Stanza Reference Manual

Patrick Li

January 2019

Contents

Chapter 1

Core Macros

The core macros are the default macros used for expanding s-expressions into Stanza core forms.

Types

Function Type

A function type is specified using the `->` operator. The arguments to the function are to the left-hand side of the `->`, and the return type of the function is on the right-hand side of the `->`.

```
(Int, Int) -> Int
```

If there is exactly a single argument, then the parentheses surrounding the argument types may be omitted.

```
Int -> Int
```

Intersection Type

An intersection type is specified using the `&` operator.

```
Seqable & Lengthable
```

Union Type

A union type is specified using the `|` operator.

```
Int | String
```

Captured Type Argument

A captured type argument is specified by prefixing an identifier with the `?` operator.

```
?T
```

Tuple Type

A tuple type is specified by surrounding a sequence of types between the `[]` brackets.

```
[Int, String]
```

Parametric Type

A parametric type is specified by following an identifier with a sequence of types between the `<>` brackets.

```
Table<Int, String>
```

Unknown Type

The special identifier `?` represents the unknown type.

```
?
```

Void Type

The special identifer `Void` represents the void type.

```
Void
```

Declarations

Package Declarations

The following form declares a new package with no imported packages.

```
defpackage mypackage
```

Bindings from other packages may be imported using the `import` statement.

```
defpackage mypackage :
  import core
  import collections
```

Prefixes may be added to bindings from an imported package. The following form will append the prefix `core-` to every imported binding from the `core` package, except for bindings named `False`, which will instead have the `C` prefix.

```
defpackage mypackage :
  import core with :
    prefix => core-
    prefix(False) => C
```

Public Visibility

Either of the following forms can be used to indicate that the following declaration(s) is/are publicly visible.

```
public decl
public : decl
```

The colon version is typically used to indicate the visibility of multiple declarations.

```
public :
  val x = 10
  val y = 20
```

Protected Visibility

Either of the following forms can be used to indicate that the following declaration(s) has/have protected visibility. This indicates that they can only be referenced from other packages through package-qualified identifiers.

```
protected decl
protected : decl
```

Type Declarations

The following form defines the type `MyType`.

```
deftype MyType
```

The following form defines the type `MyType` as a subtype of `Collection` and `Lengthable`.

```
deftype MyType <: Collection & Lengthable
```

The following form defines the type `MyType`, and also defines the types `Int` and `String` to be subtypes of `MyType`.

```
deftype MyType :
  Int <: MyType
  String <: MyType
```

Parametric types may be defined as follows.

```
deftype MyType<T>
```

These type parameters may be used in the specification of the types parents.

```
deftype MyType<T> <: Collection<T> & Lengthable
```

Struct Declarations

The following form defines the struct `MyStruct` with the given fields `x` of type `XType` and `y` of type `YType`.

```
defstruct MyStruct :
  x: XType
  y: YType
```

The above form will expand into:

1. a type declaration for the type `MyStruct`.
2. a constructor function called `MyStruct` that takes in two arguments, an `XType` and a `YType` object and returns a `MyStruct` object.
3. two getter functions named `x` and `y` that return the associated field in the struct.

To change the name of the generated constructor function, the following option may be used.

```
defstruct MyStruct :
  x: XType
  y: YType
with :
  constructor => #MyStruct
```

The generated constructor function will now be named `#MyStruct` instead of `MyStruct` with the above form.

The parent type of a struct may be provided using the following form.

```
defstruct MyStruct <: Parent :
  x: XType
  y: YType
```

To create a parametrically typed struct, type arguments can be given following the type name.

```
defstruct MyStruct<T> :
  x: XType
  y: YType
  z: Array<T>
```

Mutable fields may be declared by providing the name of a setter function. In the following example, a setter function called `set-x` will be generated for setting the value of the `x` field.

```
defstruct MyStruct :
  x: XType with: (setter => set-x)
  y: YType
```

The generated `set-x` function has the following type signature.

```
defn set-x (s:MyStruct, x:XType) -> False
```

To generate field getters and setters as methods instead of as functions, the following form may be used.

```
defstruct MyStruct :
  x: XType with: (as-method => true)
```

The above example will generate the `x` getter as a method instead of a function.

```
defmethod set-x (s:MyStruct, x:XType) -> False
```

Function Declarations

Argument lists have the following form:

```
(x:Int, y:Int, z)
```

Each argument binding consists of a binding expression (typically an identifier) followed by a colon and an optional type for the argument.

Tuples (and nested tuples) may be used for binding expressions in order to deconstruct an argument's value.

```
([x, y]:[Int, String], y:Int, z)
```

The following form defines the named function `f`.

```
defn f (x, y) :
  x + y
```

If no explicit type is given for an argument to a named function, then by default it is assigned the unknown type.

Users may optionally declare the types of the arguments and the return value using the following form.

```
defn f (x:Int, y:String) -> Char :
  x + y
```

To declare tail-recursive functions use the `defn*` tag in place of the `defn` tag in the above forms.

Multi Declarations

The following form declares a multi, `f`, with the given type signature.

```
defmulti f (x:Int, y:String) -> String
```

If no explicit types are provided for the arguments or return type, then they have the unknown type by default.

```
defmulti f (x, y)
```

Method Declarations

The following form declares a method for the multi `f`.

```
defmethod f (x, y) :
  x + y
```

If no explicit types are provided for the arguments then they are assigned the unknown type by default. Users may optionally declare the types of the arguments and the return value using the following form.

```
defmethod f (x:Int, y:String) -> Int :
  x + y
```

To declare tail-recursive methods use the `defmethod*` tag in place of the `defmethod` tag in the above forms.

Value Declarations

The following form declares a value `x` of type `Int` initialized to the expression `v`.

```
val x : Int = v
```

If the type is omitted then it is inferred from the given value.

```
val x = v
```

Tuples may be used within the binding expression to destructure the given value.

```
val [x, [y, z]] = v
```

Variable Declarations

The following form declares a variable.

```
var x : Int = v
```

If the type is omitted then it is inferred from any assignments to the variable.

```
var x = v
```

The initializing expression may be omitted to declare an uninitialized variable. It is an error to read from a variable that is not yet initialized.

```
var x
```

Expressions

There are only a handful of core forms in Stanza but the core macro library provides a large collection of syntactic sugar for expressing commonly used constructs.

Literal S-Expressions

The backtick form represents a literal s-expression. For example, the following form creates a list consisting of three elements: the symbol f, the integer 3, and the symbol g.

```
`(f 3 g)
```

Expression Sequences

Multiple expressions surrounded with parenthesis represent a sequence of expressions, whose value is equal to the value of the last expression.

```
(x + 3, 3, f(3))
```

Function Calls

Expressions followed immediately opening parenthesis expand to function call forms.

```
f(x)
```

Polymorphic Function Calls

Expressions followed by angle brackets then opening parenthesis expand to polymorphic function call forms.

```
f<Int>(x)
```

Calls to Get

Expressions followed by opening square brackets expand to a call to the `get` function.

```
f[x, y]
```

expands to

```
get(f, x, y)
```

Calls to Set

Expressions followed by opening square brackets and then an equal sign expands to a call to the `set` function.

```
f[x, y] = v
```

expands to

```
set(f, x, y, v)
```

Anonymous Functions

Expressions wrapped in curly brackets expand to an anonymous function, with underscores replaced by arguments.

```
{_ + y}
```

expands to

```
fn (x) : x + y
```

For convenience, if there are no underscores, then the created anonymous function can take both no arguments and a single argument.

```
{y}
```

expands to

```
multifn :
  () : y
  (x) : y
```

Curried Functions

Expressions followed by an opening curly bracket expand to an anonymous function, with underscores replaced by arguments.

```
f{_, y}
```

expands to

```
fn (x) : f(x, y)
```

If there are no underscores, then the created anonymous function can take both no arguments and a single argument.

```
f{y}
```

expands to

```
multifn :
  () : f(y)
  (x) : f(y)
```

Casts

A value can be downcast to a given type using the `as` operator.

```
x as Int
```

If followed with an `else` clause, then the as operator will return the given default value if the expression is not of the given type.

```
x as Int else y
```

Upcasts

A value can be upcast to a given type using the `as?` operator.

```
x as? Int
```

Type Checks

The following form returns true if the given value is of the type `Type` or `false` otherwise.

```
x is Type
```

The following form returns true if the given value is not of the type `Type` or `false` otherwise.

```
x is-not Type
```

Tuples

Tuples are created by surrounding a sequence of expressions with the `[]` brackets.

```
[x, y]
```

Objects

The following form creates a new object of the type `MyType` with no instance methods.

```
new MyType
```

The following form creates a new object of the type `MyType` with instance methods. An instance method follows the same syntax as top-level methods, but requires precisely one argument named `this` that refers to the object being created.

```
new MyType :
  defmethod f (this, y:Int) :
    x + y
```

Ranges

The following form creates a `Range` from `a` to `b` (exclusive) with a step size of 1.

```
a to b
```

A step size may be provided explicit using the `by` keyword.

```
a to b by 2
```

To indicate that the ending index should be inclusive rather than exclusive, the `through` keyword may be used in place of the `to` keyword.

```
a through b
a through b by 2
```

KeyValue Objects

The following form creates a `KeyValue` object from the given key object `k`, and the value object `v`.

```
k => v
```

Apply Operator

The following operator calls the expression `f` with the argument `x`.

```
f $ x
```

It is often used in conjunction with curried functions to chain together long expressions.

```
request-price $
{_ * 0.75} $
calc-mortgage{_, 75.00} $
base-price()
```

If Conditionals

The following form evaluates one of two branches depending on whether the predicate `p` is `true` or `false`.

```
if p :
  consequent
else :
  alternate
```

If the `else` branch is omitted, then a default `else` branch is provided that simply returns `false`.

```
if p :
  consequent
```

If the `else` branch consists of a single `if` expression, then the colon following the `else` may be omitted. This allows multiple `if` expressions to be chained elegantly.

```
if p1 :
  consequent1
else if p2 :
  consequent2
else if p3 :
  consequent3
else :
  alternate
```

When Conditionals

The following form allows for simple branches on `true`/`false` to be written concisely.

```
consequent when p else alternate
```

expands to:

```
if p :
  consequent
else :
  alternate
```

If the `else` clause is omitted, then it returns `false` by default.

Match Conditionals

The following form evaluates one of a multiple of branches depending on the types of its arguments. If no type is given for a branch argument then it is inferred from the type of the input arguments.

```
match(x, y) :
  (x:Int, y:Int) : body0
  (x:Int, y:String) : body1
  (x:String, y:Int) : body2
  (x, y) : body3
```

Switch Conditionals

The following form creates a chain of `if-else` expressions for evaluating a sequence of bodies.

```
switch f :
  a : body0
  b : body1
  c : body2
  d : body3
  else : body4
```

expands to:

```
if f(a) : body0
else if f(b) : body1
else if f(c) : body2
else if f(d) : body3
else : body4
```

If no `else` branch is given, then the default behaviour is to exit the program with a fatal error.

