The Grace Standard Dialect
Draft Specification Version 0.7.9

Andrew P. Black Kim B. Bruce James Noble

Contents

1 Introduction

This is a specification of the *standard* dialect of Grace. Grace programs run in this dialect unless they nominate a different dialect using a **dialect** statement. The *standard* dialect provides a range of methods, objects and types for general purpose programming.

This specification is incomplete, and everything is subject to change.

2 Control Structures

2.1 Conditional

Standard Grace includes a conventional if. . . then . . . else conditional, as illustrated here:

```
if (background.isDark) then {
    fontColor := white
} else {
    fontColor := black
}
```

Note that the condition following the if must answer a boolean value, and is enclosed in parentheses. The blocks of code following then and else are evaluated only when necessary, and are enclosed in braces. The conditional can also be used as an expression:

```
fontColor := if (background.isDark) then { white } else { black }
```

The above two examples are equivalent. Remember that layout matters; the whole conditional can be written on one line if it fits, but otherwise should be broken across lines as shown.

The else part of the conditional statement is optional; it can be omitted if there is nothing to do. More conditions can be added using elseif clauses:

```
if (background.darkness > 0.7) then {
    fontColor := white
} elseif {background.darkness < 0.3} then {
    fontColor := black
} else {
    fontColor := red
}
```

Note that a condition following elseif is evaluated only when none of the previous conditions is true. To make this conditional evaluation possible, all conditions after the first must be enclosed in *braces*.

Grace also includes a multi-way match. . . case statement.

2.2 Bounded Loops

The simplest loop is a *repeat–times loop*, which repeats a block of code a pre-determined number of times:

```
repeat 4 times {
    print "hello"
}
```

The first argument (which must be parenthesized if it is an expression) must evaluate to a number n. The body of the loop, which follows times, must be a zero-parameter block. The effect of the *repeat–times* loop is to execute the block n.ceiling times.

The *for–do* and *for–and–do* loops are governed by collections. They execute a block of code repeatedly, depending on the elements of the collection, and are described in the Section on Iteration and **for** loops.

2.3 Unbounded Loops

Unbounded loops execute a block of code repeatedly, so long as some condition is satisfied. They terminate when the condition becomes false. They are useful when the number of executions needed can't be specified in advance. There are two variants: *while–do loops* test the condition *before* executing the loop body, and *do–while* loops test the condition *after* executing the loop body.

Here is an example of a *while–do* loop:

```
while { (x − (guess∗guess)).abs > epsilon } do {
    guess := (guess + (x/guess))/2
}
```

So long as the condition (the block following while) is true, the body of the loop (the block following do) will be executed. Notice that if the condition is false initially, the body of the *while loop* will never be executed.

Contrast this with the *do–while* loop:

```
do {
    guess := (guess + (x/guess))/2
} while { (x − (guess∗guess)).abs > epsilon }
```

Here, the body of the loop is executed *first*, and then the condition is checked. If it is false, the loop terminates; otherwise, the body is executed again. Hence, the body of a *do–while* loop is *always* executed at least once.

Notice that condition must be a *block*—usually a literal block, surrounded by braces. This is because the condition must be re-evaluated for each execution of the loop body. If it were a boolean, then it would be evaluated once, before the start of the loop, and the loop would either execute zero times, or infinitely! Notice also that the number of times that a *while* or *do* loop will execute cannot usually be determined in advance. This is what we mean by "unbounded loop". The number of times may even be infinite—a common coding error for beginners.

2.4 Match Case

Single-parameter blocks and pattern objects can be conveniently used in the match(_)case(_)... family of methods to support multiway branching.

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

The first two blocks use numbers as patterns; the first is short for {_ : 0 −> 0 }, that is, a block with parameter _ and pattern 0. The match(_)case(_)... statement may have one or many case branches, of which just one should match. The branch labelled else is optional, but when present must come last.

If match(_)case(_)... does not find a match, it executes the else block, if there is one; otherwise it raises raises MatchError. If it finds *multiple* matches, it also raises MatchError.

2.5 ValueOf

Grace's valueOf allows a block with local declarations and a statement list where an expression is required. valueOf takes a block as argument, evaluates it, and returns the result.

```
def constant = valueOf {
    def local1 = ...
    def local2 = ...
    complicated expression involving locals
}
```

3 Built-in Types

Grace supports built-in objects with types Object, Number, Boolean, and String.

3.1 Object and EqualityObject

All Grace objects understand the methods in type Object. These methods will often be omitted when other types are described.

```
type Object = interface {
    asString −> String
        // a string describing self

    asDebugString −> String
        // a string describing the internals of self
}
```

Objects that support an equality test have additional methods

```
type EqualityObject = Object & interface {
    ==(other:Object) −> Boolean
        // true if I consider other to be equal to me
    ≠(other:Object) −> Boolean
        // the inverse of ==
    hash −> Number
        // returns a hash code such that (a == b) implies (a.hash == b.hash)
    ::(v:Object) −> Binding
        // answers a binding with self as the key and v as the value
}
```

3.2 Numbers

Number describes all numeric values in *minigrace*, including integers and numbers with decimal fractions. (Thus, *minigrace* Numbers are what some other languages call floating point numbers, floats or double-precision). Numbers are represented with a precision of approximately 51 bits. Number constants include π and infinity, the latter being larger than any finite number, as well as conventional numerals like 27, 2.5, and 7e3 (the latter meaning 7 ∗ 10^3).

```
type Number = EqualityObject & Pattern & interface {

    + (other: Number) −> Number
    // sum of self and other

    − (other: Number) −> Number
    // difference of self and other

    ∗ (other: Number) −> Number
    // product of self and other
```

/ (other: Number) —> Number
// quotient of self divided by other (in general, a fraction).

