1 Introduction

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

2.1 Object

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

```plaintext
type Object = {

  == (other: Object) -> Boolean
  // true if other is equal to self

  != (other: Object) -> Boolean

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

  hash -> 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.

## 2.2 Number

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.

```typescript
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 sequence of numbers from self to last

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

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

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

  >=(other: Number) -> Boolean
  >(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 \rightarrow \text{Number} \\
// return number obtained by throwing away self's fractional part

rounded \rightarrow \text{Number} \\
// return whole number closest to self

abs \rightarrow \text{Number} \\
// the absolute value of self

2.3 String

String literals 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.

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

type String = { 
  \* (n: \text{Number}) \rightarrow \text{String} \\
  // returns a string that contains n repetitions of self, so "abc" \* 3 = "abcabcabc"

  & (other: \text{Pattern}) \rightarrow \text{Pattern} \\
  // returns a pattern that matches whatever is matched by both self and other.

  ++(other: \text{Object}) \rightarrow \text{String} \\
  // returns a string that is the concatenation of self and other.asString

  < (other: \text{String}) \\
  // true if self precedes other lexicographically

  <= (other: \text{String}) \\
  \leq (other: \text{String}) \\
  // (self == other) \lor (self < other)

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

  != (other: \text{Object}) \\
  \neq (other: \text{Object}) \\
  // (self == other).not

  > (other: \text{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); index must be in the range 1..size

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 sortBy

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

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

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

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

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

lastIndexOf(sub:String) 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) ➞ 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.

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

}
### 2.4 Boolean

The Boolean literals are true and false.

```typescript
import { Boolean, Number } from './types';

// type definition
interface Boolean {

    not() => Boolean; // the negation of self
    prefix !() => Boolean; // the negation 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

}
```

### 2.5 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).

```typescript
interface Point {

    x() => Number; // the x-coordinate 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

}
```
2.6 Binding

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).

```plaintext
type Binding<K, T> = {
    key -> K // returns the key
    value -> T // returns the value
}
```

3 Collection objects

The objects described in this section are made available to all standard Grace programs. (This means that they are defined as part of the standardGrace 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 > used as 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.

3.1 Common Abstractions

The major kinds of collection are sequence, list, set and dictionary. Although these objects differ in their details, they share many common properties. First, they can all be created by similar methods:

```plaintext
type CollectionFactory<T> = type {
    with (*elts:Object) -> Collection<T>
    // creates a collection of my kind that contains *elts, which is a variable—length argument list.
    // Thus, set.with(5, 2) returns a set with members 2 and 5, and list.with(1, 2, 3, 5) returns a
    // list containing the elements 1, 2, 3 and 5, in that order.

    empty -> Collection<T>
    // creates and returns an empty collection of my kind.

    withAll (elts:Collection<T>) -> Collection<T>
    // like with, except that elts is a single argument, which is a collection, rather than a
    // variable—length list of arguments.
}
```

Second, they share many common methods, which are defined in the type Collection<T>. In the description below, the term elements refers to the values in a list or dictionary.

```plaintext
type Collection<T> = Object & type {
    isEmpty -> Boolean
    // returns true if has no elements

    do(action:Block1<T, Unknown>)
    // applies action to each element of self

    do(action:Block1<T, Unknown>) separatedBy(sep:Block0<Unknown>)
    // applies action to each element of self, and applies sep (to no arguments) in between.
```
map<R>(unaryFunction:Block1<T, R>) -> Enumerable<R>
// 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:Block2<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:Block1<T, Boolean>) -> Enumerable<T>
// returns a new collection containing only those elements of self for which
// condition holds. The result is ordered if self is ordered.

iterator -> Iterator<T>
// returns an iterator over my elements. It is an error to modify self while iterating over it.

++ (other: Collection<U>) -> Enumerable<T | U>
// returns a new Enumerable containing all my values, followed by all of the values from other.
// If other is ordered, then that order will be maintained.

asSequence -> Sequence<T>
// returns a Sequence containing my elements.

asList -> List<T>
// returns a (mutable) list containing my elements.

asSequence -> Sequence<T>
// returns a sequence containing my elements.

asSet -> Set<T>
// returns a (mutable) Set containing my elements, with duplicates eliminated.
// The == operation on my elements is used to identify duplicates.