The `switch` is commonly used with the `equal?` predicate function.

```
switch {x == _} :
  1 : first-action()
  2 : second-action()
  3 : third-action()
```

Let Scopes

The following form evaluates its body under a new scope.

```
let :
  body
```

Where Scopes

The following form evaluates an expression with a set of declared bindings.

```
exp where :
  body
```

expands to:

```
let :
  body
  exp
```

The following example demonstrates assigning `x` to the result of `42 - (10 * y)`.

```
x = (42 - z) where :
  val z = 10 * y
```

For "Loops"

The following form calls an operating function, `f`, with an anonymous function and a list of arguments.

```
for (x in xs, y in ys, z in zs) f :
  body
```

expands to:

```
f((fn (x, y, z) : body), xs, ys, zs)
```

If there is only a single set of bindings, then the parenthesis may be omitted.

```
for x in xs f :
  body
```

By far, the most common usage of the `for` construct is with the `do` function to iterate over sequences. The following example prints the integers between `0` and `10`.

```
for i in 0 to 10 do :
  println(i)
```

While Loops

The following form creates a loop that repeats as long as a given predicate returns `true`.

```
while p :
  body
```

Labeled Scopes

The following form creates a labeled scope which returns an object of type Type and with an exit function named myreturn.

```
label<Type> myreturn :
  body
```

If no explicit type is provided, then the body is assumed to return false, and the exit function takes no arguments.

```
label myreturn :
  body
```

The most common use of labeled scopes is to return early from a function. The following example demonstrates a function that finds the first integer whose square is over 100.

```
defn first-big-square () :
  label<Int> return :
    for i in 0 to false do :
      return(i) when i * i > 100 :
    fatal("No such number.")
```

Generators

The following form generates a sequence of objects of type Type given the generator body. Within the body, a distinguished function called yield takes a single argument and includes it in the resulting sequence. A distinguished function called break is used to end the sequence. If no argument is passed to break then the sequence ends immediately. If a single argument is passed to break then it is included in the resulting sequence before the sequence ends.

```
generate<Type> :
  body
```

If no explicit type is provided, then the default type is the unknown type.

```
generate :
  body
```

The following example demonstrates a function that returns the first hundred perfect squares.

```
generate<Int> :
  for i in 1 through 100 do :
    yield(i * i)
```

To demonstrate the break function, the above example could also be written with a while loop like so.

```
generate<Int> :
  var i = 1
  while true :
    if i == 100 : break(i * i)
    else : yield(i * 1)
    i = i + 1
```

Dynamic Variables

The following form temporarily sets the variable `x` to the value `v` while executing the given body. The variable is restored to its original value after the body is finished.

```
let-var x = v :
  body
```

If there is more than one set of bindings, then they are set and restored in parallel.

```
let-var (x = v1, y = v2) :
  body
```

Exception Handling

The following form executes a given body after installing the given exception handlers and `finally` block. Both the exception handlers and the `finally` block are optional, but a `try` expression must contain either a `finally` block or an exception handler. If the `finally` block is provided, then it is a fatal error to exit and then re-enter the body through the use of coroutines or generators.

```
try :
  body
catch (e:MyException1) :
  handler1
catch (e:MyException2) :
  handler2
finally :
  cleanup
```

Attempts and Failures

The following form executes a given body with a given failure handler. If no `else` clause is given, then the default behaviour is to return `false`.

```
attempt :
  body
else :
  handle failure
```

Operators

Operators in Stanza expand to simple function calls. The following is a listing of the core operators and their expanded form.

```
(- x)       ; expands to  negate(x)
(~ x)       ; expands to  bit-not(x)
not x       ; expands to  complement(x)
x == y      ; expands to  equal?(x, y)
x != y      ; expands to  not-equal?(x, y)
x < y       ; expands to  less?(x, y)
x <= y      ; expands to  less-eq?(x, y)
x > y       ; expands to  greater?(x, y)
x >= y      ; expands to  greater-eq?(x, y)
x + y       ; expands to  plus(x, y)
x - y       ; expands to  minus(x, y)
x * y       ; expands to  times(x, y)
x / y       ; expands to  divide(x, y)
x % y       ; expands to  modulo(x, y)
x << y      ; expands to  shift-left(x, y)
x >> y      ; expands to  shift-right(x, y)
x >>> y     ; expands to  arithmetic-shift-right(x, y)
x & y       ; expands to  bit-and(x, y)
x | y       ; expands to  bit-or(x, y)
x ^ y       ; expands to  bit-xor(x, y)
```

Anonymous Functions

The following form creates an anonymous function with two arguments.

```
fn (x, y) :
  x + y
```

If no explicit type is given for an argument then it will be inferred from the context in which the function is used. Users may optionally declare the types of the arguments and the return value using the following form.

```
fn (x:Int, y:String) -> Char :
  x + y
```

To make an anonymous function tail recursive, use the `fn*` tag in place of the `fn` tag in the above forms.

Multiple Arity Functions

The following forms creates an anonymous function with multiple arities.

```
multifn :
  (x) : x + 1
  (x, y) : x + 2
```

If no explicit type is given for an argument then it will be inferred from the context in which the function is used. Users may optionally declare the types of the arguments and the return values using the following form.

```
multifn :
  (x:Int) -> Int : x + 1
  (x:Int, y:String) -> String : x + 2
```

To make a multiple arity function tail recursive, use the `multifn*` tag in place of the `multifn` tag in the above forms.

Variable Assignments

The following form assigns the expression `v` to the variable `x`. An assignment expression returns `false`.

```
x = v
```

Quasi Quote

The `qquote` form returns an s-expression with indicated expressions substituted in.

```
qquote(a b c)
```

The above example returns the list `(a b c)`. The following example demonstrates the `~` substitution operator and returns the list `(a 3 c)`.

```
val b = 3
qquote(a ~ b c)
```

The following example demonstrates the `~@` splicing operator and returns the list returns the list `(a 1 2 3 c)`.

```
val b = `(1 2 3)
qquote(a ~@ b c)
```

Compile Time Flags

The following form will define the given compile-time flag.

```
#define(flag)
```

The following form expands into the consequent if the given compile-time flag is defined or into the alternate otherwise.

```
#if-defined(flag) :
  consequent
#else :
  alternate
```

If the `#else` branch is not provided then it expands into the empty expression `()` if the flag is not defined.

The following form expands into the consequent if the given compile-time flag is not defined or into the alternate otherwise.

```
#if-not-defined(flag) :
  consequent
#else :
  alternate
```

If the `#else` branch is not provided then it expands into the empty expression `()` if the flag is defined.

Chapter 2

Core Package

The `core` package consists of functions and types used for basic programming.

Equalable

Values that support the `equal?` operation are indicated as subtypes of Equalable.

```
deftype Equalable
```

Mandatory minimal implementation: `equal?`.

equal?

The multi `equal?` takes as arguments two `Equalable` values and returns `true` if they are equal or `false` otherwise.

```
defmulti equal? (a:Equalable, b:Equalable) -> True|False
```

A default method for equal? is provided that simply returns `false`.

Equality in Stanza is defined to mean "invariant to substitution". That is, suppose that our program contains a call to a function, `f`, with a value `x`.

```
f(x)
```

But suppose, that earlier, we have determined that

```
x == y
```

returns `true`, indicating that `x` is equal to `y`. Then we should we able to substitute the usage of `x` with `y` instead without changing the behaviour of the program.

```
f(y)
```

Consistent to this definition, in Stanza's core library, values of immutable types are defined to be equal if their subfields are equal. Different mutable values, such as arrays, are never defined to be equal unless they refer to the same object.

not-equal?

`not-equal?` returns `true` if its two arguments are not equal and `false` otherwise.

```
defn not-equal? (a:Equalable, b:Equalable) -> True|False
```

Comparable

The `Comparable` type is used to indicate that a value can compared against other values using the comparison operations. A value of type `Comparable<T>` can be compared against values of type `T`.

```
deftype Comparable<T>
```

Mandatory minimal implementation: `compare`.

compare

The following multi compares values of type `Comparable<T>` against values of type `T`. The multi returns a negative integer if the item `a` is less than the item `b`, zero if the two are equal, and a positive integer if the item `a` is greater than the item `b`. For values that are subtypes of both `Equalable` and `Comparable`, the compare operation must be consistently defined against the `equal?` operation, and return `0` if `equal?` returns `true`.

```
defmulti compare<?T> (a:Comparable<?T>, b:T) -> Int
```

less?, greater?, less-eq?, greater-eq?

The following multis are provided for subtypes to provide efficient implementations. Any defined method must return results consistent with the compare operation. Default methods are provided for these multis defined in terms of the `compare` operation.

```
defmulti less?<?T> (a:Comparable<?T>, b:T) -> True|False
defmulti greater?<?T> (a:Comparable<?T>, b:T) -> True|False
defmulti less-eq?<?T> (a:Comparable<?T>, b:T) -> True|False
defmulti greater-eq?<?T> (a:Comparable<?T>, b:T) -> True|False
```

max, min

The following functions compute the maximum or minimum of their two given arguments.

```
defn max<?T,?S> (a:?S&Comparable<?T>, b:T) -> S|T
defn min<?T,?S> (a:?S&Comparable<?T>, b:T) -> S|T
```

maximum, minimum

The following functions compute the minimum between an initial value `x0`, and a sequence of items `xs` using the provided comparison function. If no initial value is provided, then the sequence `xs` cannot be empty.

```
defn minimum<?T> (x0:T, xs:Seqable<?T>, less?: (T, T) -> True|False) ->
    T
defn minimum<?T> (xs:Seqable<?T>, less?: (T, T) -> True|False) -> T
```

The following functions compute the maximum between an initial value `x0`, and a sequence of items `xs` using the provided comparison function. If no initial value is provided, then the sequence `xs` cannot be empty.

```
defn maximum<?T> (x0:T, xs:Seqable<?T>, less?: (T, T) -> True|False) ->
    T
defn maximum<?T> (xs:Seqable<?T>, less?: (T, T) -> True|False) -> T
```

The following functions compute the minimum between an initial value `x0`, and a sequence of comparable items `xs` using the `less?` multi. If no initial value is provided, then the sequence `xs` cannot be empty.

```
defn minimum<?T> (x0:T&Comparable, xs:Seqable<?T&Comparable>) -> T
defn minimum<?T> (xs:Seqable<?T&Comparable>) -> T
```

The following functions compute the maximum between an initial value `x0`, and a sequence of comparable items `xs` using the `less?` multi. If no initial value is provided, then the sequence `xs` cannot be empty.

```
defn maximum<?T> (x0:T&Comparable, xs:Seqable<?T&Comparable>) -> T
defn maximum<?T> (xs:Seqable<?T&Comparable>) -> T
```

The following function returns the minimum in a sequence of items `xs` by comparing the keys extracted from each item using the provided `key` function.

```
defn minimum<?T> (key: T -> Comparable, xs:Seqable<?T>) -> T
```

The following function returns the maximum in a sequence of items `xs` by comparing the keys extracted from each item using the provided `key` function.

```
defn maximum<?T> (key: T -> Comparable, xs:Seqable<?T>) -> T
```

Hashable

The type `Hashable` indicates that a value supports the `hash` multi and can represented (non-uniquely) as an `Int` value.

```
deftype Hashable
```

Mandatory minimal implementation: `hash`.

hash

The `hash` multi takes a `Hashable` argument and returns an `Int`. For the correct operation of `HashTable` implementations, the definition of `hash` must be consistent with the definition of `equal?`. If two values are equal, then they must also return the same hash.

```
defmulti hash (h:Hashable) -> Int
```

Lengthable

A `Lengthable` is a value that has an unknown length.

```
deftype Lengthable
```

Mandatory minimal implementation: `length`.

length

The `length` multi calculates the length of a `Lengthable` object.

```
defmulti length (l:Lengthable) -> Int
```

Seq

A Seq represents a possibly infinite sequence of items of type T.

```
deftype Seq<T> <: Seqable<T>
```

Mandatory minimal implementation: empty?, next, peek.

empty?

The empty? multi returns true if the sequence contains more items, or false otherwise.

```
defmulti empty? (s:Seq) -> True|False
```

next

The next multi returns the next item in the sequence. Repeated calls to next results in successive items in the sequence being returned. It is a fatal error to call next on an empty sequence.

```
defmulti next<?T> (s:Seq<?T>) -> T
```

peek

The peek multi inspects the next item in the sequence but does not consume it. Repeated calls to peek results in the same item being returned. It is a fatal error to call peek on an empty sequence.

```
defmulti peek<?T> (s:Seq<?T>) -> T
```

Seqable

A Seqable represents any object that may be viewed as a sequence of items of type T.

```
deftype Seqable<T>
```

Mandatory minimal implementation: to-seq.

to-seq

The to-seq multi returns a Seq representing a sequential view of the contents of the given object s.

```
defmulti to-seq<?T> (s:Seqable<?T>) -> Seq<T>
```

A value of type `Seq` is defined to also be a `Seqable` and trivially returns itself when called with `to-seq`.