% (other: Number) —> Number
// remainder r after integer division of self by other: $0 \leq r < self$; *see also* ÷

÷ (other: Number) —> Number
// quotient q of self after integer division by other: $self = (other * q) + r$, *where* $r = self \% other$

.. (last: Number) —> Sequence
// the Sequence of numbers from self to last, so 2..4 contains 2, 3, and 4

downTo(last:Number) —> Sequence
// the Sequence of numbers from self down to last, so 2.downTo 0 contains 2, 1 and 0.

< (other: Number) —> Boolean
// true iff self is less than other

<= (other: Number) —> Boolean
// true iff self is less than or equal to other

> (other: Number) —> Boolean
// true iff self is greater than other

>= (other: Number) —> Boolean
// true iff self is greater than or equal to other

prefix− —> Number
// negation of self

compare (other:Number) —> Number
// a three−way comparison: −1 if (self < other), 0 if (self == other), and +1 if (self > other).
// This is useful when writing a comparison function for sortBy

inBase (base:Number) —> String
// a string representing self as a base number (e.g., 5.inBase 2 = "101")

asString —> String
// returns a string representing self rounded to six decimal places

asDebugString —> String
// returns a string representing self with all available precision

asStringDecimals(d) —> String
// returns a string representing self with exactly d decimal digits

isInteger —> Boolean
// true if number is an integer, i.e., a whole number with no fractional part

truncated —> Number
// number obtained by throwing away self's fractional part

rounded —> Number
// whole number closest to self

floor —> Number

// largest whole number less than or equal to self

ceiling –> Number
// smallest whole number greater than or equal to self

abs –> Number
// the absolute value of self

sgn –> Number
// the signum function: 0 when self == 0, -1 when self $<$ 0, and $+1$ when self $>$ 0

isNaN –> Boolean
// true if this Number is not a number, i.e., if it is NaN. For example, 0/0 returns NaN

isEven –> Boolean
// true if this number is even

isOdd –> Boolean
// true if this number is odd

sin –> Number
// trigonometric sine (self in radians)

cos –> Number
// cosine (self in radians)

tan –> Number
// tangent (self in radians)

asin –> Number
// arcsine of self (result in radians)

acos –> Number
// arccosine of self (result in radians)

atan –> Number
// arctangent of self (result in radians)

lg –> Number
// log base 2 of self

ln –> Number
// the natural log of self

exp –> Number
// e raised to the power of self

log10 –> Number
// log base 10 of self

prefix $>$ –> Pattern
// a pattern that matches all numbers $>$ self

prefix $\geq$ –> Pattern
// a pattern that matches all numbers $\geq$ self

prefix $<$ –> Pattern

```
    // a pattern that matches all numbers < self

    prefix ≤ -> Pattern
    // a pattern that matches all numbers ≤ self
}
```

3.3 Strings

String constructors are written surrounded by double quote characters. There are three commonly-used escape characters:

- \n means the newline character
- \\ means a single backslash character
- \" means a double quote character.

There are also escapes for a few other characters and for arbitrary Unicode codepoints; for more information, see the Grace language specification.

String constructors can also contain simple Grace expressions enclosed in braces, like this: "count = {count}." These are called string interpolations. The value of the interpolated expression is calculated, converted to a string (by requesting its asString method), and concatenated between the surrounding fragments of literal string. Thus, if the value of count is 7, the above example will evaluate to the string "count = 7."

Strings are immutable. Methods like replace(_)with(_) always return a new string; they never change the receiver.

```
type String = EqualityObject & Pattern & interface {
    * (n: Number) -> String
    // returns a string that contains n repetitions of self, so "Abc" * 3 = "AbcAbcAbc"

    ++(other: Object) -> String
    // returns a string that is the concatenation of self and other.asString

    < (other: String)
    // true if self precedes other lexicographically

    <= (other: String)
    // (self == other) || (self < other)

    == (other: Object)
    // true if other is a String and is equal to self

    != (other: Object)
    // !(self == other)

    > (other: String)
    // true if self follows other lexicographically

    >= (other: String)
    // (self == other) || (self > other)

    at(index: Number) -> String
    // returns the character in position index (as a string of size 1); index must be an integer in
    // the range 1..size
```

first –> String
// returns the first character of the string, as a String of size 1. String must not be empty

asDebugString –> String
// returns self enclosed in quotes, and with embedded special characters quoted. See also quoted.

asLower –> String
// returns a string like self, except that all letters are in lower case

asNumber –> Number
// attempts to parse self as a number; returns that number, or NaN if it can't.

asString –> String
// returns self, naturally.

asUpper –> String
// returns a string like self, except that all letters are in upper case

do(action:Function1⟦String, Done⟧) –> Done
// applies action to each character of self.

capitalized –> String
// returns a string like self, except that the initial letters of all words are in upper case

compare (other:String) –> Number
// a three−way comparison: −1 if (self < other), 0 if (self == other), and +1 if (self > other).
// This is useful when writing a comparison function for sortBy

contains (other:String) –> Boolean
// returns true if other is a substring of self

endsWith (possibleSuffix: String)
// true if self ends with possibleSuffix

filter (predicate: Function1⟦String,Boolean⟧) –> String
// returns the String containing those characters of self for which predicate returns true

fold⟦U⟧ (binaryFunction: Function2⟦U,String,U⟧) startingWith(initial: U) –> U
// performs a left fold of binaryFunction over self, starting with initial.
// For example, fold {a, b –> a + b.ord} startingWith 0 will compute the sum
// of the ords of the characters in self

hash –> Number
// the hash of self

indexOf (pattern:String) –> Number
// returns the leftmost index at which pattern appears in self, or 0 if it is not there.

