyableKey
```

7.2 Outer

The reserved word **outer** refers to the object lexically enclosing the current object; **outer.outer** (an **outer** sequence of length 2) refers to the object enclosing **outer**, and so on. Note that an **outer** sequence is not a request, and that **outer** is a reserved word, not the name of a message. The expression **outer**.x requests x on the object lexically enclosing **self**.

Grammar

```
Self ::= "self"
Outer ::= "outer"
    | ( "outer" <dot> )+  "outer"
```

Examples

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

Because **outer** is lexical, two methods in the same object may have different **outer** objects. For example, one method may be inherited, while the other is defined locally.

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 *receiver* is an expression; when evaluated it 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.

Unlike some other languages, Grace does *not* allow the overloading of method names by type: the type of the arguments supplied to the request does not influence the method being requested. However, the *number* of arguments in an argument list does determine the method being requested.

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 constructor, boolean constant (**true** or **false**), sequence constructor, block, **self** or **outer** sequence. These forms are self-delimiting, and are readily distinguished from the identifiers that comprise the name of the method being requested.

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
}
```

Grammar

```
Request ::= ImplicitRequest
    | SelfRequest
    | OuterRequest
    | DottedRequest
SelfRequest ::= "self" <dot> RequestPart
OuterRequest ::= ( "outer" <dot> )+  RequestPart
DottedRequest ::= Term <dot> RequestPart
RequestPart ::= RequestPartNoArguments
    | RequestPartsWithArguments
RequestPartNoArguments ::= <id>
RequestPartsWithArguments ::= <id> TypeArguments ArgumentList  ( <id> ArgumentList )*
    | <id> TypeArguments
    | <id> ArgumentList  ( <id> ArgumentList )*
```

7.3.2 Implicit Requests

If the receiver of a method request is **self** or an **outer** sequence, the receiver may be left implicit, *i.e.*, the **self** or **outer** sequence, and the following dot, may both be omitted. An implicit request is interpreted as a **self** request, or as an **outer** request on an outer sequence of the appropriate length.

When interpreting an implicit request of a method named m, 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, in the nested scope in which the implicit request is written, and not within the scope of any object (class, or trait) that may inherit the method containing the implicit request.

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"
```

Grammar

```
ImplicitRequest ::= RequestPartsWithArguments
```

7.4 Assignments and Assignment Requests

An assignment is a variable followed by := ; an assignment request is 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 :=.

An assignment binds the variable to the value of the argument, and returns done. An assignment method executes the method body; by convention, assignment methods also return done;

Examples

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

Grammar

```
Assignment ::= Identifier ":=" Expression
    | AssignmentRequest
AssignmentRequest ::= Term <dot> <id> ":=" Expression
    | "self" <dot> <id> ":=" Expression
    | ( "outer" <dot> )+  <id> ":=" Expression
```

7.5 Binary Operator Requests

Binary operators are methods whose names are <operator>s Binary operators have a receiver and one argument; the receiver must be explicit.

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

Grammar

```
BinaryRequest ::= Factor  ( <operator> TypeArguments?  Factor )+
```

7.6 Unary Prefix Operator Requests

Grace supports unary methods named by operator symbols that precede the explicit receiver. (Since binary operator requests 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
```

Grammar

```
UnaryRequest ::= <operator> TypeArguments? Term
```

7.7 Precedence of Method Requests

The precedence of method requests is defined by Grace's Grammar. The grammar implies the following precedence levels, where lower numbers bind more tightly.

1. Numerals and constructors for strings, objects, collections, 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. Infix operators, whose binding must be given explicitly using parenthesis, except that a repeated sequence of the same operator need not be parenthesized, and associates to the left.
5. Assignments, and method requests that use := as a suffix to a method name.

There is one exception to the rule that the binding between infix operators must be given explicitly: * "multiplicative" operators ∗ and / are left-associative, and bind more tightly than * "additive" operators + and −, which also left associative.

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 Unknown.