Collection

Similar to `Seqable`, a `Collection` also represents an object that may be viewed as a sequence of items of type `T`. The crucial difference between a `Collection` and `Seqable` is that a `Collection` must be *repeatedly* viewable as a sequence of items.

As an example, consider a function, `print-all-twice`, that iterates through and prints out the items in a sequence twice.

```
defn print-all-twice (xs:Seqable) -> False :
  val seq1 = to-seq(xs)
  while not empty?(seq1) :
    println(next(seq1))

  val seq2 = to-seq(xs)
  while not empty?(seq2) :
    println(next(seq2))
```

Such a function would not work as expected when called with a `Seq`. Recall that calling `to-seq` on a `Seq` is a trivial operation that returns the `Seq` directly. Thus the first while loop will print out every item in the sequence, but the second while loop will not complete a single iteration as the sequence is already empty. The correct type signature for `print-all-twice` would be

```
defn print-all-twice (xs:Collection) -> False
```

`Seq` is a subtype of `Seqable`, but not a subtype of `Collection`. Thus the new type signature correctly disallows `print-all-twice` from being called with objects of type `Seq`.

Mandatory minimal implementation: `to-seq`.

IndexedCollection

An `IndexedCollection` represents a mutable collection containing a series of items, each of which is associated with an integer index. `deftype IndexedCollection<T> <: Lengthable & Collection<T>` All `IndexedCollections` are subtypes of `Collection` and support the `to-seq` operation. A default method for `to-seq` defined in terms of `length` and `get` is provided. For efficiency purposes, subtypes of `IndexedCollections` may provide a customized method of `to-seq`, but it is not required.

Mandatory minimal implementation: `length`, `get(IndexedCollection<T>, Int)`, `set(IndexedCollection<T`

get, set

Appropriate methods for getting and setting an item at a given index `i` must be provided for all subtypes. `get` retrieves the object at integer index `i` in the object `a`. `set` assigns the value `v` to the object `a` at integer index `i`. The index `i` must be non-negative and less than the length of `a`.

```
defmulti get<?T> (a:IndexedCollection<?T>, i:Int) -> T
defmulti set<?T> (a:IndexedCollection<?T>, i:Int, v:T) -> False
```

get

The following function returns a range of items in an `IndexedCollection`. The given range must be a *dense index range* with respect to the collection `a`. A default method defined in terms of `get` is provided, but subtypes may provide customized methods for efficiency purposes if desired.

```
defmulti get<?T> (a:IndexedCollection<?T>, r:Range) -> Collection<T>
```

set

The following function sets a range of indices within the collection to items taken sequentially from `vs`. The given range must be a *dense index range* with respect to the collection `a`. The sequence `vs` must contain at least as many items as indices being assigned. A default method defined in terms of `set` is provided, but subtypes may provide customized methods for efficiency purposes if desired.

```
defmulti set<?T> (a:IndexedCollection<?T>, r:Range, vs:Seqable<T>) ->
   False
```

map!

The following function iterates through the given collection and replaces each item with the result of calling `f` on the item. A default method defined in terms of `get` and `set` is provided, but subtypes may provide customized methods for efficiency purposes if desired.

```
defmulti map!<?T> (f: T -> T, xs:IndexedCollection<?T>) -> False
```

empty?

`empty?` returns `true` if the collection is empty or `false` otherwise.

```
defn empty? (v:IndexedCollection) -> True|False
```

reverse!

The following function reverses the order in which items appear in the collection.

```
defn reverse!<?T> (xs:IndexedCollection<?T>) -> False
```

in-reverse

The following function returns a sequence containing the items in the collection in reversed order. The original collection is unchanged.

```
defn in-reverse<?T> (xs:IndexedCollection<?T>) -> Seq<T>
```

OutputStream

An `OutputStream` represents a destination to which we can print characters, and write values. The most common output stream used in daily programming is the standard output stream which represents the user's terminal.

```
deftype OutputStream
```

Mandatory minimal implementation: `print(OutputStream, Char)`.

print

The fundamental operation for an `OutputStream` is the function for printing a specific value `x` to the output stream.

```
defmulti print (o:OutputStream, x) -> False
```

For all types in the core Stanza library, there is a default implementation of `print` for that type that prints each `Char` in its string representation to the output stream. Because of this, the only mandatory method that needs to be implemented by subtypes of `OutputStream` is that for `Char`.

```
defmethod print (o:OutputStream, c:Char) -> False
```

print-all

`print-all` prints all the item in the given sequence to the `OutputStream`.

```
defmulti print-all (o:OutputStream, xs:Seqable) -> False
```

There is a default method implemented for `print-all` that calls `print` on each item in the sequence. For efficiency purposes, users are free to provide customized versions of `print-all` for specific types.

println

`println` prints the item `x` to the `OutputStream` followed by the newline character.

```
defn println (o:OutputStream, x) -> False
```

println-all

`println-all` prints all items in the sequence `xs` to the `OutputStream` followed by the newline character.

```
defn println-all (o:OutputStream, xs:Seqable) -> False
```

STANDARD-OUTPUT-STREAM

This global value holds the standard output stream representing the user's terminal.

```
val STANDARD-OUTPUT-STREAM : OutputStream
```

STANDARD-ERROR-STREAM

This global value holds the standard error stream. It represents the user's terminal if error messages are not redirected or the error buffer otherwise.

```
val STANDARD-ERROR-STREAM : OutputStream
```

with-output-stream

The current output stream is, by default, the standard output stream. The following function will set the current output stream to `o` before calling the function `f` and then restore the current output stream afterwards.

```
defn with-output-stream<?T> (o:OutputStream, f: () -> ?T) -> T
```

current-output-stream

The current output stream may be retrieved with the following function.

```
defn current-output-stream () -> OutputStream
```

print to current OutputStream

The following functions behave identically to the versions that do not take an OutputStream argument. Instead they print to the current output stream.

```
defn print (x) -> False
defn println (x) -> False
defn print-all (xs:Seqable) -> False
defn println-all (xs:Seqable) -> False
```

InputStream

An `InputStream` represents a source from which we can read characters and values.

```
deftype InputStream
```

Mandatory minimal implementation: `get-char`, `get-byte`.

get-char

This multi reads a single character from the given input stream. `false` is returned if there are no more characters in the stream.

```
defn get-char (i:InputStream) -> Char|False
```

get-byte

The following function reads a single byte from the given input stream. `false` is returned if there are no more bytes in the stream.

```
defn get-byte (i:InputStream) -> Byte|False
```

STANDARD-INPUT-STREAM

This global value holds the standard input stream representing the user's terminal.

```
val STANDARD-INPUT-STREAM : InputStream
```

fill

The following multi reads characters continuously from the input stream and stores them into the given `CharArray` at the specified range. The range must be a *dense index range* with respect to `xs`. The number of characters read is returned.

```
defmulti fill (xs:CharArray, r:Range, i:InputStream) -> Int
```

A default method implemented in terms of `get-char` is provided, but for efficiency purposes, users are free to provide customized methods for subtypes of `InputStream`.

IndentedStream

An `IndentedStream` wraps over an `OutputStream` to provide the ability to automatically print indenting spaces as needed.

```
deftype IndentedStream <: OutputStream
```

`IndentedStream` is a subtype of `OutputStream` and can be used as a target for the `print` function. When asked to print a newline character, an `IndentedStream` will print the newline character followed by the number of spaces indicated during creation of the stream. For all other characters, an `IndentedStream` simply calls print on its wrapped `OutputStream`.

IndentedStream

This function creates an `IndentedStream` that indents `n` spaces by wrapping over the given `OutputStream`.

```
defn IndentedStream (o:OutputStream, n:Int) -> IndentedStream
```

do-indented

This function calls the function `f` with a new `IndentedStream` created by wrapping over the given output stream.

```
defn do-indented<?T> (f: IndentedStream -> ?T, o:OutputStream) -> T
```

indented

This function wraps over the current output stream with an `IndentedStream` and then calls the function `f`. The current output stream is restored afterwards.

```
defn indented<?T> (f: () -> ?T) -> T
```

FileOutputStream

A `FileOutputStream` represents an external file with an output stream interface to which we can write values.

```
deftype FileOutputStream <: OutputStream
```

A `FileOutputStream` is a subtype of `OutputStream` and can be used as a target for the `print` function.

FileOutputStream

These functions create a new `FileOutputStream` given the path to the file, and a boolean flag indicating, in the case that the file already exists, whether new characters should be appended to the end of the file or whether the file should be overwritten. If no `append?` flag is given, then by default, the file is overwritten.

```
defn FileOutputStream (filename:String, append?:True|False) ->
   FileOutputStream
defn FileOutputStream (filename:String) -> FileOutputStream
```

put

These functions write the following values as binary data following little endian conventions.

```
defn put (o:FileOutputStream, x:Byte) -> False
defn put (o:FileOutputStream, x:Int) -> False
defn put (o:FileOutputStream, x:Long) -> False
defn put (o:FileOutputStream, x:Float) -> False
defn put (o:FileOutputStream, x:Double) -> False
```

close

This function closes a `FileOutputStream`.

```
defn close (o:FileOutputStream) -> False
```

with-output-file

This function sets the given file as the current output stream before calling the function `f`. Afterwards, it restores the current output stream and closes the file.

```
defn with-output-file<?T> (file:FileOutputStream, f: () -> ?T) -> T
```

spit

This function prints the given object to the given file.

```
defn spit (filename:String, x) -> False
```

Pretty Printing

The following pretty printing functions are used for printing a value in a form that preserves its structure. For example, `String` objects are surrounded in quotes and non-printable characters are escaped. Numbers have suffixes to indicate their type. For the core types that make up an s-expression, values are written out in a form that can be read back in using the reader.

write

The write multi pretty prints the given object `x`. A default method is provided that simply calls `print`. Objects whose pretty printed representations differ from their printed representations should provide a custom method for `write`.

```
defmulti write (o:OutputStream, x) -> False
```

write-all

This function pretty prints every item in the sequence `xs` separated by spaces.

```
defn write-all (o:OutputStream, xs:Seqable) -> False
```

FileInputStream

A `FileInputStream` represents an external file with an input stream interface from which we can read values.

```
deftype FileInputStream <: InputStream
```

`FileInputStream` is a subtype of `InputStream` and supports the fundamental operations, `get-char` and `get-byte`, for reading a character or a byte from the stream.

FileInputStream

This function creates a `FileInputStream` given the name of the file.

```
defn FileInputStream (filename:String) -> FileInputStream
```

close

This function closes a `FileInputStream`.

```
defn close (i:FileInputStream) -> False
```

Reading Values

These functions read values as binary data following little endian conventions.

```
defn get-int (i:FileInputStream) -> False|Int
defn get-long (i:FileInputStream) -> False|Long
defn get-float (i:FileInputStream) -> False|Float
defn get-double (i:FileInputStream) -> False|Double
```

slurp

This function reads the entire contents of a file and returns the contents as a `String`.

```
defn slurp (filename:String) -> String
```

StringInputStream

A `StringInputStream` represents an input stream backed by an underlying string.

```
deftype StringInputStream <: InputStream & Lengthable
```

`StringInputStream` is a subtype of `InputStream` and implements appropriate methods for reading characters and bytes.

`StringInputStream` is a subtype of `Lengthable` and implements an appropriate method for the `length` multi that returns the number of characters left unread in the underlying string.

StringInputStream

This function creates a `StringInputStream` given the underlying string and the name of the file that it comes from. If no filename is given, then its default value is `"UnnamedStream"`.

```
defn StringInputStream (string:String, filename:String) ->
   StringInputStream
defn StringInputStream (string:String) -> StringInputStream
```

peek?