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.

indexOf (pattern:String) startingAt (offset) –> Number
// like indexOf(pattern), but returns the smallest index ≥ offset, or 0 if pattern is not found.

indexOf⟦W⟧ (pattern:String) startingAt(offset) ifAbsent (action:Function0⟦W⟧) –> Number | W
// like the above, except that it returns the result of applying action if there is no such index.

```
indices —> Sequence⟦Number⟧
keys —> Sequence⟦Number⟧
// an object representing the range of indices of self (1..self.size).

isEmpty —> Boolean
// true if self is the empty string

iterator —> Iterator⟦String⟧
// an iterator over the characters of self

keysAndValuesDo(action:Function2⟦Number, String, Done⟧) —> Done
// applies action to two arguments for each character in self: the key (index) of the character,
// and the character itself.

lastIndexOf (sub:String) —> Number
// returns the rightmost index at which sub appears in self, or 0 if it is not there.

lastIndexOf (sub:String) startingAt (offset) —>  Number
// like the above, except that it returns the rightmost index ≤ offset.

lastIndexOf⟦W⟧ (sub:String) ifAbsent (absent:Function0⟦W⟧) —> Number | W
// returns the rightmost index at which sub appears in self; applies absent if it is not there.

lastIndexOf⟦W⟧ (sub:String)
   startingAt (offset)
   ifAbsent (action:Function0⟦W⟧) —>  Number | W
// like the above, except that it returns the rightmost index ≤ offset.

map⟦U⟧ (function:Function1⟦String,U⟧) —> Collection⟦U⟧
// returns a Collection containing the results of successive applications of function to the
// individual characters of self. Note that the result is not a String, even if type U happens to be String.
// If a String is desired, use fold (_) startingWith "" with a function that concatenates.

ord —> Number
// a numeric representation of the first character of self, or NaN if self is empty.

replace (pattern:String) with (new:String) —> String
// a string like self, but with all occurrences of pattern replaced by new

size —> Number
// returns the size of self, i.e., the number of characters it contains.

split(splitter:String) —> List⟦String⟧
// answers a list of substrings of self, split before and after each
// occurrence of splitter in self.  If self is empty, the result list
// will also be empty; otherwise, if self does not contain splitter,
// the result list will be of size 1.

startsWith (possiblePrefix:String) —> Boolean
// true when possiblePrefix is a prefix of self

startsWithDigit —> Boolean
// true if the first character of self is a (Unicode) digit.

startsWithLetter —> Boolean
// true if the first character of self is a (Unicode) letter
```

```
    startsWithPeriod −> Boolean
    // true if the first character of self is a period

    startsWithSpace −> Boolean
    // true if the first character of self is a (Unicode) space.

    substringFrom (start:Number) size (max:Number) −> String
    // returns the substring of self starting at index start and of length max characters,
    // or extending to the end of self if that is less than max.  If start = self.size + 1 or
    // stop < start, the empty string is returned.   If start is outside the range
    // 1..self.size+1, BoundsError is raised.

    substringFrom (start:Number) to (stop:Number) −> String
    // returns the substring of self starting at index start and extending
    // either to the end of self, or to stop.    If start = self.size + 1, or
    // stop < start, the empty string is returned.   If start is outside the range
    // 1..self.size+1, BoundsError is raised.

    substringFrom (start:Number) −> String
    // returns the substring of self starting at index start and extending
    // to the end of self.    If start = self.size + 1, the empty string is returned.
    // If start is outside the range 1..self.size+1, BoundsError is raised.

    trim −> String
    // a string like self except that leading and trailing spaces are omitted.

    quoted −> String
    // returns a quoted version of self, with internal characters like " and \ and newline escaped,
    // but without surrounding quotes.  See also asDebugString

    >>(target:Sink⟦String⟧) −> Collection
    // returns target << self

    <<(source:Enumerable⟦String⟧) −> String
    // returns a string containing me, followed in order by the elements of source.
}
```

3.4 Booleans

The Boolean literals are **true** and **false**.

```
type Boolean = EqualityObject & interface {

    not −> Boolean
    prefix ! −> Boolean
    // the negation of self

    && (other: PredicateOrBoolean) −> Boolean
    // returns true when self and other are both true

    || (other: PredicateOrBoolean) −> Boolean
    // returns true when either self or other (or both) are true

    == (other:Object) −> Boolean
    // returns true of other is the same Boolean as self
    ≠ (other:Object) −> 
    // if other is a boolean, returns the exclusive OR of self and other; otherwise returns false
}
```

The condition in an if statement, and the argument to the operators && and ||, can be either a Boolean, or a zero-parameter block that returns a Boolean. This means that && and || can be used as "short circuit" operators, also known as "non-commutative", operators: they will evaluate their argument only if necessary.

```
type PredicateOrBoolean = Predicate0 | Boolean
```