Examples

```
sumSq⟦Number⟧(1, 20)

sumSq(1, 20)
```

7.9 Manifest Expressions

The parent expressions in **inherit** parent and **use** parent statements must be *manifest*. This means that Grace must be able to determine the fields and methods defined in the object that is being inherited on a module-by-module basis.

If parent is an implicit request, it is first converted to an explicit request by applying the disambiguation rules for Implicit Requests. Once disambiguated, let the parent expression be $r.p_1.p_2. \ldots .p_n$, where the p_i are canonical names. The expression $r.p_1.p_2. \ldots .p_n$ is manifest if the receiver r is

1. bound to a module in an **import** statement, or
2. an **outer** sequence that refers to a module

and, for all i, p_i is defined in a DefDeclaration a MethodDeclaration, or a ClassDeclaration, and the value bound to, or returned by, p_i is an object.

Note that the arguments to a manifest expression need not themselves be manifest.

Example

Consider a module containing the following code:

```
class a {
    class x {
        method one {}
        method two {}
    }

    class b {
        inherit outer.outer.a.x
            // this uniquely defines x, without the possibility of overriding
        method three { ... }

    }
}
```

Suppose that the current module imports the above module with nickname m, and that the current module defines a class c that inherits m.a and overrides x:

```
import "module above" as m
class c {
    inherit m.a
    class x { ... }
}
```

If class b were to simply **inherit** x, then c.b would acquire the fields and methods of this overriding x — which are unknown to m. By writing **outer.outer** to refer to the module enclosing a the parent expression in b's **inherit** statement is made to refer to x lexically that is, the parent expression becomes manifest (rule 2 above).

7.10 Fresh Objects

The parent expression in **inherit** parent and **use** parent statements must return a *fresh* object, that is, a newly-created object to which there is no other reference.

1. An object returned from a method that is defined using the class or trait syntax is always fresh.
2. Any method that contains no **return** statements, and whose final statement is (or returns) an object constructor, returns a fresh object.
3. Any method that contains no **return** statements, and whose final statement is (or returns) a Manifest Expression that yields a fresh object, itself returns a fresh object.

8 Pattern Matching

Pattern matching is based on Pattern objects that respond to the matches(subject) request by returning a Boolean, which is either **false** if the match fails, or **true** if the match succeeds.

- All type objects are Patterns, which match objects that have that type.
- Numbers, Booleans and Strings are *self-matching*: they are patterns that match themselves. So, 5 is a pattern that matches the number 5, and **true** is a pattern that matches the Boolean true.
- The prefix operations $<$, $\leq$, $>$ and $\geq$ on numbers return appropriate patterns, so $\geq$5 is a pattern that matches any number greater than or equal to 5.
- In addition, libraries supply Patterns, and programmers are free to implement their own Patterns.
- Patterns can be combined with the pattern operators & (for and), | (or), and prefix *neg* (not).

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.matches(cp)          // true
Point.matches(42)          // false
```

8.1 Blocks as Patterns

Blocks are also patterns, that is, they respond to the request matches(_) as well as apply(_). When apply(_) would raise a type error because the block's argument would not conform to its parameter type, matches(_) returns **false**.

The parameter declarations of a block take the form Identifier PatternOption, rather than Identifier TypeOption. This means that the annotation after the : can be any Expression that evaluates to a Pattern, and is not restricted to being a TypeExpression.

If the Identifier in the BlockParameter is _, and the PatternOption is not empty, then the underscore and the following colon can be omitted, provided that the pattern is not an identifier, that is, if it is parenthesized, or is a string constructor, a boolean literal, or a numeral. This rule (the *delimited argument rule*) means that the pattern can be distinguished from the declaration of a parameter to the block.

Grammar