A `StringInputStream` allows for the characters beyond the immediate following character to be read. The following function returns the `i`'th next character in the stream, if one exists. If the index `i` is not given, then its default value is `0`.

```
defn peek? (s:StringInputStream, i:Int) -> False|Char
defn peek? (s:StringInputStream) -> False|Char
```

info

`StringInputStream` keeps track of the current position of its read marker. The following function returns the `FileInfo` representing the current position in the string.

```
defn info (s:StringInputStream) -> FileInfo
```

RandomAccessFile

A `RandomAccessFile` represents an external file that allows users to read and write at non-consecutive locations.

```
deftype RandomAccessFile
```

RandomAccessFile

This function creates a `RandomAccessFile` for the given filename and boolean flag for indicating whether the file should be writable. A `FileOpenException` is thrown if the file could not be opened.

```
defn RandomAccessFile (filename:String, writable:True|False) ->
   RandomAccessFile
```

close

This function flushes all pending writes and closes an open `RandomAccessFile`. A `FileCloseException` is thrown if the file could not be closed.

```
defn close (file:RandomAccessFile) -> False
```

writable?

This function returns `true` if the given file is writable, or false otherwise.

```
defn writable? (file:RandomAccessFile) -> True|False
```

length

This function returns the length in bytes for the given file.

```
defn length (file:RandomAccessFile) -> Long
```

set-length

This function sets the length in bytes of the given file to the given value. If the current length of the file is greater than the given length, then the file is truncated. If the current length is less than the given length, then the file is padded with undefined values until the given length. A `FileSetLengthException` is thrown if the operation is not successful.

```
defn set-length (file:RandomAccessFile, length:Long) -> False
```

seek

This function sets the read/write cursor of the file to the given position. A `FileSeekException` is thrown if the operation is not successful.

```
defn seek (file:RandomAccessFile, pos:Long) -> False
```

skip

This function increments the read/write cursor of the file by `pos` number of bytes. The cursor can be moved backwards towards the beginning of the file by providing a negative `pos`. A `FileSeekException` is thrown if the operation is not successful.

```
defn skip (file:RandomAccessFile, pos:Long) -> False
```

Reading Values

These functions read values as binary data following little endian conventions from the given file. The functions either return the value read, or `false` if it has reached the end of the file. A `FileReadException` is thrown if the operations are not successful.

```
defn get-byte (file:RandomAccessFile) -> Byte|False
defn get-char (file:RandomAccessFile) -> Char|False
defn get-int (file:RandomAccessFile) -> Int|False
defn get-long (file:RandomAccessFile) -> Long|False
defn get-float (file:RandomAccessFile) -> Float|False
defn get-double (file:RandomAccessFile) -> Double|False
```

Writing Values

These functions write values as binary data following little endian conventions into the given file. A `FileWriteException` is thrown if the operations are not successful.

```
defn put (file:RandomAccessFile, x:Byte) -> False
defn put (file:RandomAccessFile, x:Char) -> False
defn put (file:RandomAccessFile, x:Int) -> False
defn put (file:RandomAccessFile, x:Long) -> False
defn put (file:RandomAccessFile, x:Float) -> False
defn put (file:RandomAccessFile, x:Double) -> False
```

Reading Blocks

This function reads contiguous bytes from the given file and stores them into the range `r` in the byte array `a`. `r` must be a dense index range with respect to `a`. The number of bytes read is returned by the function. A `FileReadException` is thrown if the operation is not successful.

```
defn fill (a:ByteArray, r:Range, file:RandomAccessFile) -> Long
```

If the range `r` is not given, then the function stores the read bytes starting from the beginning of the array `a` and proceeds until the end.

```
defn fill (a:ByteArray, file:RandomAccessFile) -> Long
```

Writing Blocks

This function writes contiguous bytes from the range `r` in the given byte array `xs` into the given file. `r` must be a dense index range with respect to `xs`. A `FileWriteException` is thrown if the operation is not successful.

```
defn put (file:RandomAccessFile, xs:ByteArray, r:Range) -> False
```

If the range `r` is not given, then the function writes the entire array `xs` into the file.

```
defn put (file:RandomAccessFile, xs:ByteArray) -> False
```

Numbers

The following types make up Stanza's basic numerical types.

```
deftype Byte <: Equalable & Hashable & Comparable<Byte>
deftype Int <: Equalable & Hashable & Comparable<Int>
deftype Long <: Equalable & Hashable & Comparable<Long>
deftype Float <: Equalable & Hashable & Comparable<Float>
deftype Double <: Equalable & Hashable & Comparable<Double>
```

A `Byte` represents an 8-bit unsigned integer between 0 and 255 (inclusive). An `Int` represents a 32-bit signed integer. A `Long` represents a 64-bit signed integer. A `Float` represents a 32-bit real number in IEEE 754 encoding. A `Double` represents a 64-bit real number in IEEE 754 encoding.

Each of the numerical types are subtypes of `Equalable`, and hence support the `equal?` multi. Note that numerical values of different types are never defined to be equal to one another. Thus the `Int`, `0`, is not equal to the `Long`, `0L`.

Each of the numerical types are subtypes of `Comparable`, and can be compared against themselves.

Each of the numerical types are subtypes of `Hashable`, and supports the `hash` multi.

Integer Arithmetic

Integer numerical types support the standard arithmetic operations.

```
defn plus (x:Byte, y:Byte) -> Byte
defn minus (x:Byte, y:Byte) -> Byte
defn times (x:Byte, y:Byte) -> Byte
defn divide (x:Byte, y:Byte) -> Byte
defn modulo (x:Byte, y:Byte) -> Byte
```

```
defn plus (x:Int, y:Int) -> Int
defn minus (x:Int, y:Int) -> Int
defn times (x:Int, y:Int) -> Int
defn divide (x:Int, y:Int) -> Int
defn modulo (x:Int, y:Int) -> Int
```

```
defn plus (x:Long, y:Long) -> Long
defn minus (x:Long, y:Long) -> Long
defn times (x:Long, y:Long) -> Long
defn divide (x:Long, y:Long) -> Long
defn modulo (x:Long, y:Long) -> Long
```

Floating Point Arithmetic

Floating point numerical types support the all standard arithmetic operations except the modulo operation.

```
defn plus (x:Float, y:Float) -> Float
defn minus (x:Float, y:Float) -> Float
defn times (x:Float, y:Float) -> Float
defn divide (x:Float, y:Float) -> Float
```

```
defn plus (x:Double, y:Double) -> Double
defn minus (x:Double, y:Double) -> Double
defn times (x:Double, y:Double) -> Double
defn divide (x:Double, y:Double) -> Double
```

Note that arithmetic operations must be performed between values of the same numerical type. Users must manually convert values from one type to another if their types differ.

Signed Arithmetic

All numerical values except of type `Byte` support the additional negation and absolute value operations. `Byte` values are unsigned and hence do not support these operations.

```
defn negate (x:Int) -> Int
defn negate (x:Long) -> Long
defn negate (x:Double) -> Double
defn negate (x:Float) -> Float
```

```
defn abs (x:Int) -> Int
defn abs (x:Long) -> Long
defn abs (x:Double) -> Double
defn abs (x:Float) -> Float
```

Bitwise Operations

Integer numerical types support the standard bitwise operations.

```
defn shift-left (x:Byte, y:Byte) -> Byte
defn shift-right (x:Byte, y:Byte) -> Byte
defn bit-or (x:Byte, y:Byte) -> Byte
defn bit-and (x:Byte, y:Byte) -> Byte
defn bit-xor (x:Byte, y:Byte) -> Byte
defn bit-not (x:Byte) -> Byte
```

```
defn shift-left (x:Int, y:Int) -> Int
defn shift-right (x:Int, y:Int) -> Int
defn arithmetic-shift-right (x:Int, y:Int) -> Int
defn bit-or (x:Int, y:Int) -> Int
defn bit-and (x:Int, y:Int) -> Int
defn bit-xor (x:Int, y:Int) -> Int
defn bit-not (x:Int) -> Int
```

```
defn shift-left (x:Long, y:Long) -> Long
defn shift-right (x:Long, y:Long) -> Long
defn arithmetic-shift-right (x:Long, y:Long) -> Long
defn bit-or (x:Long, y:Long) -> Long
defn bit-and (x:Long, y:Long) -> Long
defn bit-xor (x:Long, y:Long) -> Long
defn bit-not (x:Long) -> Long
```

Numerical Limits

The maximum and minimum values for the integer numerical types are defined in the following global values.

```
val BYTE-MAX : Byte
val BYTE-MIN : Byte
val INT-MAX : Int
val INT-MIN : Int
val LONG-MAX : Long
val LONG-MIN : Long
```

Numerical Conversion

Numerical types can be converted from one type to another using the following functions.

```
defn to-byte (b:Byte) -> Byte
defn to-byte (i:Int) -> Byte
defn to-byte (l:Long) -> Byte
defn to-byte (f:Float) -> Byte
defn to-byte (d:Double) -> Byte
defn to-int (b:Byte) -> Int
defn to-int (i:Int) -> Int
defn to-int (l:Long) -> Int
defn to-int (f:Float) -> Int
defn to-int (d:Double) -> Int
defn to-long (b:Byte) -> Long
defn to-long (i:Int) -> Long
defn to-long (l:Long) -> Long
defn to-long (f:Float) -> Long
defn to-long (d:Double) -> Long
defn to-float (b:Byte) -> Float
defn to-float (i:Int) -> Float
defn to-float (l:Long) -> Float
defn to-float (f:Float) -> Float
defn to-float (d:Double) -> Float
defn to-double (b:Byte) -> Double
defn to-double (i:Int) -> Double
defn to-double (l:Long) -> Double
defn to-double (f:Float) -> Double
defn to-double (d:Double) -> Double
```

Integer types can be converted without loss of precision from a type with less bits to a type with more bits. When converting from a type with more bits to a type with less bits, the most significant bits are truncated.

When converting integer types to floating point types, the closest floating point number is returned.

bits

`bits` returns an integer type whose bit representation is equivalent to the IEEE754 bit representation of the given floating point number.

```
defn bits (x:Float) -> Int
defn bits (x:Double) -> Long
```

bits-as-float, bits-as-double

`bits-as-float` and `bits-as-double` returns a floating point number whose IEEE754 bit representation is equivalent to the bit representation of the given integer.

```
defn bits-as-float (x:Int) -> Float
defn bits-as-double (x:Long) -> Double
```

rand

The following function generates a random positive integer.

```
defn rand () -> Int
```

The following function generates a random positive integer that is guaranteed to be strictly less than n.

```
defn rand (n:Int) -> Int
```

The following function generates a random positive integer within the range specified by `r`. `r` must have a step size of one.

```
defn rand (r:Range) -> Int
```

ceil-log2

`ceil-log2` computes the ceiling of the base-2 logarithm of the integer `i`.

```
defn ceil-log2 (i:Int) -> Int
defn ceil-log2 (i:Long) -> Int
```

floor-log2

`floor-log2` computes the floor of the base-2 logarithm of the integer `i`.

```
defn floor-log2 (i:Int) -> Int
defn floor-log2 (i:Long) -> Long
```

next-pow2

`next-pow2` computes the smallest power of 2 that is greater than or equal to the given integer `i`. `i` cannot be negative.

```
defn next-pow2 (i:Int) -> Int
defn next-pow2 (i:Long) -> Long
```

prev-pow2

`prev-pow2` computes the largest power of 2 that is smaller than or equal to the given integer `i`. `i` cannot be negative.

```
defn prev-pow2 (i:Int) -> Int
defn prev-pow2 (i:Long) -> Long
```

sum

`sum` computes the sum of the given sequence of numbers.

```
defn sum (xs:Seqable<Int>) -> Int
defn sum (xs:Seqable<Long>) -> Long
defn sum (xs:Seqable<Float>) -> Float
defn sum (xs:Seqable<Double>) -> Double
```

product

`product` computes the product of the given sequence of numbers.

```
defn product (xs:Seqable<Int>) -> Int
defn product (xs:Seqable<Long>) -> Long
defn product (xs:Seqable<Float>) -> Float
defn product (xs:Seqable<Double>) -> Double
```

Boolean Types

Boolean values are represented using the types `True` and `False`.

```
deftype True <: Equalable
deftype False <: Equalable
```

Both `True` and `False` are subtypes of `Equalable` and hence support the `equal?` operation. Values of type `True` are only equal to other values of type `True`. Values of type `False` are only equal to other values of type `False`.

complement

`complement` returns the logical complement of its argument. If `a` is true, then it returns `false`, otherwise it returns `true`.

```
defn complement (a:True|False) -> True|False
```

Character Type

A `Char` represents a single byte ascii character.

```
deftype Char <: Equalable & Hashable & Comparable<Char>
```

`Char` is defined to be a subtype of `Equalable`, `Hashable`, and `Comparable`, and thus it supports the equality operator, the `hash` multi, and can be compared other values of type `Char`.