3.5 Blocks and Functions

Blocks implement anonymous functions that take zero or more arguments and return one result. A family of function types describe block objects:

```
type Function0⟦R⟧ = interface {
    apply −> R
}
type Function1⟦T, R⟧ = interface {
    apply(a:T) −> R
    matches(a:Object) −> R       // answers true if a <: T
}
type Function2⟦T1, T2, R⟧ = interface {
    apply(a1:T1, a2:T2) −> R
    matches(a1:Object, a2:Object) → Boolean
        // answers true if a1 <: T1 and a2 <: T2
}
type Function3⟦T1, T2, T3, ResultT⟧  = Object & interface {
    apply(a1:T1, a2:T2, a3:T3) → ResultT
    matches(a1:Object, a2:Object, a3:Object) → Boolean
        // answers true if a1 <: T1 and a2 <: T2 and a3 :< T3
}

// Predictates are functions that return a Boolean
type Predicate0 = Function0⟦Boolean⟧
    // Function with no arguments returning Boolean
type Predicate1⟦T1⟧ = Function1⟦T1, Boolean⟧
    // Function with 1 argument of type T1, returning Boolean
type Predicate2⟦T1, T2⟧ = Function2⟦T1, T2, Boolean⟧
    // Function with 2 arguments of types T1 and T2, returning Boolean
type Predicate3⟦T1, T2, T3⟧ = Function3⟦T1, T2, T3, Boolean⟧
    // Function with 3 arguments of types T1, T2, and T3, returning Boolean
```

Block objects are normally created using Grace's block syntax { a:P, b:Q −> ... }.

3.6 Patterns

Any object that implements the interface

```
type Pattern = Object & interface {
    matches(value:Object) → Boolean // true if this pattern matches value
    & (other:Pattern) → Pattern      // pattern AND
    | (other:Pattern) → Pattern      // pattern OR
    prefix ¬ → Pattern               // the negation of this pattern
    isType → Boolean                 // is this pattern also a type?
}
```

is a pattern. Strings and Numbers are patterns; they match strings and numbers to which they are equal. A type `T` is a pattern that matches objects that have type `T`. Blocks are patterns: if the block's parameteter is annotated with a type or pattern `P`, then the block matches objects that are matched by `P`. (This also works for blocks with multiple parameters.)

Patterns can be combined using the operators `&` (AND), | (OR) and ¬ (NOT) to construct compound patterns.

3.7 Points

Points can be thought of as locations in the cartesian plane, or as 2-dimensional vectors from the origin to a specified location. Points are created from Numbers using the `@` infix operator. Thus, `3 @ 4` represents the point with coordinates (3, 4).

```
type Point =  EqualityObject && interface {

    x −> Number
    // the x−coordinates of self

    y −> Number
    // the y−coordinate of self

    == (other:Object) −> Boolean
    // true if other is a Point with the same x and y coordinates as self.

    + (other:Point) −> Point
    // the Point that is the vector sum of self and other, i.e. (self.x+other.x) @ (self.y+other.y)

    − (other:Point) −> Point
    // the Point that is the vector difference of self and other, i.e. (self.x−other.x) @ (self.y−other.y)

    prefix − −> Point
    // the point that is the negation of self

    ∗ (factor:Number) −> Point
    // this point scaled by factor, i.e., (self.x∗factor) @ (self.y∗factor)

    / (factor:Number) −> Point
    // this point scaled by 1/factor, i.e., (self.x/factor) @ (self.y/factor)

    length −> Number
    // distance from self to the origin

    distanceTo(other:Point) −> Number
    // distance from self to other

    dot (other:Point) −> Number
    · (other:Point) −> Number
    // dot product of self and other

    norm −> Point
    // the unit vector (vector of length 1) in same direction as self
}
```

3.8 Bindings

A binding is an immutable pair comprising a key and a value. Bindings are created with the infix :: operator, as in k::v, or by requesting binding.key(k) value(v).

```
type Binding⟦K, T⟧ = interface {
    key −> K
    // returns the key
    value −> T
    // returns the value
}
```

The type K of keys must be an EqualityObject

4 Collection objects

The objects described in this section are made available to all Grace programs that are written in the *standard* dialect. As is natural for collections, the types are parameterized by the types of the elements of the collection. Type arguments are enclosed in ⟦ and ⟧ brackets. This enables us to distinguish, for example, between Set⟦Number⟧ and Set⟦String⟧. In Grace programs, type arguments and their brackets can be omitted; this is equivalent to using Unknown as the argument, which says that the programmer either does not know, or does not care to state, the type.