```
BlockParameter ::= Identifier PatternOption
    | NonIdExpression
PatternOption ::= Empty
    | ":" Expression
NonIdExpression ::= BinaryRequest
    | NonIdFactor
NonIdFactor ::= NonIdTerm
    | ObjectConstructor
    | UnaryRequest
NonIdTerm ::= DelimitedTerm
    | InterfaceLiteral
    | UnknownType
    | SelfType
    | Request
    | Ellipsis
```

8.2 Match ... case ... else

Matching blocks and self-matching objects can be conveniently used in the match(_)case(_)...else(_) family of methods; case may appear multiple times, with a block as argument. The else is optional; if present, it must be followed by a parameterless block.

If more than one of the case patterns is true, a MatchError is raised. If none of the case patterns is true, the else block is executed, if there is one; if not, a MatchError is raised.

Examples

```
once method fib(n:Number) → Number {
    match (n)
        case { 0 → 0 }
        case { 1 → 1 }
        case { >1 → fib(n−1) + fib(n−2) }
}
```

The patterns in the first two blocks use self-matching objects. { 0 → 0 } is short for { _:0 → 0 }. The first two cases could be combined into { 0|1 → n }. The pattern in the third block uses the prefix > operator to create a pattern that matches any number greater than 1.

If fib is requested with a negative argument, none of the pattern blocks will match, and a MatchError will be raised.

```
{ 0 → "Zero" }
    // match against the Number literal 0

{ s:String → print(s) }
    // match against the type String, 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 the empty type annotation; equivalent to a:Unknown.
    // This matches any object, and binds it to `a`, and hence
    // is not useful in combination with other case−matching blocks
```

9 Exceptions

Grace supports exceptions (more precisely, exception packets), which can be raised and caught. 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, and an optional data object.

Raising an exception does two things: it creates an ExceptionPacket 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. Execution continues when the exception is *caught.*

Examples

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

9.1 Kinds of Exception

Grace defines a hierarchy of *kinds* of exception; each kind of exception corresponds to a different kind of exceptional situation. All exceptions have the same *type*, that is, they understand the same set of requests. A hierarchy of *exception kinds* is used to classify exceptions.

```
type ExceptionKind = Pattern & {
    parent → ExceptionKind
    // answers the ExceptionKind that is the parent of this exception in the
    // hierarchy. The parent of Exception is defined to be Exception. The parent
    // of any other ExceptionKind is the object that was refined to create it.

    refine (name:String) → ExceptionKind
    // answers a new ExceptionKind object, which is a refinement of self.

    name → String
    // answers the name given when this ExceptionKind object was created.

    raise (message:String) → None
    // creates an exception of this kind, terminating the current execution,
    // and transferring control to an appropriate handler.

    raise (message:String) with (data:Object) → None
    // similar to raise(_), except that the object data is associated with the
    // new exception.

    == (other:Object) → Boolean
    // answers true if other is an ExceptionKind such that parent == other.parent
    // and name = other.name, otherwise false.
}
```

The root of the hierarchy of ExceptionKinds is Exception; all other ExceptionKinds are (direct or indirect) refinements of Exception. The name of Exception is "Exception", and the parent of Exception is Exception itself.

Because ExceptionKinds are also Patterns, they support the pattern protocol (matches, &, and |). This means that ExceptionKinds can be used as the patterns of the catch blocks in a try(_)catch(_)...finally(_) construct. An ExceptionKind object e matches any exception raised from e', and any exception raised by a refinement of e', for all e' == e.

Grace defines three direct refinements of Exception:

- EnvironmentException: those exceptions arising from interactions between the program and the environment, including network exceptions, file system exceptions, and inappropriate user input.

- ProgrammingError: exceptions arising from programming errors. Examples are IndexOutOfBounds, NoSuchMethod, and NoSuchObject.
- ResourceException: exceptions arising from an implementation insufficiency, such as running out of memory or disk space.

Notice that there is no category for "expected" exceptions. This is deliberate; expected events should not be represented by exceptions, but by other values and control structures. For example, if you you have a key that may or may not be in a dictionary, you should not request the at method and catch the NoSuchObject exception. Instead, you should request the at(_)ifAbsent(_) method.

9.2 Exception Packets

Exception packet objects are generated when an exception is raised.