Two values of type `Char` are compared according to the numerical value of their ascii encoding.

digit?

`digit?` checks whether a given character is a digit character, representing one of the numerals between `0` and `9`.

```
defn digit? (c:Char) -> True|False
```

letter?

`letter?` checks whether a given character is either a lower or upper-case character, representing one of the letters between `'A'` and `'Z'` or between `'a'` and `'z'`.

```
defn letter? (c:Char) -> True|False
```

upper-case?

`upper-case?` checks whether a given character is an upper-case character, representing one of the letters between `'A'` and `'Z'`.

```
defn upper-case? (c:Char) -> True|False
```

lower-case?

`lower-case?` checks whether a given character is a lower-case character, representing one of the letters between `'a'` and `'z'`.

```
defn lower-case? (c:Char) -> True|False
```

lower-case

`lower-case` converts a given character to its lower-case representation if it is a letter, otherwise returns the original character.

```
defn lower-case (c:Char) -> Char
```

upper-case

`upper-case` converts a given character to its upper-case representation if it is a letter, otherwise returns the original character.

```
defn upper-case (c:Char) -> Char
```

Range

A `Range` represents a span of integers between a starting index and an optional ending index.

```
deftype Range <: Collection<Int> & Equalable
```

Range

The following function creates a range with a given starting index, an optional ending index, a step size, and a flag indicating whether the ending index is inclusive or exclusive.

```
defn Range (start:Int, end:Int|False, step:Int, inclusive?:True|False)
   -> Range
```

Users will typically use one of the macro forms for creating a range. The following form creates a range with the starting index `a`, exclusive ending index `b`, and step size `1`.

```
a to b
```

The keyword `by` can be used to provide a custom step size.

```
a to b by n
```

To create ranges with inclusive ending indices, use the keyword `through` instead of `to`.

```
a through b
a through b by n
```

`Range` is a subtype of `Collection` and implements an appropriate method for the `to-seq` multi. A range is viewed as a sequence of numbers starting from its given starting index, and proceeding until its ending index by the given step size. If the ending index is inclusive, then the sequence may contain the ending index. If the ending index is exclusive, then the sequence will not contain the ending index. If no ending index is given, then the sequence is infinite.

`Range` is a subtype of `Equalable` and two ranges are defined to be equal if they have equivalent starting and ending indices, step sizes, and inclusion flags.

Dense Index Ranges

For many of the core library functions, ranges are used to specify bounds within a `Lengthable` collection. For these ranges, the range argument must be a *dense index range* with respect to the given collection. A range is a *dense index range* with respect to a collection if it satisfies the following restrictions:

1. The step size of the range must be 1.
2. If the range is infinite, then the starting index must be non-negative and less than or equal to the length of the collection.
3. If the range is finite and inclusive, then the starting and ending indices must be non-negative and less than the length of the collection. The starting index must be less than or equal to the ending index.
4. If the range is finite and exclusive, then the starting and ending indices must be non-negative and less than or equal to the length of the collection. The starting index must be less than or equal to the ending index.

start

`start` returns the starting index of the given range.

```
defn start (r:Range) -> Int
```

end

`end` returns the ending index of the given range if there is one, or `false` otherwise.

```
defn end (r:Range) -> Int
```

step

`step` returns the step size of the given range.

```
defn step (r:Range) -> Int
```

inclusive?

`inclusive?` returns `true` if the ending index of the given range is inclusive, or `false` otherwise.

```
defn inclusive? (r:Range) -> True|False
```

map

`map` returns a list containing the result of applying the function `f` to each integer within the given range. It is a fatal error to call `map` on a range with infinite length.

```
defn map<?R> (f: Int -> ?R, r:Range) -> List<R>
```

String

A `String` represents an immutable collection of values of type `Char`.

```
deftype String <: Lengthable &
                Collection<Char>
   &
                Equalable &
                Hashable &
                Comparable<String
   >
```

`String` is a subtype of `Equalable` and hence supports the `equal?` operation. Two strings are equal if they have the same length and all of their characters are respectively equal.

`String` is a subtype of `Lengthable` and hence supports the `length` operation.

`String` is a subtype of `Hashable`, hence supports the `hash` operation and can be used as keys in a `HashTable`.

`String` is a subtype of `Comparable`, and two values of type `String` can be compared using the standard comparison operations according to their lexicographic ordering.

`String` is a subtype of `Collection`, and supports the `to-seq` operation for viewiewing the `String` as a sequence of values of type `Char`.

String

The following function creates a new `String` containing `n` copies of the `c` character.

```
defn String (n:Int, c:Char) -> String
```

The following function converts the given sequence of characters into a `String`.

```
defn String (cs:Seqable<Char>) -> String
```

get

`get` allows for retrieval of individual characters in a `String` by index. It is a fatal error to provide an index that is beyond the bounds of the string.

```
defn get (s:String, i:Int) -> Char
```

get

`get` allows for retrieval of a range of characters in a `String`. The given range must be a *dense index range* with respect to the given string.

```
defn get (s:String, r:Range) -> String
```

For example, assuming that `s` contains the string `"Hello World"`, here is what various calls to `get` will return:

```
s[1 to 4]        ; returns  "ell"
s[1 through 4]   ; returns  "ello"
s[1 to false]    ; returns  "ello World"
```

Parsing Numbers

The following functions convert the string representations of numbers to their numerical types.

```
defn to-byte (s:String) -> False|Byte
defn to-int (s:String) -> False|Int
defn to-long (s:String) -> False|Long
defn to-float (s:String) -> False|Float
defn to-double (s:String) -> False|Double
```

Each of the above functions returns `false` if the given string cannot be converted to the appropriate type, either because of an incorrectly formatted string, or because the resulting number cannot be represented using the appropriate number of bits.

to-string

The following multi converts a given object to its string representation.

```
defmulti to-string (x) -> String
```

A default method defined in terms of `print` is provided to convert arbitrary objects to their string representation. User defined types are advised to rely upon this default method and *not* provide their own method for `to-string`. To provide custom printing behaviour for a type, users should provide methods for `print` instead.

String Interpolation

The `modulo` operator creates a printable object given a format string and a sequence of arguments.

```
defn modulo (format:String, args:Seqable) -> Printable
```

The characters in the format string are printed one by one, where occurrences of a splicing operator prints the next item in the sequence. The available splicing operators are:

1. `%_` : Prints the next item in the sequence.
2. `%*` : Assumes the next item in the sequence is `Seqable`, and prints all elements in the item.
3. `%,` : Assumes the next item in the sequence is `Seqable`, and prints all elements in the item separated by commas.
4. `%~` : Pretty prints the next item in the sequence using the `write` function.
5. `%@` : Assumes the next item in sequence is `Seqable`, and pretty prints all elements in the item separated by spaces.
6. `%%` : Prints the percentage sign character.

For example, the following command will print "`On Tuesdays, Thursdays, Wednesdays, Patrick goes for a`

```
val days = ["Tuesdays", "Thursdays", "Wednesdays"]
val name = "Patrick"
val activity = "walk"
println("On %,, %_ goes for a %_." % [days, name, activity])
```

Note that the `modulo` function does *not* return a `String`. The `to-string` function may be used to convert the printable object into a string.

```
val days = ["Tuesdays", "Thursdays", "Wednesdays"]
val name = "Patrick"
val activity = "walk"
to-string("On %,, %_ goes for a %_." % [days, name, activity])
```

matches?

`matches?` returns `true` if the string `b` can be found at position `start` in the string `a`, or `false` otherwise.

```
defn matches? (a:String, start:Int, b:String) -> True|False
```

prefix?

`prefix?` returns `true` if the string `a` starts with the given `prefix`, or `false` otherwise.

```
defn prefix? (s:String, prefix:String) -> True|False
```

suffix?

`suffix?` returns `true` if the string `a` ends with the given suffix, or `false` otherwise.

```
defn suffix? (s:String, suffix:String) -> True|False
```

empty?

`empty?` returns `true` if the string contains no characters.

```
defn empty? (s:String) -> True|False
```

append

append returns a new string resulting from concatenating string `b` to the end of string `a`.

```
defn append (a:String, b:String) -> String
```

append-all

`append-all` returns a new string resulting from concatenating together all strings in a sequence.

```
defn append-all (xs:Seqable<String>) -> String
```

string-join

`string-join` returns the string resulting from printing out each item in the sequence `xs`. If the argument `j` is given, then each item in the sequence is separated by `j`.

```
defn string-join (xs:Seqable) -> String
defn string-join (xs:Seqable, j) -> String
```

index-of-char

`index-of-char` searches the string `s`, within the bounds indicated the range `r`, for the first occurrence of the character `c`. If `c` is found, then its index is returned, otherwise `false` is returned. If a range is not given, then by default the function searches through the entire string. The given range must be a *dense index range* with respect to the string.

```
defn index-of-char (s:String, r:Range, c:Char) -> False|Int
defn index-of-char (s:String, c:Char) -> False|Int
```

index-of-chars

`index-of-chars` searches the string `s`, within the bounds indicated by the range `r`, for the first occurrence of the substring `b`. If it is found, then its index is returned, otherwise `false` is returned. If a range is not given, then by default the function searches through the entire string. The given range must be a *dense index range* with respect to the string.

```
defn index-of-chars (a:String, r:Range, b:String) -> False|Int
defn index-of-chars (a:String, b:String) -> False|Int
```

last-index-of-char

`last-index-of-char` searches the string `s`, within the bounds indicated the range `r`, for the last occurrence of the character `c`. If `c` is found, then its index is returned, otherwise `false` is returned. If a range is not given, then by default the function searches through the entire string. The given range must be a *dense index range* with respect to the string.

```
defn last-index-of-char (s:String, r:Range, c:Char) -> False|Int
defn last-index-of-char (s:String, c:Char) -> False|Int
```

last-index-of-chars

`last-index-of-chars` searches the string `s`, within the bounds indicated by the range `r`, for the last occurrence of the substring `b`. If it is found, then its index is returned, otherwise `false` is returned. If a range is not given, then by default the function searches through the entire string. The given range must be a *dense index range* with respect to the string.

```
defn last-index-of-chars (a:String, r:Range, b:String) -> False|Int
defn last-index-of-chars (a:String, b:String) -> False|Int
```

replace

`replace` returns the result of replacing every occurrence of the character `c1` in the string `s` with the character `c2`.

```
defn replace (s:String, c1:Char, c2:Char) -> String
```

replace

`replace` returns the result of replacing every occurrence of the substring `s1` in the string `str` with the substring `s2`.

```
defn replace (str:String, s1:String, s2:String) -> String
```

split

`split` returns a lazily computed sequence resulting from splitting the string `str` at occurrences of the substring `s`. If the argument `n` is given, then the resulting sequence is restricted to contain at most `n` strings, where the last string contains the unsplit remainder of `str`.

```
defn split (str:String, s:String) -> Seq<String>
defn split (str:String, s:String, n:Int) -> Seq<String>
```

The following relation is guaranteed return `true`.

```
string-join(split(str, s), s) == str
```

lower-case

`lower-case` the result of converting all letters contained within the string `s` to lower-case.

```
defn lower-case (s:String) -> String
```

upper-case

`upper-case` the result of converting all letters contained within the string `s` to upper-case.

```
defn upper-case (s:String) -> String
```

trim

`trim` returns the result of removing all leading and trailing characters that satisfy the function `pred` from the string `s`.

```
defn trim (pred: Char -> True|False, s:String) -> String
```

If `pred` is not given, then `trim`, by default, removes all leading and trailing whitespace characters.