4.1 Common Abstractions

The major kinds of collection are sequence, list, set and dictionary. Although these collection objects differ in their details, they share many common methods, which are defined in a hierarchy of types, each extending the one above it in the hierarchy. The simplest is the type Collection⟦T⟧, which captures the idea of a (potentially unordered, and possibly unbounded) collection of *elements*, each of type T, over which a client can iterate:

```
type Collection⟦T⟧ = interface {
    iterator −> Iterator⟦T⟧
    // Returns an iterator over my elements.  It is an error to modify self while iterating over it.
    // Note: all other methods can be defined using iterator. Iterating over a dictionary
    // yields its values.

    isEmpty −> Boolean
    // True if self has no elements

    size −> Number
    // returns the number of elements in self; raises SizeUnknown if size is not known.

    sizeIfUnknown(action: Function0⟦Number⟧) −> Number
    // returns the number of elements in self, or the result of evaluating action if size is not known

    first −> T
    // The first element of self; raises BoundsError if there is none.
    // If self is unordered, then first returns an arbitrary element.

    do(action: Function1⟦T,Unknown⟧) −> Done
    //  Applies action to each element of self.

    do(action:Function1⟦T, Unknown⟧) separatedBy(sep:Function0⟦Unknown⟧) −> Done
    // applies action to each element of self, and applies sep (to no arguments) in between.
    map⟦R⟧(unaryFunction:Function1⟦T, R⟧) −> Collection⟦T⟧
    // returns a new collection whose elements are obtained by applying unaryFunction to
    // each element of self. If self is ordered, then the result is ordered.

    fold⟦R⟧(binaryFunction:Function2⟦R, T, R⟧) startingWith(initial:R) −> R
    // folds binaryFunction over self, starting with initial. If self is ordered, this is
    // the left fold. For example, fold {a, b −> a + b} startingWith 0
    // will compute the sum, and fold {a, b −> a * b} startingWith 1 the product.

    filter(condition:Function1⟦T, Boolean⟧) −> Collection⟦T⟧
    // returns a new collection containing only those elements of self for which
    // condition holds. The result is ordered if self is ordered.

    anySatisfy(condition:Function1⟦T, Boolean⟧) −> Boolean
    // returns true if self contains an element x for which conditon.apply(x)) holds.
    // Otherwise (including the case when self is empty), returns false.

    allSatisfy(condition:Function1⟦T, Boolean⟧) −> Boolean
    // returns false if conditon.apply(x)) is false for any element x of self.
    // Otherwise (including the case when self is empty), returns true.

    ++(other:Enumerable⟦T⟧) −> Collection⟦T⟧
    // returns a new object whose elements include those of self and those of other.

    >>(target:Sink⟦T⟧) −> Collection⟦T⟧
    // sends my values into target

    <<(source:Enumerable⟦T⟧) −> Collection⟦T⟧
    // appends the values in source to my values.

}
```

4.2 Creating Collections

Collections can be created using objects with the CollectionFactory interface:

```
type CollectionFactory⟦T⟧ = interface {
    empty −> Collection⟦T⟧
        // an empty collection
    with(element:T) −> Collection⟦T⟧
        // a collection containing a single element
    withAll(elements:Collection⟦T⟧) −> Collection⟦T⟧
        // a collection containing elements
    <<(source:Collection⟦T⟧) −> Collection⟦T⟧
        // identical to withAll(_)
}
```

The classes list, set, and sequence all support this interface; dictionary supports a very similar interface DictionaryFactory. In addition, the *methods* list⟦T⟧(_), set⟦T⟧(_) and sequence⟦T⟧(_) are defined in *standard*, and are equivalent to list⟦T⟧.withAll(_), set⟦T⟧.withAll(_) and sequence⟦T⟧.withAll(_).

4.3 Enumerables

Additional methods are available in the type Enumerable; an Enumerable is a Collection, that allows its elements to be *enumerated* one by one, in order, using a computational process. The key difference between a Collection and an Enumerable is that Enumerables have a natural order, so lists are Enumerable, whereas sets are just Collections.

There is no requirment that the elements of an Enumerable are stored explicitly. For this reason, operations that require access to all of the elements at the same time, like sorting, are not supported directly. Instead, Enumerables can be converted to other collections that store their elements.

```
type Enumerable⟦T⟧ = Collection⟦T⟧ & interface {

    values −> Enumerable⟦T⟧
    // an enumeration of my values: the elements in the case of sequence or list,
    // the values in the case of a dictionary.

    keysAndValuesDo (action:Function2⟦Number, T, Object⟧) −> Done
    // applies action, in order, to each of my keys and the corresponding element.

    sorted −> List⟦T⟧
    // returns a new list containing all of my elements, but sorted by their < and == operations.

    sortedBy(sortBlock:Function2⟦T, T, Number⟧) −> List⟦T⟧
    // returns a new list containing all of my elements, but sorted according to the ordering
    // established by sortBlock, which should return −1 if its first argument is less than its
    // second argument, 0 if they are equal, and +1 otherwise.
}
```

4.4 Sequences

The type Sequence⟦T⟧ describes sequences of values of type T. Sequence objects are immutable; they can be constructed explicitly, using a sequence constructor such as [1, 3, 5, 7], or as ranges such as 1..10 or 10.downto 1, or using the sequence factory.

```
type Sequence⟦T⟧ = EqualityObject & Sequenceable⟦T⟧

type Sequenceable⟦T⟧ = Enumerable⟦T⟧ & interface {

    at(n:Number) −> T
    // returns my element at index n (starting from 1), if n is integral and l ≤ n ≤  size

    at⟦W⟧(n:Number) ifAbsent(action:Function0⟦W⟧) −> T | W
    // returns my element at index n (starting from 1), if n is integral and l ≤ n ≤  size.
    // Otherwise, executes action and returns its result

    first −> T
    // returns my first element

    second −> T
    // returns my second element

    third −> T
    // returns my third element

    fourth −> T
    // returns my fourth element

    fifth −> T
    // returns my fifth element

    last −> T
    // returns my last element

    indices -> Sequence⟦Number⟧
    // returns the sequence of my indices, 1..size

    keys -> Sequence⟦Number⟧
    // same as indices; the name keys is for compatibility with dictionaries.

    indexOf(sought:T)  -> Number
    // returns the index of my first element v such that v == sought.  Raises NoSuchObject
    // if there is none.

    indexOf⟦W⟧(sought:T) ifAbsent(action:Function0⟦W⟧)  -> Number | W
    // returns the index of my first element v such that v == sought.  Performs action if
    // there is no such element.

    reversed -> Sequence⟦T⟧
    // returns a Sequence containing my values, but in the reverse order.

    contains(sought:T) -> Boolean
    // returns true if I contain an element v such that v == sought
}
```

Because a Sequence is imutable, its == and hash methods are stable, and it can be used as a key in a Dictionary. This is not true of a List or a Set.