```
type ExceptionPacket = type {
    exception → ExceptionKind    // the exceptionKind that raised this exception.
    message → String             // the message provided when this exception was raised.

    data → Object                // the data object associated with this exception
                                 // when it was raised, if there was one. Otherwise,
                                 // the string "no data".

    lineNumber → Number          // the source−code line of the raise request
                                 //  that created this exception.

    moduleName → String          // the name of the module containing the raise
                                 // request that created this exception.

    backtrace → List⟦String⟧
    // a description of the call stack at the time that this exception was raised.
    // backtrace.first is the initial execution environment; backtrace.last is the
    // context that raised the exception.
}
```

The data field of an ExceptionPacket may be populated with any object by requesting raise(_)with(_) on an ExceptionKind object. For example:

```
MyException.raise "A message" with (dataObject)
```

The dataObject is stored in the exception packet so that it can be used (if desired) when the exception is caught.

9.3 Catching Exceptions & Final Actions

An exception in expression can be caught by a dynamically-enclosing try(_)catch(_)... or try(_)catch(_)...finally(_) request, which takes the following form.

```
try { expression }
    catch { e1:Exception_1 → block_1 }
    ...
    catch { en:Exception_n → block_n }
    finally { finalBlock }
```

If an exception is raised during the evaluation of expression, the catch blocks 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(_)...finally(_).

The clause finally { finalBlock } is optional. If present, finalBlock is always executed before control leaves the try(_)catch(_)...finally(_) construct, whether or not an exception is raised, and whether or not expression, or one of the catch blocks, executes a **return**.

The value of a try(_)catch(_)...finally(_) request is the last value in expression, unless an exception is raised, in which case it is the the last value in whichever catch-block catches the exception. If the finalBlock executes to completion, its value is ignored.

If finalBlock terminates by raising an exception, or by executing a **return**, any prior **return** or raised exception is forgotten.

Examples

```
try {
    def f = io.open("data.store", "r")
} catch {
    e: NoSuchFile → print "{e.message}\nFile does not exist."
} catch {
    e: PermissionError → print "Permission denied"
} catch {
    _: Exception → print "Unidentified Error"
    system.exit(1)
} finally {
    f.close
}
```

A single handler may be defined for more than one kind of exception using the | pattern combinator:

```
try {
    try_block
} case { e:MyError | AnotherError →
    handler
}
```

handler will be run when either MyError or AnotherError is raised inside the try_block.

10 Types

Grace uses structural typing, as do Modula-3 and WhiteOak. Malayeri and Aldrich discuss the differences between nominal and structural typing.

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. Dialects can implement a variety of static typing regimes.

10.1 Predeclared Types

A number of types are declared in the *standard* dialect, and included in most other dialects, including None, Done, Boolean, Object, Number, String, Function$n$, Procedure$n$, Predicate$n$, Iterator, Pattern, ExceptionPacket, ExceptionKind, and Type.

10.1.1 Type None

Type None has all methods. It is "uninhabited", that is, no actual object has type None.

10.1.2 Type Object

In *standard*, type Object includes just the public Default Methods declared in graceObject.

```
type Object = interface {
    asString → String                  // a string for use by the client
    asDebugString → String               // a string for use by the implementor
}
```

Notice that isMe, and myIdentityHash, although present in graceObject, are not present in type Object, because they are *confidential*.

10.1.3 Type EqualityObject

In *standard*, type EqualityObject adds the family of equality methods to Object:

```
type EqualityObject = Object & interface {
    ::(value:Object) → Binding
    ==(other:Object) → Boolean
    ≠(other:Object) → Boolean
    hash → Number
    prefix == → Pattern
}
```

10.1.4 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.

10.1.5 Types Function, Procedure, and Predicate

The type Function0⟦T⟧ describes a block with zero parameters that returns a result of type T. Function1⟦A1,T⟧ describes a block with one parameter of type A1 and a result of type T. Function2⟦A1, A2,T⟧ describes a block with two parameters of types A1 and A2, and and a result of type T. Function3⟦A1, A2, A3, T⟧ describes a block with three parameters of types A1, A2, and A3, and and a result of type T.

The type Procedure$n$ (where $n = 0$, 1, 2, or 3) is like Function$n$, except that the result type is Done. The type Predicate$n$ (where $n = 0$, 1, 2, or 3) is like Function$n$, except that the result type is Boolean. So, for example, Predicate1⟦A1⟧ is the type of an object with an apply method that expects an argument of type A1, and returns a Boolean.

10.1.6 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. Omitted type arguments are also equivalent to Unknown.

Static 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
```