Additional methods are available in the type Enumerable; an Enumerable is like a Sequence, but where the elements must be enumerated one by one, in order, using a computational process, rather than being stored explicitly. For this reason, operations that require access to all of the elements at one time are not supported, except for conversion to other collections that store their elements.

type Enumerable<T> = type {
  size -> Number
  // the number elements in self. If it is computationally expensive to
  // calculate size, it's permissible to raise the exception SizeUnknown.
  // A client who really needs to know my size should iterate through my elements and
  // count them, or convert me to a list or sequence.

  values -> Collection<T>
  // an enumeration of my values: the elements in the case of sequence or list,
  // the values the case of a dictionary.

  asDictionary -> Dictionary<Number, T>
  // returns a dictionary containing my indices as keys and my elements as values, so that
  // my i_th element is self.asDictionary.at(i).
}
keysAndValuesDo (action:Block2<Number, T, Object>) —> Done
  // applies action, in sequence, to each of my keys and the corresponding element.

onto(f:CollectionFactory<T>) —> Collection<T>
  // uses the factory f to create a new collection, then populates it with my elements;
  // returns the new collection.

into(existing:Collection<T>) —> Collection<T>
  // adds my elements to existing, and returns existing.

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

sortedBy(sortBlock:Block2<T, T, Number>) —> Sequence<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.

}  

3.2  Sequence

The type Sequence<T> describes sequences of values of type T. Sequence objects are immutable; they can be constructed either explicitly, using sequence.with(1, 3, 5, 7), or as ranges like 1..10.

type Sequence<T> = Indexable<T> & type {

  at(ix:Number) —> T
  [[ix:Number]) —> T
  // returns my ixth element, provided ix is integral and 1 ≤ ix ≤ size

  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.
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:Block0<W>) -> Number | W
  // returns the index of the 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

3.3 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 conveniently used to create ranges. Thus, 3..9 is the same as range.from 3 to 9, and (3..9).reversed is the same as range.from 9 downTo 3.

3.4 List

The type List<T> describes objects that are mutable lists of elements that have type T. Like sets and sequences, list objects can be constructed using the empty, with, and withAll requests, as in list.empty<T>, list.with<T>(a, b, c,...), or list.withAll<T>(existingCollection).

  type List<T> = Indexable<T> & type {

    at(n: Number) put(new:T) -> List<T>
      // updates self so that my n^{th} element is new. Returns self.
      // Requires 1 ≤ n ≤ size+1; when n = size+1, equivalent to addLast(new).

    []:=(n: Number, new:T) -> Done
      // updates self so that my n^{th} element is new; requested by writing lst[n] := new. Returns done
      // 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. Change the index of all of the existing elements.

addAllFirst(news:Collection<T>) => List<T>
// adds news as the first elements of self. Change 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
// remove and return last element of self.

removeAt(n:Number) => T
// removes and returns n\textsuperscript{th} element of self.

remove(*element:T) => List<T>
// removes element(s) from self. Raises a NoSuchObject exception if not.self.contains(element).
// Returns self

remove(*element:T) ifAbsent(action:Block0<Unknown>) => List<T>
// removes element(s) from self; executes action if any of them is not contained in self. Returns self

removeAll(elements:Collection<T>) => List<T>
// removes elements from self. Raises a NoSuchObject exception if any one of
// them is not contained in self. Returns self

removeAll(elements:Collection<T>) ifAbsent(action:Block0<Unknown>) => List<T>
// removes elements from self; executes action if any of them is not contained in self. Returns self

++ (other:List<T>) => List<T>
// returns a new list formed by concatenating self and other

addAll(extension:List<T>) => List<T>
// extends self by appending 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 the same elements
// in the same order.

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

sortBy(sortBlock:Block2<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>
3.5 Sets

Sets are unordered collections of elements without duplicates. The `==` method on the elements is used to detect and eliminate duplicates; it must be symmetric.

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

dd(element:T) -> Set<T>
    // adds element(s) to self. Returns self.

ddAll(elements:Collection<T>) -> Set<T>
    // adds elements to self. Returns self.

rmove(element: T) -> Set<T>
    // removes element(s) from self. It is an error if element is not present. Returns self.

rmove(elements: T) ifAbsent(block: Block0<Done>) -> Set<T>
    // removes element(s) from self. Executes action if element is not present. Returns self.

rmoveAll(elems:Collection<T>)
    // removes elems from self. It is an error if any of the elems is not present. Returns self.

rmoveAll(elems:Collection<T>) ifAbsent(action:Block0<Done>) -> Set<T>
    // removes elems from self. Executes action if any of elems is not present. Returns self.

contains(elem:T) -> Boolean
    // true if self contains elem

includes(predicate: Block1<T,Boolean>) -> Boolean
    // true if predicate holds for any of the elements of self

find(predicate: Block1<T,Boolean>) ifNone(notFoundBlock: Block0<T>) -> T
    // returns an element of self for which predicate holds, or the result of applying notFoundBlock is there is none.