4.5 Sequence Constructors

The Grace language uses brackets as a syntax for constructing Sequence objects. [2, 3, 4] is a sequence containing the three numbers 2, 3 and 4; [ ] constructs the empty sequence.

Although sequences are immutable, they can be used for initializing mutable collections, as in [2, 3, 4] >> list, which creates a list, or set⟦String⟧ << ["red", "green", "yellow"], which creates a set.

4.6 Ranges

Ranges are sequences of consecutive integers. They behave exactly like other sequences, but are stored compactly. Ranges are created by two methods on the range class:

```
    range.from(lower:Number) to(upper:Number)
    // The sequence of integers from lower to upper, inclusive.  If lower = upper,
    // the range contains a single value.  If lower > upper, the range is empty.
    // It is an error for lower or upper not to be an integer.

    range.from(upper:Number) downTo(lower:Number)
    // The sequence from upper to lower, inclusive.  If upper = lower,
    // the range contains a single value. If upper < lower, the range is empty.
    // It is an error for lower or upper not to be an integer.
```

The .. operation on Numbers can also be used to create ranges. Thus, 3..9 is the same as range.from 3 to 9. Note that 9..9 is a range containing just one element, and 9..8 and 9..3 are empty ranges. Downward ranges can also be constructed using the downTo(_) method on Numbers, so 9.downTo 3 is the same as range.from 9 downTo 3.

4.7 Lists

The type List⟦T⟧ describes objects that are mutable lists of elements that each have type T. List objects can be constructed by requesting the list method, as in list⟦T⟧.withAll [ ], list⟦T⟧.withAll [a, b, c], or list.withAll (existingCollection).

```
type List⟦T⟧ = Sequenceable⟦T⟧ & interface {

    at(n: Number) put(new:T) –> List⟦T⟧
    // updates self so that my element at index n is new.  Returns self.
    // Requires 1 ≤ n ≤ size+1; when n = size+1, equivalent to addLast(new).


    add(new:T) –> List⟦T⟧
    addLast(new:T) –> List⟦T⟧
    // adds new to end of self.  (The first form can be also be applied to sets, which are not Indexable.)

    addFirst(new:T) –> List⟦T⟧
    // adds new as the first element(s) of self.  Changes the index of all of the existing elements.

    addAllFirst(news: Collection⟦T⟧) –> List⟦T⟧
    // adds news as the first elements of self.  Changes the index of all of the existing elements.

    removeFirst –> T
    // removes and returns first element of self.  Changes the index of the remaining elements.

    removeLast –> T
    // removes and returns last element of self.

    removeAt(n:Number) –> T
    // removes and returns n^th element of self

    remove(element:T) –> List⟦T⟧
    // removes element from self.  Raises NoSuchObject if self.contains(element).not
    // Returns self

    remove(element:T) ifAbsent(action:Function0⟦Unknown⟧) –> List⟦T⟧
    // removes element from self; executes action if it is not contained in self.  Returns self

    removeAll(elements:Collection⟦T⟧) –> List⟦T⟧
    // removes elements from self. Raises NoSuchObject exception if any  of
    // them is not contained in self. Returns self

    removeAll(elements:Collection⟦T⟧) ifAbsent(action:Function0⟦Unknown⟧) –> List⟦T⟧
    // removes elements from self; executes action if any of them is not contained in self. Returns self

    insert(element:T) at(n:Number) –>  List⟦T⟧
    // inserts element into self at index n; any element with index i > n is
    // moved so that it has index i + 1.  Returns self, that is, the modified list.

    clear –> List⟦T⟧
    // removes all the elements of self, leaving self empty.  Returns self

    ++ (other:List⟦T⟧) –> List⟦T⟧
    // returns a new list formed by concatenating self and other

    addAll(extension:Collection⟦T⟧) –> List⟦T⟧
    // extends self by appending the contents of extension; returns self.
    contains(sought:T) -> Boolean
    // returns true when sought is an element of self.

    == (other: Object) -> Boolean
    // returns true when other is a Sequence of the same size as self, containing elements that
    // are == to those in self, in the same order.

    sort -> List⟦T⟧
    // sorts self in place, using the < and == operations on elements.  Returns self.
    // Compare with sorted, which constructs a new list.

    sortBy(sortBlock:Function2⟦T, T, Number⟧) -> List⟦T⟧
    // sorts self according to the ordering determined by sortBlock, which should return −1 if its first
    // argument is less than its second argument, 0 if they are equal, and +1 otherwise.  Returns self.
    // Compare with sortedBy, which constructs a new list.

    copy -> List⟦T⟧
    // returns a list that is a (shallow) copy of self

    reverse -> List⟦T⟧
    // mutates self in−place so that its elements are in the reverse order.  Returns self.
    // Compare with reversed, which creates a new collection.
}
```