10.1.7 Type Type

All types have type Type, which is defined as

```
type Type⟦T⟧ = interface {
    name → String                  // the name of this type
    isNone → Boolean         // true for the type None, otherwise false
    matches (value:Object) → Boolean
    & (other:Type) → Type
    | (other:Type) → Type
    + (other:Type) → Type
    :> (other:Type) → Boolean       // other conforms to self
    <: (other:Type) → Boolean       // self conforms to other
    :=: (other:Type) → Boolean      // (self <: other) && (other :> self)
    == (other:Type) → Boolean        // object identity
    ≠ (other:Type) → Boolean
    hash → Number
    interfaces → Sequence⟦Interface⟧
    subject → Type                 // the parameter T
    asString → String
    asDebugString → String
}
```

This type captures the idea that a type is a disjunction of interfaces. The interface literal syntax defines a type containing a single interface, so the interfaces method of an interface returns a sequence of length 1 containing itself. (The object identity operation is necessary to avoid infinite regress when comparing two recursive types for conformity.)

```
type Interface⟦T⟧ = Type⟦T⟧ & interface {
    methods → Dictionary⟦String, Signature⟧
        // keys are the canonical names of the methods,
        // and values their signatures
    − (other:Interface) → Interface
}

type Signature = interface {
    name → String
        // the canonical name of the method
    arguments → Sequence⟦Type⟧
        // the types of the parameters, in order
    result → Type
        // the type of the result
}
```

These types say that each interface comprises a mapping from (canonical) method names to method signatures, and a mapping from type names to type objects. Each Signature comprises the (canonical) name of the method, the types of its arguments, and the type of its result.

10.2 Interfaces and Interface Literals

Interfaces characterize objects by detailing their public methods, and the types of the parameters and results of those methods. A readable field x:T is equivalent to a method x → T, and a writable field y:T is equivalent to a method y:=(*nu*:T) → Done. A declaration **type** N = TypeExpr is represented in the interface by a method N → Type⟦TypeExpr⟧

The various cat object constructors and classes described above (Objects, Classes, and Traits) create objects that conform to this interface:

```
interface {
    colour → Colour
    name → String
    miceEaten → Number
    eatMouse → Done
    asString → String
    asDebugString → String
}
```

Note that the public methods of graceObject, inherited by the cat objects, are included in the interface literal, but confidential methods are excluded.

For commonality with method declarations, parameters are normally named in interface literals. These names are useful when writing specifications of the methods. If a parameter name is omitted, it must be replaced by an underscore, as in method at(_) ifAbsent(_). 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.

Types are disjunctions of interfaces; interfaces are sets of methods. An interface literal consists of the keyword interface followed by an opening brace, a sequence of method signatures, and a closing brace. Type declarations may not be overridden.

Examples

```
type MyCatType = interface {
  // I care about just names and colours
    color → Colour
    name → String
}

type MyParametricType⟦A,B
      where A <: Hashable, B <: DisposableReference⟧ = interface {
        at (_:A) put (_:B) → Boolean
        cleanup(_:B)
}
```

Grammar

```
TypeDeclaration ::= "type" Identifier TypeParameterList Annotations "=" TypeExpression
TypeParameterList ::= Empty
    | "⟦" TypeParameter  ( "," TypeParameter )*  Where "⟧"
InterfaceLiteral ::= "interface" "{" "}"
    | "interface" "{" Signature  ( Ss Signature )*  "}"
TypeExpression ::= Type  ( <typeOperator> TypeArguments?  Type )*
Where ::= Empty
```

```
    | "where" WhereCondition  ( "," WhereCondition )*
WhereCondition ::= <id> <typeRelation> Type
```

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.