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).

onto(f:CollectionFactory<T>) → Collection<T>
// uses the factory f to create a new collection, then populates it with my elements;
// returns the new collection.

into(existing:Collection<T>) → Collection<T>
// adds my elements to existing, and returns existing.
}

3.6 Dictionary

The type Dictionary<K, T> describes objects that are mappings from keys of type K to values of type T. Like sets and sequences, dictionary objects can be constructed using the empty, with, and withAll requests, but in this case the arguments to with must be of type Binding, i.e., they must have methods key and value. Bindings can be conveniently created using the infix :: operator, as in dictionary.empty<K, T>, dictionary.with<K, T>(k::v, m::w, n::x, ...), or dictionary.withAll<K, T> (existingCollection: Enumerable<Binding<K, T>>).

type Dictionary<K, T> = Collection<T> & type {
  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

  []=(key:K, value:T) → Done
  // puts value at key; returns done

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

  at(k:K) ifAbsent(action:Block0<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)
  containsValue(v)
  // returns true if one of my values == v

  removeAllKeys(keys: Collection<K>) → Dictionary<K, T>
  // removes all of the keys from self, along with the corresponding values. Returns self.

  removeKey(*keys: K) → Dictionary<K, T>
  // removes keys from self, along with the corresponding values. Returns self.

  removeAllValues(removals: Collection<V)) → Dictionary<K, T>
  // removes from self all of the values in removals, along with the corresponding keys. Returns self.
removeValue(removals)
// removes from self removals, along with the corresponding keys. Returns self.

keys -> Enumerable<K>
// returns my keys as a lazy sequence

values -> Enumerable<K>
// returns my values as a lazy sequence

bindings -> Enumerable<Binding<K, V>>
// returns my bindings as a lazy sequence

keysAndValuesDo(action: Block2<K, T, Object>) -> Done
// applies action, in arbitrary order, to each of my keys and the corresponding value.

keysDo(action: Block2<K, Object>) -> Done
// applies action, in arbitrary order, to each of my keys.

valuesDo(action: Block2<T, Object>) -> Done
// applies action, in arbitrary order, to each of my values.

do(action: Block2<T, Object>) -> Done
// applies action, in arbitrary order, to each of my values.

copy -> Dictionary<K, V>
// returns a new dictionary that is a shallow copy of self

asDictionary -> Dictionary<K, T>
// returns 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

3.7 Iterator<T>

Object of type Iterator<T> implement the external iterator pattern: they provide way of iterating through a sequence of values of type T, one element at a time. Once an iterator has been exhausted, it will remain exhausted. It is an error (presently undetected) to modify an object while iterating through it.

type Iterator<T> = type {
  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 the Exhausted exception.
}
3.8 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 through `size - 1`. They are intended as building blocks for more user-friendly objects. Most programmers should use `list`, `set` or `dictionary` rather than `primitiveArray`.

```plaintext
type Array<T> = {
  size -> Number
  // return the number of elements in self

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

  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

  sortInitial(n:Number) by(sortBlock:block2<T, T, Number>) -> Boolean
  // 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 iterator through the elements of self. It is an error to modify the array while
  // iterating through it.
}
```
4 Built-In Libraries

4.1 Math

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

- \texttt{sin(\theta: Number) \to Number}
  \hfill \texttt{// trigonometric sine (\theta in radians)}

- \texttt{cos(\theta: Number) \to Number}
  \hfill \texttt{// cosine (\theta in radians)}

- \texttt{tan(\theta: Number) \to Number}
  \hfill \texttt{// tangent (\theta in radians)}

- \texttt{asin(r: Number) \to Number}
  \hfill \texttt{// arcsine (result in radians)}

- \texttt{acos(r: Number) \to Number}
  \hfill \texttt{// arccosine (result in radians)}

- \texttt{atan(r: Number) \to Number}
  \hfill \texttt{// arctangent (result in radians)}

- \texttt{pi \to Number}
  \hfill \texttt{\pi \to Number}
  \hfill \texttt{// 3.14159265...}

- \texttt{abs(r: Number) \to Number}
  \hfill \texttt{// absolute value}

- \texttt{random \to Number}
  \hfill \texttt{// random number between 0 and 1}

- \texttt{lg(n: Number) \to Number}
  \hfill \texttt{// log_{2\pi}n}

- \texttt{ln (n: Number) \to Number}
  \hfill \texttt{// log_{e}n}

- \texttt{exp(n: Number) \to Number}
  \hfill \texttt{// e^n}

- \texttt{log10 (n: Number) \to Number}
  \hfill \texttt{// log_{10}n}