4.8 Sets

A Set is an unordered collections of elements, without duplicates. The == method on the elements is used to detect and eliminate duplicates; it must be symmetric.

```
type Set⟦T⟧ = Collection⟦T⟧ & interface {
    size -> Number
    // the number of elements in self.

    add(element:T) -> Set⟦T⟧
    // adds element to self.  Returns self.

    addAll(elements:Collection⟦T⟧) -> Set⟦T⟧
    // adds elements to self.  Returns self.

    remove(element: T) -> Set⟦T⟧
    // removes element from self.  It is an error if element is not present.   Returns self.

    remove(elements: T) ifAbsent(block: Function0⟦Done⟧) -> Set⟦T⟧
    // removes element from self.  Executes action if element is not present.   Returns self.

    removeAll(elems:Collection⟦T⟧)
    // removes elems from self.  Raises NoSuchObject if any of the elems is
    // not present.   Returns self.

    removeAll(elems:Collection⟦T⟧) ifAbsent(action:Function1⟦T, Done⟧) -> Set⟦T⟧
    // removes elems from self.  Executes action.apply(e) for each e in elems that is
    // not present.   Returns self.

    clear -> Set⟦T⟧
    // removes all the elements of self, leaving self empty.  Returns self
    contains(elem:T) -> Boolean
    // true if self contains elem

    find(predicate: Function1⟦T,Boolean⟧) ifNone(notFoundBlock: Function0⟦T⟧) -> T
    // returns an element of self for which predicate holds, or the result of
    // applying notFoundBlock if there is no such element.

    copy -> Set⟦T⟧
    // returns a copy of self

    ** (other:Set⟦T⟧) -> Set⟦T⟧
    // set intersection; returns a new set that is the intersection of self and other

    -- (other:Set⟦T⟧) -> Set⟦T⟧
    // set difference (relative complement); the result contains all of my elements that are
    // not also in other.

    ++ (other:Set⟦T⟧) -> Set⟦T⟧
    // set union; the result contains elements that were in self or in other (or in both).

    isSubset(s2: Set⟦T⟧) -> Boolean
    // true if I am a subset of s2

    isSuperset(s2: Collection⟦T⟧) -> Boolean
    // true if I contain all the elements of s2

    into(existing:Collection⟦T⟧) -> Collection⟦T⟧
    // adds my elements to existing, and returns existing.
}
```

4.9 Dictionary

The type Dictionary⟦K, T⟧ describes an object that is a mapping from *keys* of type K to *values* of type T.

4.9.1 Creating Dictionaries

Like sets and sequences, dictionary objects can be constructed using the class dictionary, but the argument to dictionary.withAll must be of type Collection⟦Binding⟧. This means that each element of the argument must have methods key and value. Bindings can be conveniently created using the infix :: operator, as in dictionary⟦K, T⟧.withAll [k::v, m::w, n::x, ...], or equivalently, [k::v, m::w, n::x, ...] >> dictionary⟦K, T⟧.

The object dictionary⟦K,T⟧ supports the DictionaryFactory interface:

```
type DictionaryFactory⟦K,T⟧ = interface {
    empty -> Dictionary⟦K,T⟧
    // an empty dictionary

    with(b:Binding⟦K,T⟧) -> Dictionary⟦K,T⟧
    // a dictionary containing single mapping from b.key to b.value

    withAll(bs:Binding⟦K,T⟧) -> Dictionary⟦K,T⟧
    // a dictionary containing a mapping for each binding in bs

    <<(bs:Binding⟦K,T⟧) -> Dictionary⟦K,T⟧
    // identical to withAll(_)
}
```

DictionaryFactory has the same four methods as CollectionFactory, but with different argument and result types.

4.9.2 Dictionary Methods

```
type Dictionary⟦K, T⟧ = Collection⟦T⟧ & interface {
    size –> Number
    // the number of key::value bindings in self

    at(key:K) put(value:T) –> Dictionary⟦K, T⟧
    // puts value at key; returns self

    at(k:K) –> T
    // returns my value at key k; raises NoSuchObject if there is none.

    at(k:K) ifAbsent(action:Function0⟦T⟧) –> T
    // returns my value at key k; returns the result of applying action if there is none.

    containsKey(k:K) –> Boolean
    // returns true if one of my keys == k

    contains(v:T) –> Boolean
    containsValue(v:T) –> Boolean
    // returns true if one of my values == v

    removeAllKeys(keys: Collection⟦K⟧) –> Self
    // removes all of the keys from self, along with the corresponding values.  Returns self.

    removeKey(key: K) –> Self
    // removes key from self, along with the corresponding value.  Returns self.

    removeAllValues(removals: Collection⟦T⟧) –> Self
    // removes from self all of the values in removals, along with the corresponding keys.
    // Returns self.

    removeValue(removal:T) –> Self
    // removes from self the value removal, along with the corresponding key.
    // Returns self.

    clear –> Dictionary⟦T⟧
    // removes all the elements of self, leaving self empty.  Returns self

    keys –> Collection⟦K⟧
    // returns my keys as a lazy sequence in arbitrary order

    values –> Collection⟦T⟧
    // returns my values as a lazy sequence in arbitrary order

    bindings –> Enumerable⟦ Binding⟦K, T⟧ ⟧
    // returns my bindings as a lazy sequence

    keysAndValuesDo(action:Procedure2⟦K, T⟧) –> Done
    // applies action, in arbitrary order, to each of my keys and the corresponding value.

    keysDo(action:Procedure1⟦K⟧) –> Done
    // applies action, in arbitrary order, to each of my keys.

    valuesDo(action:Procedure1⟦T⟧) –> Done
    do(action:Procedure1⟦T⟧) −> Done
    // applies action, in arbitrary order, to each of my values.

    copy −> Self
    // returns a new dictionary that is a (shallow) copy of self

    ++ (other:Dictionary⟦K, T⟧) −> Dictionary⟦K, T⟧
    // returns a new dictionary that merges the entries from self and other.
    // A value in other at key k overrides the value in self at key k.

    −− (other:Dictionary⟦K, T⟧) −> Dictionary⟦K, T⟧
    // returns a new dictionary that contains all of my entries except
    // for those whose keys are in other

    >> (target:Sink⟦Binding⟦K, T⟧ ⟧)
    // sends my bindings into target

    << (source:Collection⟦Binding⟦K, T⟧ ⟧)
    // adds the bindings in source to my bindings, overriding values with common keys.
    // Similar to ++(_), except that source is not a Dictionary.
}
```

