Grace Documentation: Built-in Types and Data Structures

Kim B. Bruce Andrew P. Black

November 14, 2014

1 Introduction

Grace has several built-in types and object, and a growing selection of dialects and libraries that define other useful types and objects. Built-ins can be used without formality; dialects must be declared with the **dialect** statement, and libraries must be imported with the **import** statement.

This document records the current capabilities.

2 Built-In Types

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

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

```
type Object = {
  == (other: Object) −> Boolean
  // is other equal to self

  != (other: Object) −> Boolean
  /= (other: Object) −> Boolean
  ≠ (other: Object) −> Boolean
  // the inverse of ==.  All three variants have the same meaning.

  hashcode −> Number
  // the hash code of self, a Number in the range 0..2^32

  match (other: Object) −> SuccessfulMatch | FailedMatch
  // returns a SuccessfulMatch if self "matches" other
  // returns FailedMatch otherwise.
  // The exact meaning of "matches" depends on self.

  asString −> String
  // a string describing self

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

  :: (other:Object) −> Binding
  // a Binding object with self as key and other as value.
}
```

We will also not discuss or include the pattern combinators | (or) and & (and) used in generating compound patterns.

2.1 Number

Number represents all numeric values in minigrace, including integers and numbers with decimal fractions. (Thus, minigraceNumbers are what some other languages call floating point numbers, floats or double-precision). Numbers are represented with a precision of approximately 51 bits.

```
type Number = {

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

  -(other: Number) -> Number
  // return difference of self and other

  *(other: Number) -> Number
  // return product of self and other

  /(other: Number) -> Number
  // return quotient of self and other

  \%(other: Number) -> Number
  // return modulus of self and other (remainder after division)

  ..(last: Number) -> Number
  // return list of numbers from self to last

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

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

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

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

  prefix- -> Number
  // return negation of self

  inBase(base:Number) -> String
  // return a string representing self as a base number (e.g., base 2)

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

  rounded -> Number
  // return whole number closest to self
}
```

2.2 String

String literals are written surrounded by double quote characters. There are three escape characters:

- \n results in a newline
- \\ results in a single slash
- \" results in a double quote

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

```
type String = {
 * (n: Number) -> String
 // returns a string that contains n repetitions of self, so "abc" * 3 = "abcabcabc"

 & (other: Pattern) -> Pattern
 // returns a pattern that matches whatever is matched by both self and other.

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

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

 != (other: Object)
 \= (other: Object)
 ≠ (other: Object)
 // (self == other).not

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

 at(index: Number) -> String
 [ ](index:Number) -> String
 // returns the character in position index (as a string of size 1)

 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

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 \code{sortBy}

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

encode −> String
// returns an encodes version of self, with characters like " and \ escaped. (This method needs work!)

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

hashcode −> Number
// the hash of self

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

indexOf(sub:String) ifAbsent(absent:Block0<W>) −> Number | W
// returns the leftmost index at which sub appears in self; applies absent if it is not there.

indexOf(pattern:String)startingAt(offset)ifAbsent (action:Block0<W>) −> Number | W
// like the above, except that it returns the first index ≥ offset.

indices −> IteratorFactory
// 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

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

lastIndexOf() ifAbsent(absent:Block0<W>) −> Number | W
// returns the rightmost index at which sub appears in self; applies absent if it is not there.

lastIndexOf(pattern:String)startingAt(offset)ifAbsent (action:Block0<W>) −> Number | W
// like the above, except that it returns the last index ≤ offset.

match(other:Object) −> SuccessfulMatch | FailedMatch
// returns SuccessfulMatch match if self matches other, otherwise FailedMatch

```
    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.

    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)
    // 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.