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 if and only if, for every method with canonical name m in A, there is a method with the same canonical name m 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

10.5 Composite types

Grace offers a number of operators to compose 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 type bounds in **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 of unions.

```
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 declarations may be nested inside objects, and hence also inside classes and traits. This allows types to be declared in, and imported from, other modules. Such declarations can be accessed by making requests on the containing object. Because type declarations cannot be overridden, such requests are Manifest Requests.

10.6 Type Annotations

When parameters, fields, and method results are annotated 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.

10.6.1 Static Type Checking

When implementing the static 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.

Static type checking is implemented by dialects; various static typing dialects may impose varied restrictions on Grace.

10.6.2 Dynamic Type Checking

Currently, the dynamic interpretation of types is *shallow*, that is, it considers only the methods present in an interface, and not the types of the arguments or the results of those methods. This is because, in the absence of type annotations, Grace has no information about the argument types or the return type of a method. This means that if programmers annotate a declaration

```
var x:Number
```

they can be sure that any object assigned to x has a method +(_), but are *not* assured that this +(_) method will expect an argument that is also a Number, nor that the result will be a Number, even though these details are part of the Number interface. Similarly, when the operators <:, :> and == between types are evaluated dynamically, argument and result types are ignored, even if they are present in the type definitions.

This treatment is types not entirely satisfactory, and is subject to review and change.

Examples

```
assert (B <: A) description "B does not conform to A"
assert (B <: interface { foo(_) } ) description "B has no foo(_) method"
assert (B <: interface {foo(_:C) → D} ) description "B has no foo(_) method"
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 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*.

11.1.1 Importing Modules

Modules may begin with one or more **import** *moduleName* **as** *nickname* statements, where *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. In the importing module, *nickname* is used to refer to the imported module object; *nickname* is confidential by default, but can be annotated as public.

Because importing a module creates a module object, public attributes of an imported module are accessed by requesting a method on the module's nickname. Confidential attributes are not visible to the importing module.

Example

```
import "list" as list is public
import "sparseMatrix" as matrix
```

Grammar

```
Import ::= "import" StringLiteral "as" Identifier Annotations
```

11.1.2 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 loads and initialises all transitively imported modules in depth-first order, thus executing the "main" module *last*, after all its dependencies have been 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
```

11.2 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 to nest the module that uses the dialect inside an enclosing scope that contains the public definitions of the dialect. 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 intrinsic type declarations, classes, traits, control structures, and even the graceObject trait that defines the default methods.

Modules that do not declare a **dialect** are treated as being written in the dialect *standard*. If a module really wishes to use no dialect, it should specify **dialect** "none".

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 might declare an unless(_)do(_) control structure that is like if, but backwards.

bcpl.grace module:

```
method do (block: Function0) 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)
```

Grammar

```
Dialect ::= "dialect" StringLiteral
```

11.3 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 that defines 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. Confidential names are not visible.

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). These rules allow dialects to import modules, and to be written in other 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. Garbage collection 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 specified.

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 extended BNF defines the context-free syntax of Grace. Productions are arranged in alphabetical order.

- A star ∗ indicates zero or more repetitions of the previous item,
- a plus + indicates one or more repetitions, and
- a question mark ? indicates that the previous item is optional.
- Parenthesis (and) group terminals and non-terminals.
- Terminal symbols are enclosed in "quotes"; the following additional terminals are in <angle brackets>:
  - <id> is an identifier: a sequence of letters, digits, single quotes ' and underscores, starting with a letter, as described in the Section on Identifiers
  - <newline> is a line break, as described in the Section on newlines
  - <dquote> is a double-quote character "
  - <dot> is a full stop (also known as a period)
  - <operator> is a sequence of operator characters
  - <decimalNumeral>, <baseExponentNumeral>, and <explicitRadixNumeral> are described in the Section on Numbers
  - <stringSegment> is a sequence of characters that does not include an unescaped ", newline, or {; it may contain the string escapes.
  - <uninterpretedString> is a sequence of *any* characters except ❯, the closing guillemet quotation mark; see the Section on Uninterpreted Strings
  - <typeRelation> is one of <:, :>,<∗, or ∗>