4.10 Iteration and *for* loops

Collections implement the type Collection⟦T⟧, and thus implement the internal and external iterator patterns. These patterns provide for iteration through a collection of elements of type T, one element at a time. The method do(_) and its variant do(_)separatedBy(_) implement internal iterators, while the method iterator returns an external iterator object, with the following interface:

```
type Iterator⟦T⟧ = interface {
    next −> T
    // returns the next element of the collection over which I am the iterator; raises the Exhausted
    // exception if there are no more elements. Repeated request of this method will yield all of the
    // elements of the underlying collection, one at a time.

    hasNext −> Boolean
    // returns  true if there is at least one more element, i.e., if next will not raise `Exhausted`.
    // Once an iterator is exhausted (i.e., once hasNext returns false), it will remain exhausted.
}
```

Multiple iterators can exist on the same collection, for example, multiple iterator objects and multiple dos can be used to read through a file. Requesting next on one iterator advances its conceptual position, but does not affect other iterators over the same collection; nor does requesting do on a collection disturb any iterator objects. However, *it is an error to modify a collection object while iterating through it.* If you implement your own iterator, it is good practice to detect this error and raise ConcurrentModification.

for–do loops on Collection objects are provided by the *standard* dialect. The method for(_)do(_) takes two arguments, a collection and a one-parameter block body. It repeatedly applies body to the elements of collection. For example:

```
def fruits = sequence ["orange", "apple", "mango", "guava"]
for (fruits) do { each −>
    print(each)
}
```

The elements of the sequence fruits are bound in turn to the parameter each of the block that follows do, and the block is then executed. This continues until all of the elements of fruits have been supplied to the block, or the block terminates the surrounding method by executing a **return**.

for(_)do(_) is precisely equivalent to requesting the do method of the Collection, which is usually both faster and clearer:

```
    fruits.do { each ->
        print(each)
    }
```

A variant for(_)and(_)do(_) allows one to iterate through two collections in parallel, terminating when the smaller is exhausted:

```
    def result = list [ ]
    def xs = [1, 2, 3, 4, 5]
    def ys = ["one", "two", "three"]
    for (xs) and (ys) do { x, y ->
        result.add(x::y)
    }
```

After executing this code, result == [1::"one", 2::"two", 3::"three"].

The need for external iterators becomes apparent when it is necessary to iterate through two collections, but not precisely in parallel. For example, this method merges two sorted Collections into a sorted list:

```
  method merge ⟦T⟧(cs: Collection⟦T⟧) and (ds: Collection⟦T⟧) -> List⟦T⟧ {
    def cIter = cs.iterator
    def dIter = ds.iterator
    def result = list.empty

    if (cIter.hasNext.not) then { return result.addAll(ds) }
    if (dIter.hasNext.not) then { return result.addAll(cs) }

    var c := cIter.next
    var d := dIter.next

    while {cIter.hasNext && dIter.hasNext} do {
        if (c <= d) then {
            result.addLast(c)
            c := cIter.next
        } else {
            result.addLast(d)
            d := dIter.next
        }
    }

    if (c <= d) then {
        result.addAll [c,d]
    } else {
        result.addAll [d,c]
    }
    while {cIter.hasNext} do { result.addLast(cIter.next) }
    while {dIter.hasNext} do { result.addLast(dIter.next) }
    result
  }
```

4.11 Primitive Array

Primitive arrays can be constructed using primitiveArray.new(size) where size is the number of slots in the array. Initially, the contents of the slots are undefined. Primitive arrays are indexed from 0 though size − 1. They are intended as building blocks for more user-friendly objects. Most programmers should use list, set or dictionary rather than primitiveArray.

```
type Array⟦T⟧ = {
    size −> Number
    // returns the number of elements in self

    at(index: Number) −> T
    // returns the element of self at index

    at(index: Number) put (newValue: T) −> Done
    // updates the element of self at index to newValue

    sortInitial(n:Number) by(sortBlock:Function2⟦T, T, Number⟧) −> Self
    // sorts elements 0..n. The ordering is determined by sortBlock, which should return −1 if
    // its first argument is less than its second argument, 0 if they are equal, and +1 otherwise.

    iterator −> Iterator⟦T⟧
    // returns an iterator through my elements. It is an error to modify the array while
    // iterating through it.
}
```

4.12 Pipelines

When working with collections, it is often convenient to use the pipeline operator >>, because this allows the operations to be written in the order that they will be performed. For example,

```
list.withAll((1..10).filter{ each −> each.isEven })
```

constructs the range 1..10, removes the elememnts that are not even, and puts the result into a list. This sequence is revealed more clearly using a pipeline:

```
(1..10).filter{ each −> each.isEven } >> list
```

In general, the argument to >> can be any Sink, where

```
type Sink⟦T⟧ = interface {
    <<(source:Enumerable⟦T⟧) −> Collection⟦T⟧
}
```

The Sink interface is implemented by CollectionFactory, and by Collection. When the argument to >> is a factory like list or set, the result is a new collection of the appropriate kind, containing the elements of the receiver. When the argument to >> is an existing collection, the elements of the receiver will be added to it (if it is mutable), or a new collection containing the concatenation will be constructed (if it is imutable). The approriate behaviour is obtained in the implementation by requesting << on the argument.