StringBuffer

A `StringBuffer` is an extendable array for holding characters.

```
deftype StringBuffer <: IndexedCollection<Char> & OutputStream
```

`StringBuffer` is a subtype of a `IndexedCollection` and implements appropriate methods for retrieving its length, and getting and setting characters by index.

When setting a character at an index `i`, if `i` is less than the length of the buffer, then the character currently at index `i` will be overwritten. If `i` is equal to the length of the buffer, then `i` will be added to the end of the buffer, and the buffer's length will increase by 1.

`StringBuffer` is a subtype of `OutputStream` and can be used as a target for the `print` function.

StringBuffer

This function creates a `StringBuffer` with the initial capacity `n`. The capacity of a StringBuffer is the number of characters it may hold before it undergoes resizing.

```
defn StringBuffer (n:Int) -> StringBuffer
```

If no capacity is provided, then the default capacity is 32.

```
defn StringBuffer () -> StringBuffer
```

add

This function will add the character c to the end of the buffer.

```
defn add (s:StringBuffer, c:Char) -> False
```

add-all

This function will add all the characters in the sequence cs to the end of the buffer.

```
defn add-all (s:StringBuffer, cs:Seqable<Char>) -> False
```

clear

This function will clear all characters in the StringBuffer.

```
defn clear (s:StringBuffer) -> False
```

Array

Arrays are the most fundamental IndexedCollection in Stanza, representing a finite series of items with constant time access to elements by index.

```
deftype Array<T> <: IndexedCollection<T> & Equalable
```

Array is a subtype of IndexedCollection and hence supports the fundamental get, set, and length operations.

Array is a subtype of Equalable and two arrays are defined to be equal only if they are the same array.

Array

This function creates an Array with length n for holding values of type T. It is a fatal error to read from an index that has not been initialized, and n cannot be negative.

```
defn Array<T> (n:Int) -> Array<T>
```

This function creates an Array with length n for holding values of type T with each index initialized to the item x. n cannot be negative.

```
defn Array<T> (n:Int, x:T) -> Array<T>
```

to-array

This function converts a sequence of items of type `T` to an array.

```
defn to-array<T> (xs:Seqable<T>) -> Array<T>
```

map

This function creates a new array where each item is initialized to the result of calling the function `f` on each item in the given array `xs`. Note that the element type of the result array, `R`, must be provided explicitly.

```
defn map<R,?T> (f:T -> R, xs:Array<?T>) -> Array<R>
```

CharArray

`CharArray` objects are specialized arrays designed specifically for efficiently storing `Char` objects.

```
deftype CharArray <: Array<Char>
```

`CharArray` objects behave equivalently to `Array<Char>` objects.

CharArray

This function creates a `CharArray` of length `n`, with each index initialized to the space character.

```
defn CharArray (n:Int) -> CharArray
```

This function creates a `CharArray` of length `n`, with each index initialized to the given character `x`.

```
defn CharArray (n:Int, x:Char) -> CharArray
```

get-chars

This function allows for efficient retrieval of characters from a `CharArray`. The range `r` must be a *dense index range* with respect to `cs`.

```
defn get-chars (cs:CharArray, r:Range) -> String
```

set-chars

This function allows for efficient assignment of characters in a `CharArray`. The range `r` must be a *dense index range* with respect to `cs`.

```
defn set-chars (cs:CharArray, r:Range, s:String) -> False
```

ByteArray

`ByteArray` objects are specialized arrays designed specifically for efficiently storing `Byte` objects.

```
deftype ByteArray <: Array<Byte>
```

`ByteArray` objects behave equivalently to `Array<Byte>` objects.

ByteArray

This function creates a `ByteArray` of length `n`, with each index initialized to `0Y`.

```
defn ByteArray (n:Int) -> ByteArray
```

This function creates a `ByteArray` of length `n`, with each index initialized to the given byte `x`.

```
defn ByteArray (n:Int, x:Byte) -> ByteArray
```

Tuple

A `Tuple` represents a finite and immutable series of items each with an integer index.

```
deftype Tuple<T> <: Lengthable & Collection<T> & Equalable & Hashable &
    Comparable<Tuple<T>>
```

`Tuple` is a subtype of `Equalable` and two tuples are defined to be equal if they have the same length and all of their components are equal. It is an error to check whether two tuples are equal if they contain elements that are not `Equalable`.

`Tuple` is a subtype of `Collection` and implements an appropriate method for the `to-seq` multi to access the elements of the tuple as a sequence.

`Tuple` is a subtype of `Hashable` and computes its hash by combining the hashes of all of its items. It is a fatal error to call `hash` on a tuple containing items that are not `Hashable`.

`Tuple` is a subtype of `Comparable` and implements an appropriate method for the `compare` multi that compares two tuples by their lexicographic order. It is a fatal error to compare two tuples containing elements that are not `Comparable` with each other.

`Tuple` is a subtype of `Lengthable` and its length can be queried by calling the `length` multi.

Tuple

`Tuple` creates a tuple of length `n` with each element initialized to `x`.

```
defn Tuple<?T> (n:Int, x:?T) -> Tuple<T>
```

to-tuple

`to-tuple` converts a finite sequence of items into a tuple. The function will not terminate if given an infinite sequence.

```
defn to-tuple<?T> (xs:Seqable<?T>) -> Tuple<T>
```

get

`get` will return the item at index `i` within the tuple `x`. `i` must be within the bounds of the range.

```
defn get<?T> (x:Tuple<?T>, i:Int) -> T
```

get

`get` will return a new `Tuple` containing the items spanned by the given range. The range must be a *dense index range* with respect to the tuple.

```
defn get<?T> (xs:Tuple<?T>, r:Range) -> Tuple<T>
```

map

`map` returns a new tuple containing the result of calling the function `f` on each item in the tuple `xs`.

```
defn map<?T,?R> (f:T -> ?R, xs:Tuple<?T>) -> Tuple<R>
```

empty?

`empty?` returns `true` if the tuple `t` is empty, otherwise it returns `false`.

```
defn empty? (t:Tuple) -> True|False
```

List

A List represents an immutable linked list of items.

```
deftype List<T> <: Lengthable & Collection<T> & Equalable & Hashable &
   Comparable<List<T>>
```

`List` is a subtype of `Lengthable` and its length can be queried by calling the `length` multi. Note that the length of a list is not cached, and computing the length of a list takes time proportional to the length of the list.

`List` is a subtype of `Collection` and implements an appropriate method for the `to-seq` multi for viewing the list as a sequence.

`List` is a subtype of `Equalable` and two lists are defined to be equal if they have the same length and all of their components are equal. It is a fatal error to check whether two lists are equal if they contain elements that are not `Equalable`.

`List` is a subtype of `Hashable` and implements an appropriate method for the `hash` multi. The hash is computed by combining the hashes of all the items in the list. It is a fatal error to call hash on a list containing elements that are not `Hashable`.

`List` is a subtype of `Comparable` and implements an appropriate method for the `compare` multi. Two lists are compared according to their lexicographic ordering. It is a fatal error to compare two lists that contain elements that are not `Comparable` to each other.

List

The following functions create lists with no, one, two, three, or four elements respectively. The `to-list` function can be used to create lists with more than four elements.

```
defn List () -> List<Void>
defn List<?T> (x:?T) -> List<T>
defn List<?T> (x:?T, y:?T) -> List<T>
defn List<?T> (x:?T, y:?T, z:?T) -> List<T>
```

```
defn List<?T> (w:?T, x:?T, y:?T, z:?T) -> List<T>
```

cons

`cons` (short for "construct") creates new lists by appending one, two, or three elements to the head of an existing list. The `append` function can be used to append more than three elements to a list.

```
defn cons<?T> (x:?T, t:List<?T>) -> List<T>
defn cons<?T> (x:?T, y:?T, t:List<?T>) -> List<T>
defn cons<?T> (x:?T, y:?T, z:?T, t:List<?T>) -> List<T>
```

to-list

`to-list` creates a new list containing the items in the sequence `xs`. The function will not terminate if given an infinite sequence.

```
defn to-list<?T> (xs:Seqable<?T>) -> List<T>
```

head

`head` returns the head of a list. It is a fatal error to call `head` on an empty list.

```
defmulti head<?T> (x:List<?T>) -> T
```

tail

`tail` returns the tail of a list. It is a fatal error to call `tail` on an empty list.

```
defmulti tail<?T> (x:List<?T>) -> T
```

empty?

`empty?` returns `true` if the given list is empty or `false` otherwise.

```
defmulti empty? (x:List) -> True|False
```

get

`get` returns the `i`'th element in the list `xs`. `i` must be within the bounds of `xs`.

```
defn get<?T> (xs:List<?T>, i:Int) -> T
```

headn

`headn` returns a list containing the `n` first elements in the list `l`. `n` must be less than or equal to the length of the list.

```
defn headn<?T> (l:List<?T>, n:Int) -> List<T>
```

tailn

`tailn` returns the list after removing the first `n` elements from the head of the list `l`. `n` must be less than or equal to the length of the list.

```
defn tailn<?T> (l:List<?T>, n:Int) -> List<T>
```

reverse

`reverse` returns a new list containing the items in `xs` in reversed order.

```
defn reverse<?T> (xs:List<?T>) -> List<T>
```

in-reverse

`in-reverse` returns a sequence containing the items in the list in reverse order.

```
defn in-reverse<?T> (xs:List<?T>) -> Seq<T>
```

last

`last` returns the last element in the list. The list must not be empty.

```
defn last<?T> (xs:List<?T>) -> T
```

but-last

`but-last` returns a list containing all elements in `xs` except the last. `xs` must not be empty.

```
defn but-last<?T> (xs:List<?T>) -> List<T>
```

append

`append` returns the list resulting from appending all items from `xs` to the beginning of the list `ys`.

```
defn append<?T> (xs:Seqable<?T>, ys:List<?T>) -> List<T>
```

append-all

`append-all` returns the list resulting from appending together all lists in the sequence `xs`.

```
defn append-all<?T> (xs:Seqable<List<?T>>) -> List<T>
```

transpose

`transpose` returns a transposed version of the input list `xs`. The first element in the result list is a list containing the first elements in each list within `xs`. The second element in the result list is a list containing the second elements in each list within `xs`. The third element in the result list is a list containing the third elements in each list within `xs`, et cetera. The result list has the same length as the shortest list within xs.

```
defn transpose<?T> (xs:List<List<?T>>) -> List<List<T>>
```

map

`map` returns a new list consisting of the results of applying the function `f` to each element in the input list `xs`.

```
defn map<?T,?R> (f: T -> ?R, xs:List<?T>) -> List<R>
```

map

The following functions returns a new list consisting of the results of applying the function `f` to each pair (or triplet) of elements in the given input lists (or sequences). The result list has the same length as the shortest input list or sequence.

```
defn map<?T,?S,?R> (f: (T,S) -> ?R,
  xs:List<?T>,
  ys:Seqable<?S>) -> List<R>

defn map<?T,?S,?U,?R> (f: (T,S,U) -> ?R,
  xs:List<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> List<R>
```

seq-append

`seq-append` returns a new list by calling the function `f` on every item in the sequence `xs`, and then appending together all of the resultant lists.

```
defn seq-append<?T,?R> (f: T -> List<?R>, xs:Seqable<?T>) -> List<R>
```

seq-append

The following functions returns a new list by calling the function `f` on each pair (or triplet) of items in the input sequences, and then appending together all of the resultant lists.

```
defn seq-append<?T,?S,?R> (f: (T,S) -> List<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> List<R>

defn seq-append<?T,?S,?U,?R> (f: (T,S,U) -> List<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> List<R>
```

FileInfo

A `FileInfo` object contains position information about a specific point in a text file.

```
deftype FileInfo <: Equalable & Hashable & Comparable<FileInfo>
```

`FileInfo` is a subtype of `Equalable` and two `FileInfo` objects are equal when their filenames, lines, and columns are equal.