```
AliasClause ::= "alias" MethodHeader "=" MethodHeader

AnnotationArgList ::= "(" Expression  ( "," Expression )+  ")"
    | Numeral
    | String
    | SequenceConstructor
    | SpecialTerm
    | "(" Expression ")"
```

```
AnnotationLabel ::= <id>
    | <id> TypeArguments AnnotationArgList  ( <id> AnnotationArgList )*
    | <id> TypeArguments
    | <id> AnnotationArgList  ( <id> AnnotationArgList )*

Annotations ::= Empty
    | "is" AnnotationLabel  ( "," AnnotationLabel )*

ArgumentList ::= DelimitedTerm
    | "(" Expression  ( "," Expression )+  ")"

Assignment ::= Identifier ":=" Expression
    | AssignmentRequest

AssignmentMethodHeader ::= Identifier ":=" TypeParameterList SingleMethodParameter

AssignmentRequest ::= Term <dot> <id> ":=" Expression
    | "self" <dot> <id> ":=" Expression
    | ( "outer" <dot> )+  <id> ":=" Expression

BinaryMethodHeader ::= <operator> TypeParameterList SingleMethodParameter

BinaryRequest ::= Factor  ( <operator> TypeArguments?  Factor )+

Block ::= "{" BlockParameterList "→" Ss?  ( Statement  ( Ss Statement )*  )? "}"
    | "{"  ( Statement  ( Ss Statement )*  )? "}"

BlockParameter ::= Identifier PatternOption
    | NonIdExpression

BlockParameterList ::= BlockParameter  ( "," BlockParameter )*

Boolean ::= "true"
    | "false"

ClassDeclaration ::= "class" MethodHeader ReturnTypeOption Annotations ObjectBody?

Declaration ::= VarDeclaration
    | DefDeclaration

DefDeclaration ::= "def" Identifier TypeOption Annotations  ( "=" Expression )?

DelimitedTerm ::= Numeral
    | Block
    | String
    | SequenceConstructor
    | SpecialTerm
    | "(" Expression ")"

Dialect ::= "dialect" StringLiteral

DottedRequest ::= Term <dot> RequestPart

Ellipsis ::= "..."

Empty ::=
```

```
ExcludeClause ::= "exclude" MethodHeader

Expression ::= BinaryRequest
    | Factor

Factor ::= Term
    | ObjectConstructor
    | UnaryRequest

Identifier ::= <id>

ImplicitRequest ::= RequestPartsWithArguments

Import ::= "import" StringLiteral "as" Identifier Annotations

InheritStatement ::= "inherit" Expression  ( ReuseModifier )*

InterfaceLiteral ::= "interface" "{" "}"
    | "interface" "{" Signature  ( Ss Signature )*  "}"

MethodBody ::= "{"  ( Statement  ( Ss Statement )*  )? "}"

MethodDeclaration ::= "once"? "method" MethodHeader ReturnTypeOption Annotations MethodBody?

MethodHeader ::= AssignmentMethodHeader
    | ParameterizedMethodHeader
    | ParameterlessMethodHeader
    | BinaryMethodHeader
    | UnaryMethodHeader

MethodParameter ::= Identifier TypeOption

MethodParameterList ::= "(" MethodParameter  ( "," MethodParameter )*  ")"

Module ::= Ss?  ( Pragma Ss )*  ( Dialect Ss )?  ( ObjectItem Ss )*  ObjectItem?