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

2.3 Boolean

The Boolean literals are true and false.

```
type Boolean = {

  not -> Boolean
  prefix ! -> Boolean
  // negates of value of self

  && (other: Boolean) -> Boolean
  // return true when self and other are both true

  || (other: Boolean) -> Boolean
  // return true when either self or other (or both) are true

  andAlso (other: BlockBoolean) -> Boolean
  // other is a nullary block returning a Boolean.
  // short circuit version of &&; other is evaluated only when self is true

  orElse (other: BlockBoolean) -> Boolean
  // other is a nullary block returning a Boolean.
  // short circuit version of ||; other is evaluated only when self is false
}
```

2.4 Point

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 = {
  x -> Number
  // the x−coordinates of self

  y -> Number
  // the y−coordinate of 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)

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

  distanceTo(other:Point) -> Number
  // distance from self to other
}
```

3 Built-in data structures

These data structures are available to all Grace programs.

3.1 Iterator<T>

Items of type Iterator<T> provide ways of getting a sequence of values of type T.

```
type Iterator<T> = {
  // add elt to end of list
  havemore -> Boolean

  // add let to end of list
  next -> T
}
```

3.2 Array<T>

Primitive arrays can be built in Grace using class PrimitiveArray with PrimitiveArray.new(size) where size is the number of slots in the array. The primitive arrays are designed to be 0-indexed though that is not enforced in the Javascript backend.

```
type Array<T> = {
  size -> Number
  // return length of self

  at(index: Number) -> T
  // return element of array at given index
  [](index: Number) -> T

  at(index: Number) put (newValue: T) -> Done
  // update element of list at given index to newValue

  []:=(index: Number, newValue: T) -> Done
  // same as above, written as myList[n] := newValue

  contains(element: T) -> Boolean
  // does self contain element?

  iterator -> Iterator<T>
  // returns iterator through elements of list
}
```

3.3 List<T>

The type List<T> represents objects that are mutable lists of elements of type T. Object of type List<T> can be build with list.with<T>(a, b, c, ...) or `list.empty<T>`

Warning: Both List<T> and Iterator<T> will likely be given new definitions in the near future.

```
type List<T> = {
  // length of the list
  size -> Number

  // return element at nth position (starting from 1)
  at(n: Number) -> T
  [](n: Number) -> T

  // update element at nth position to x
  // requires 1 <= n <= size+1
  at(n: Number)put(x: T) -> List<T>
  []:=(n: Number,x: T) -> List<T>

  // add new element to end of list
  add(x: T) -> List<T>
  push(x: T) -> List<T>
  addLast(x: T) -> List<T>    // compatibility

  // remove and return last element of list
  removeLast -> T

  // add x as first element of list
  addFirst(x: T) -> List<T>

  // remove and return first element of list
  removeFirst -> T

  // remove and return nth element of list
  removeAt(n: Number) -> T

  // remove last element of list
  pop -> T

  // return list [1..size]
  indices -> Collection<T>   // range type?

  // return first, etc. element of list
  first -> T
  second -> T
  third -> T
  fourth -> T

  // return last element of list
  last -> T

  // concatenate receiver with o to form new list
  ++(o: List<T>) -> List<T>
```

```
    addAll(l: List<T>) -> List<T>

    // return string representation
    asString -> String

    extend(l: List<T>) -> Done

    // return whether element is in list
    contains(element) -> Boolean

    // apply operation in block1 to all element of list
    do(block1: Block1<T,Done>) -> Done

    // return whether lists have same elements in same order
    ==(other: Object) -> Boolean

    // return iterator through list
    iterator -> Iterator<T>

    // return a copy of the receiver
    copy -> List<T>
}
```

4 Built-In Libraries

4.1 Math

The mathematics module object can be imported using **import** `"math"` **as** m, for any identifier of your choice m. The object m responds to the following methods.

sin(θ: Number) −> Number
// trigonometric sine (θ in radians)

cos(θ: Number) −> Number
// cosine (θ in radians)

tan(θ: Number) −> Number
// tangent (θ in radians)

asin(r: Number) −> Number
// arcsine (result in radians)

acos(r: Number) −> Number
// arccosine (result in radians)

atan(r: Number) −> Number
//arctangent (result in radians)

pi −> Number
π −> Number
// 3.14159265...

abs(r: Number) −> Number
// absolute value

random −> Number
// random number between 0 and 1

lg(n: Number) −> Number
// $log_2 n$

ln (n: Number) −> Number
// $log_e n$

exp(n: Number) −> Number
// e^n

log10 (n: Number) −> Number
// $log_{10} n$