`FileInfo` is a subtype of `Hashable` and implements an appropriate method for the `hash` multi by combining the hashes of its filename, line, and column information.

`FileInfo` is a subtype of `Comparable` and implements an appropriate method for the `compare` multi. Two `FileInfo` objects are compared by ordering their filenames, followed their line index, followed by their column index.

FileInfo

`FileInfo` creates a `FileInfo` from the given filename, line, and column information.

```
defn FileInfo (f:String, l:Int, c:Int) -> FileInfo
```

filename

`filename` retrieves the filename of the given `FileInfo` object.

```
defn filename (i:FileInfo) -> String
```

line

`line` retrieves the line number of the given `FileInfo` object.

```
defn line (i:FileInfo) -> Int
```

column

`column` retrieves the column number of the given `FileInfo` object.

```
defn column (i:FileInfo) -> Int
```

Token

A `Token` represents a value that has been associated with a `FileInfo` object indicating the position from which it originated in a text file.

```
deftype Token <: Equalable
```

A `Token` is a subtype of `Equalable` and two tokens are considered equal if its `FileInfo` objects are equal and the wrapped values are equal. It is a fatal error to check whether two `Token` objects are equal if their wrapped values are not `Equalable`.

Token

This function creates a `Token` from the given value and its associated `FileInfo` object.

```
defn Token (item, info:FileInfo) -> Token
```

item

This function retrieves the wrapped value in a `Token` object.

```
defn item (t:Token) -> ?
```

info

This function retrieves the associated FileInfo object in a `Token` object.

```
defn info (t:Token) -> FileInfo
```

unwrap-token

This function returns the possibly wrapped value in a token by recursively unwrapping the value if it is a token.

```
defn unwrap-token (t) -> ?
```

unwrap-all

This function recursively calls `unwrap-token` on all tokens and lists in the given value. This function is commonly used to strip away all file information from an s-expression.

```
defn unwrap-all (t) -> ?
```

KeyValue

A `KeyValue` pair represents a pairing of a key object of type `K` with a value object of type `V`. It is typically used to represent entries in datastructures where values are looked up according to a given key.

```
deftype KeyValue<K,V>
```

Typically, users will use the following macro form for creating KeyValue pairs.

```
k => v
```

KeyValue

This function creates a `KeyValue` pair from the given key and value objects.

```
defn KeyValue<?K,?V> (k:?K, v:?V) -> KeyValue<K,V>
```

key

This function retrieves the key object in a `KeyValue` object.

```
defn key<?K> (kv:KeyValue<?K,?>) -> K
```

value

This function retrieves the value object in a `KeyValue` object.

```
defn value<?V> (kv:KeyValue<?,?V>) -> V
```

Symbols

`Symbols` are used to represent identifiers in s-expressions, and can be compared against each other efficiently and used as keys in hashtables.

```
deftype Symbol <: Equalable & Hashable
```

Symbols are divided into two categories: symbols created from strings (`StringSymbol`), and uniquely generated symbols (`GenSymbol`).

```
deftype StringSymbol <: Symbol
deftype GenSymbol <: Symbol
```

`Symbol` is a subtype of `Equalable` and two symbols can be efficiently compared to see whether they are equal. Two symbols created from strings are equal if the strings they are created from are equal. Two uniquely generated symbols are equal if and only if they were created by the same call to `gensym`.

`Symbol` is a subtype of `Hashable` and implements an appropriate method for the `hash` multi. For symbols constructed from strings, the hash is computed from its name. For generated symbols, the hash is computed from its id.

to-symbol

This function constructs a symbol from the given value by converting it to a string using `to-string` and then creating a symbol from the resulting string. If `x` is already a symbol then it is returned directly.

```
defn to-symbol (x) -> Symbol
```

symbol-join

This function constructs a symbol from the given sequence by converting it to a string using `string-join` and then creating a symbol from the resulting string.

```
defn symbol-join (xs:Seqable) -> Symbol
```

gensym

This function generates a unique symbol given an object `x` whose string representation is used as its name. The resulting symbol is guaranteed to be equal to no other symbol currently live in the program.

```
defn gensym (x) -> Symbol
```

If no object `x` is provided, then the default name is the string "$gen".

```
defn gensym () -> Symbol
```

name

This function retrieves the string that the symbol was constructed with.

```
defn name (x:Symbol) -> String
```

id

This function retrieves the identifier associated with a generated symbol. Every generated symbol is guaranteed to be associated with a unique identifier.

```
defn id (x:GenSymbol) -> Int
```

qualified?

A qualified symbol is a symbol created from a string containing a `'/'` character. In Stanza, qualified symbols are used to indicate a package qualified identifier. The following function returns `true` if the given symbol is a qualified symbol, or `false` otherwise.

```
defn qualified? (a:Symbol) -> True|False
```

qualifier

This function splits a qualified symbol into a 2 element tuple containing its qualifier and its unqualified name. If the argument is not a qualified symbol, then the qualifier is `false`.

```
defn qualifier (a:Symbol) -> [False|Symbol, Symbol]
```

String Representation

Note that two unique symbols, which are not equal to each other, may still have equivalent string representations. For example, a generated symbol with the name `"x"` and the id `253` has the same string representation as the symbol created from the string `"x253"`.

A typical method of converting a list of unique symbols into a list of unique strings is the following:

1. For `StringSymbol` objects, directly use their name as their string representation.
2. For `GenSymbol` objects, choose a name that is guaranteed not to be a prefix of any `StringSymbol`, then append the name to the id of the symbol. This name is typically chosen in an application-specific manner.

Maybe

A `Maybe` object is used to indicate the presence or absence of an object.

```
deftype Maybe<T> <: Equalable & Comparable<Maybe<T>>
```

The `One` subtype of `Maybe` indicates the presence of an object of type `T`.

```
deftype One<T> <: Maybe<T>
```

The `None` subtype of `Maybe` indicates the absence of an object.

```
deftype None <: Maybe<Void>
```

Maybe is a subtype of Equalable. Two None objects are always equal. Two One objects are equal if their wrapped values are equal. It is an error to check whether two Maybe objects are equal if they wrap over values that are not Equalable.

Maybe is a subtype of Comparable and implements an appropriate method for the compare multi. None objects are always less than One objects. Two One objects are compared by comparing their wrapped values. It is an error to compare two Maybe objects if they wrap over values that are not Comparable.

One

This function wraps up the value x in a One object.

```
defn One<?T> (x:?T) -> One<T>
```

None

This function creates a new None object.

```
defn None () -> None
```

value

This function will retrieve the wrapped value in a One object.

```
defn value<?T> (x:One<?T>) -> T
```

value?

This function will retrieve the wrapped value in a Maybe object if one exists, otherwise it returns the default value.

```
defn value?<?T> (x:Maybe<?T>, default:?T) -> T
```

If no default value is given, then the default value is false.

```
defn value?<?T> (x:Maybe<?T>) -> T|False
```

value!

This function assumes that the given `Maybe` object is a `One` object and returns its wrapped value.

```
defn value!<?T> (x:Maybe<?T>) -> T
```

empty?

This function returns `true` if the given `Maybe` object contains no wrapped value (i.e. it is a `None` object).

```
defn empty? (x:Maybe) -> True|False
```

Exception

An `Exception` object is used to represent the conditions under which an exceptional behaviour as occurred.

```
deftype Exception
```

Exception

Users are encouraged to create their own subtypes of `Exception` to represent different types of exceptional behaviour. However, the following function may be used to create a generic `Exception` object with the given error message.

```
defn Exception (msg) -> Exception
```

throw

This function throws the given exception object to the current exception handler. The return type of `Void` indicates that `throw` never returns.

```
defn throw (e:Exception) -> Void
```

with-exception-handler

This function runs the `body` function after installing the `handler` function as the current exception handler. If the body runs without throwing an exception, then its result is returned. Otherwise the result of the exception handler is returned.

```
defn with-exception-handler<?T> (body: () -> ?T, handler: Exception ->
   ?T) -> T
```

with-finally

This function runs the body function ensuring that the given `finally` function is run immediately afterwards. If the body runs to completion, then its result is returned after running the finally function. Otherwise, the finally function is run before execution exits the scope. Note that once the body function has started, execution is not allowed to exit and then re-enter the scope.

```
defn with-finally<?T> (body: () -> ?T, finally: (True|False) -> ?) -> T
```

try-catch-finally

Users will not typically call `with-exception-handler` or `with-finally` directly and instead use the standard macro forms. The following form runs the given body with a handler for catching `MyException` objects. The code within the optional finally block is ensured to run whether or not the body runs to completion.

```
try :
  body code
catch (e:MyException) :
  exception handling code
finally :
  finalization code
```

Fatal Errors

A fatal error occurs when a program somehow enters an illegal state that indicates an error has occurred in the program. It is impossible for a program to recover from a fatal error, and thus fatal errors do not occur in correct programs.

fatal

This function will print out the given error message to the screen, display a stack trace allowing users to find the source of the error, and immediately terminate the program.

```
defn fatal (msg) -> Void
```

On the highest (and unsafe) optimization settings, Stanza assumes that your program is correct and hence no calls to `fatal` ever occurs. For example, on the highest optimization settings, the following code:

```
if n < 0 :
  fatal("n should not be negative")
else :
  f()
```

is allowed to be optimized to the following:

```
f()
```

as Stanza assumes that your program is correct, and therefore it is impossible for execution to enter the consequent branch of the if expression.

Attempt and Failure

The attempt and failure functions provide a convenient non-local exit for users.

fail

The `fail` function exits from the current attempt scope and calls the current failure handler.

```
defn fail () -> Void
```

with-attempt

This function calls the `conseq` function after installing the `alt` function as the current failure handler. If the `conseq` function runs to completion then its result is returned. Otherwise, if `fail` is called during execution of `conseq`, the `alt` function is called and its result is returned.

```
defn with-attempt<?T> (conseq: () -> ?T, alt: () -> ?T) -> T
```

attempt-else

Users will not typically call `with-attempt` directly and instead use the standard macro forms. The following form runs the given body with a failure handler that executes when fail is called.

```
attempt :
  body code
else :
  failure code
```

If no `else` branch is provided, then a default `else` branch is provided that simply returns `false`.

Labeled Scopes

Labeled scopes provide the ability to exit early from a block of code.

LabeledScope

This function executes the `thunk` function in a new labeled scope. The `thunk` function is passed an exit function that, when called, will immediately exit `thunk`. If the `thunk` runs to completion without ever calling the exit function then its result is returned. If the exit function is called during execution of `thunk` then the argument passed to the exit function is returned.

```
defn LabeledScope<T> (thunk: (T -> Void) -> T) -> T
```

label

Users will typically use the standard macro form for creating labeled scopes instead of directly calling `LabeledScope`. The following code will return the first prime number between 100 and 200.

```
label<Int> return :
  for i in 100 to 200 do :
    if prime?(i) :
      return(i)
  fatal("No prime found!")
```

Generators

Generators provide the ability to execute a function in a new coroutine and return its results in the form of a sequence.