NonIdExpression ::= BinaryRequest
    | NonIdFactor

NonIdFactor ::= NonIdTerm
    | ObjectConstructor
    | UnaryRequest

NonIdTerm ::= DelimitedTerm
    | InterfaceLiteral
    | UnknownType
    | SelfType
    | Request
    | Ellipsis

Numeral ::= <decimalNumeral>
    | <baseExponentNumeral>
    | <explicitRadixNumeral>

ObjectBody ::= "{"  ( ObjectItem  ( Ss ObjectItem )*  )? "}"
```

```
ObjectConstructor ::= "object" Annotations ObjectBody

ObjectItem ::= Statement
    | MethodDeclaration
    | TypeDeclaration
    | ClassDeclaration
    | TraitDeclaration
    | UseStatement
    | InheritStatement

Outer ::= "outer"
    | ( "outer" <dot> )+  "outer"

OuterRequest ::= ( "outer" <dot> )+  RequestPart

ParameterizedMethodHeader ::= <id> TypeParameterList MethodParameterList  ( <id> MethodParameterList )*

ParameterlessMethodHeader ::= <id> TypeParameterList

PatternOption ::= Empty
    | ":" Expression

Pragma ::= "#pragma" <id>

Request ::= ImplicitRequest
    | SelfRequest
    | OuterRequest
    | DottedRequest

RequestPart ::= RequestPartNoArguments
    | RequestPartsWithArguments

RequestPartNoArguments ::= <id>

RequestPartsWithArguments ::= <id> TypeArguments ArgumentList  ( <id> ArgumentList )*
    | <id> TypeArguments
    | <id> ArgumentList  ( <id> ArgumentList )*

Return ::= "return" Expression?

ReturnTypeOption ::= Empty
    | "→" TypeExpression

ReuseModifier ::= ExcludeClause
    | AliasClause

Self ::= "self"

SelfRequest ::= "self" <dot> RequestPart

SelfType ::= "Self"

SequenceConstructor ::= "[" "]"
    | "[" Expression  ( "," Expression )*  "]"

Signature ::= MethodHeader ReturnTypeOption
    | Ellipsis
```

```
SingleMethodParameter ::= "(" MethodParameter ")"

SpecialTerm ::= Self
    | Outer
    | Boolean

Ss ::= ";"
    | <newline>
    | Ss  ( ";" | <newline> )

Statement ::= Expression
    | Declaration
    | Assignment
    | Return
    | Import
    | <error>

String ::= StringLiteral
    | StringConstructor
    | UninterpretedString

StringConstructor ::= <dquote> <stringSegment>?   (  "{" Expression  "}" <stringSegment>?  )+   <dquote>

StringLiteral ::= <dquote> <stringSegment>?   <dquote>

Term ::= NonIdTerm
    | Identifier

TraitDeclaration ::= "trait" MethodHeader ReturnTypeOption Annotations ObjectBody?

Type ::= UnknownType
    | SelfType
    | InterfaceLiteral
    | Identifier TypeArguments?
    | Type <dot> RequestPartNoArguments TypeArguments?
    | "(" TypeExpression ")"

TypeArguments ::= "⟦" Type  ( "," Type )*  "⟧"

TypeDeclaration ::= "type" Identifier TypeParameterList Annotations "=" TypeExpression

TypeExpression ::= Type  ( <typeOperator> TypeArguments?  Type )*

TypeOption ::= Empty
    | ":" TypeExpression

TypeParameter ::= Identifier

TypeParameterList ::= Empty
    | "⟦" TypeParameter  ( "," TypeParameter )*  Where "⟧"

UnaryMethodHeader ::= "prefix" <operator> TypeParameterList

UnaryRequest ::= <operator> TypeArguments?  Term

UninterpretedString ::= "‹" <uninterpretedString>?   "›"
```

```
UnknownType ::= "Unknown"

UseStatement ::= "use" Expression ( ReuseModifier )*

VarDeclaration ::= "var" Identifier TypeOption Annotations ( ":=" Expression )?

Where ::= Empty
    | "where" WhereCondition ( "," WhereCondition )*

WhereCondition ::= <id> <typeRelation> Type
```