Generator

This function executes the `thunk` function in a new coroutine and returns a sequence. The `thunk` function is passed two functions as arguments. The first argument is the yield function, which takes a single argument. During execution of `thunk`, calls to the yield function will suspend the coroutine, and its argument will be included in the result sequence. The second argument is the `break` function, which takes either zero or one argument. During execution of `thunk`, a call to the break function with no arguments will immediately close the coroutine and end the result sequence. A call to the break function with one argument will immediately close the coroutine, include the argument as the last item in the result sequence, and then end the result sequence.

```
defn Generator<T> (thunk : (T -> False, (T -> Void)&(() -> Void)) -> ?)
    -> Seq<T>
```

generate

Users will typically use the standard macro form for creating generators instead of directly calling `Generator`. The following code creates a sequence containing the first 100 prime numbers.

```
generate<Int> :
  var n = 0
  for i in 2 to false do :
    if prime?(i) :
      n = n + 1
      if n == 100 : break(i)
      else : yield(i)
```

Coroutine

Coroutines provide the ability to execute a function in a context that can be saved and resumed later. This ability is most often used to execute a piece of code concurrently with the main program. A coroutine for which objects of type `I` are sent to its wrapped function, and for which objects of type `O` are sent back from its wrapped function is represented by the following type.

```
deftype Coroutine<I,O> <: Equalable
```

`Coroutine` is a subtype of `Equalable`. Two coroutines are equal if they were created by the same call to `Coroutine`.

Coroutine

This function creates a coroutine for which objects of type `I` are sent to its wrapped function, and for which objects of type `O` are sent back from its wrapped function. The argument represents the wrapped function of the coroutine.

```
defn Coroutine<I,O> (enter: (Coroutine<I,O>, I) -> O) -> Coroutine<I,O>
```

The first argument to the wrapped function is the coroutine that is created. The second argument to the wrapped function is the argument passed in the first call to `resume` on the coroutine. The result of the function `enter` is the last value sent back from the wrapped function.

resume

This function transfers control into a coroutine, and sends it the value `x`. The return value of `resume` is the value that is sent back by the coroutine.

```
defn resume<?I,?O> (c:Coroutine<?I,?O>, x:I) -> O
```

suspend

This function transfers control out of a coroutine, and back to the main program. The value `x` is sent back to the main program. The return value of `suspend` is the value that is sent to the coroutine by the next call to `resume`.

```
defn suspend<?I,?O> (c:Coroutine<?I,?O>, x:O) -> I
```

break

This function transfers control flow out of a coroutine, back to the main program, and closes the coroutine. The value `x` is sent back to the main program. `break` has no return value as the coroutine is closed and cannot be resumed afterwards.

```
defmulti break<?O> (c:Coroutine<?,?O>, x:O) -> Void
```

close

This function closes a coroutine, and disallows any further calls to `resume` on the coroutine.

```
defn close (c:Coroutine) -> False
```

active?

This function returns `true` if the given coroutine is currently running. Only active coroutines can be suspended or broken from.

```
defn active? (c:Coroutine) -> True|False
```

open?

This function returns `true` if the given coroutine is not currently running and open to be resumed. Only open coroutines can be resumed.

```
defn open? (c:Coroutine) -> True|False
```

Dynamic Wind

Consider the following code which attempts to ensure that the global variable `X` is set to the value 42 while the function `f` is running, and for `X` to be restored after `f` is done.

```
val old-X = X
X = 42
f()
X = old-X
```

However, because of coroutines it is possible for the above code to finish during the call to `f`, (for example, if `f` throws an exception) and thus leave `X` unrestored. The `dynamic-wind` function solves this problem by allowing users to perform operations upon entering or leaving a scope.

dynamic-wind

`dynamic-wind` takes a `body` function accompanied by an optional entering function, `in`, and an exiting function, `out`. `dynamic-wind` performs a call to `body` while ensuring that `in` is called every time execution enters the scope of `body`, and that `out` is called every time execution exits the scope of `body`. The exiting function takes a single boolean argument which takes on the value `true` when it can be proven that this is the last time execution will exit the scope, or `false` otherwise.

```
defn dynamic-wind<?T> (in:False|(() -> ?),
  body:() -> ?T,
  out:False|(True|False -> ?)) -> T
```

Using `dynamic-wind`, the above example can be rewritten as follows:

```
val old-X = X
dynamic-wind(
  fn () : X = 42
  f
  fn () : X = old-X)
```

Sequence Library

The core package contains a large number of convenience functions that operate on sequences. The majority of collection datastructures in Stanza's core library are subtypes of `Seqable` and thus can be used with the sequence library.

Operating Functions

Operating functions, such as `do`, `seq`, and `find`, are functions with type signatures compatible with the `for` macro form. The following macro form:

```
for x in xs do :
  body code
```

is equivalent to the following direct call to the `do` operating function:

```
do(fn (x) :
    body code
  xs)
```

Multi-argument operating functions are used with the following macro form.

```
for (x in xs, y in ys, z in zs) do :
  body code
```

is equivalent to the following direct call to the `do` operating function:

```
do(fn (x, y, z) :
    body code
  xs, ys, zs)
```

As a matter of style, if the body consists of more than a single function call then the macro form is used, as in the following example:

```
val x = 671
val prime? = for i in 2 to x none? :
  x % i == 0
```

If the body consists of a single function call then the standard function call form is used instead.

```
val x = 671
defn divides-x? (i:Int) : x % i == 0
val prime? = none?(divides-x?, 2 to x)
```

The above example can also be written using anonymous function notation as follows:

```
val x = 671
val prime? = none?({x % _ == 0}, 2 to x)
```

do

This multi calls the function `f` on each item in the sequence `xs`. A default method defined in terms of the fundamental sequence operations is provided, but users may provide customized methods for subtypes of `Seqable` for efficiency.

```
defmulti do<?T> (f:T -> ?, xs:Seqable<?T>) -> False
```

These multis operate similarly to the single collection version of do but instead calls the function `f` with successive items from multiple sequences. The sequences are stepped through in parallel and iteration stops as soon as one of them is empty.

```
defmulti do<?T,?S> (f:(T,S) -> ?,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> False

defmulti do<?T,?S,?U> (f:(T,S,U) -> ?,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> False
```

find

This function iterates through the sequence `xs` and calls the function `f` on each item, searching for the first item for which `f` returns `true`. If `f` returns `true` on some item, then that item is returned by `find`. If `f` does not return `true` on any item in the sequence, then `find` returns `false`.

```
defn find<?T> (f: T -> True|False, xs:Seqable<?T>) -> T|False
```

This function iterates through the sequence `xs` and `ys` in parallel and calls the function `f` with an item from each sequence. Iteration stops as soon as either sequence is empty. If `f` returns `true` when called by an item, `x`, from `xs`, and an item, `y`, from `ys`, then `x` is returned by `find`.

```
defn find<?T,?S> (f: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> T|False
```

find!

These functions behave identically to `find` except that they assume that there exists an item (or pair of items) for which `f` returns `true`.

```
defn find!<?T> (f: T -> True|False, xs:Seqable<?T>) -> T

defn find!<?T, ?S> (f: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> T
```

first

This function iterates through the sequence xs and calls the function f on each item, searching for the first item for which f returns a One object. If f returns a One object on some item, then the One object is returned. If f returns a None object for all items in the sequence, then a None object is returned by first.

```
defn first<?T,?R> (f: T -> Maybe<?R>, xs:Seqable<?T>) -> Maybe<R>
```

This function iterates through the sequence xs and ys in parallel and calls the function f repeatedly with an item from each sequence. Iteration stops as soon as either sequence is empty. If f returns a One object on some pair of items, then the One object is returned. If f returns a None object for all items in the sequences, then a None object is returned by first.

```
defn first<?T,?S,?R> (f: (T,S) -> Maybe<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Maybe<R>
```

first!

These functions behave identically to first except that they assume that there exists an item (or pair of items) for which f returns a One object. The wrapped value within the One object is returned.

```
defn first!<?T,?R> (f: T -> Maybe<?R>, xs:Seqable<?T>) -> R

defn first!<?T,?S,?R> (f: (T,S) -> Maybe<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> R
```

seq

This function constructs the sequence resulting from calling the function f on each item in the sequence xs.

```
defn seq<?T,?S> (f:T -> ?S, xs:Seqable<?T>) -> Seq<S>
```

These functions behave similarly to the single collection version of seq but instead calls the function f with successive items from multiple sequences. The sequences are stepped through in parallel and iteration stops as soon as one of them is empty.

```
defn seq<?T,?S,?R> (f:(T,S) -> ?R,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Seq<R>

defn seq<?T,?S,?U,?R> (f:(T,S,U) -> ?R,
  xs:Seqable<?T>,
```

```
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> Seq<R>
```

seq?

This function constructs a sequence by calling the function `f` on each item in the sequence `xs`. For each item in `xs`, if `f` returns a `One` object, then the unwrapped value is included in the result sequence. If `f` returns a `None` object then the item is not included in the result sequence.

```
defn seq?<?T,?R> (f: T -> Maybe<?R>, xs:Seqable<?T>) -> Seq<R>
```

These functions construct a sequence by iterating through the given sequences in parallel and calling the function `f` repeatedly with an item from each sequence. For each pair (or triplet) of items, if `f` returns a `One` object, then the unwrapped value is included in the result sequence. If `f` returns a `None` object then the item is not included in the result sequence.

```
defn seq?<?T,?S,?R> (f: (T,S) -> Maybe<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Seq<R>

defn seq?<?T,?S,?U,?R> (f: (T,S,U) -> Maybe<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> Seq<R>
```

filter

This function constructs a sequence by calling the function `f` on each item in the sequence `xs`, and including the item in the result sequence only if the call to `f` returns true.

```
defn filter<?T> (f: T -> True|False, xs:Seqable<?T>) -> Seq<T>
```

This function constructs a sequence by iterating through the `xs` and `ys` sequences in parallel and calling the function `f` repeatedly with an item from each sequence. For each pair of items, if `f` returns `true`, then the item from the `xs` sequence is included in the result sequence.

```
defn filter<?T,?S> (f: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Seq<T>
```

index-when

This function iterates through the items in the `xs` sequence, calling `f` on each one, and returns the first index at which the call to `f` returns `true`. If no call to `f` returns `true`, then `index-when` returns `false`.

```
defn index-when<?T> (f: T -> True|False, xs:Seqable<?T>) -> Int|False
```

This function iterates through the `xs` and `ys` sequences in parallel, calling `f` on each pair of items from `xs` and `ys`, and returns the first index at which the call to `f` returns `true`. If no call to `f` returns `true`, then `index-when` returns `false`.

```
defn index-when<?T,?S> (f: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Int|False
```

index-when!

These functions behave identically to `index-when` except that they assume there exists an item (or pair of items) for which the call to `f` returns `true`.

```
defn index-when!<?T> (f: T -> True|False, xs:Seqable<?T>) -> Int

defn index-when!<?T,?S> (f: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Int
```

split!

This function iterates through the items in the `xs` sequence, separating them into two collections depending on whether calling `f` on the item returns `true` or `false`. The function returns a tuple of two collections, the first of which contains all items in the sequence for which `f` returned `true`, and the second of which contains all items for which `f` returned false.

```
defn split!<?T> (f: T -> True|False, xs:Seqable<?T>) ->
  [Collection<T>&Lengthable, Collection<T>&Lengthable]
```

split

This function iterates through the items in the `xs` sequence, separating them into two sequences depending on whether calling `f` on the item returns `true` or `false`. The function returns a tuple of two sequences, the first of which contains all items in the sequence for which `f` returned `true`, and the second of which contains all items for which `f` returned false. The original argument sequence should not be used after this call.

```
defn split<?T> (f: T -> True|False, xs:Seqable<?T>) -> [Seq<T>, Seq<T>]
```

fork

This function takes an argument sequence and returns two other sequences containing the same items. The argument sequence is only iterated through once. The original argument sequence should not be used after this call.

```
defn fork<?T> (xs:Seqable<?T>) -> [Seq<T>, Seq<T>]
```

If an additional argument, `n`, is given, then the function returns `n` new sequences containing the same items.

```
defn fork<?T> (xs:Seqable<?T>, n:Int) -> [Seq<T>, Seq<T>]
```

take-while

This function constructs a sequence by taking items successively from the `xs` sequence as long as calling `f` upon the item returns `true`. The resulting sequence ends as soon as `xs` is empty or `f` returns `false`. The item for which `f` returns `false` is not included in the resulting sequence.

```
defn take-while<?T> (f: T -> True|False, xs:Seqable<?T>) -> Seq<T>
```

take-until

This function constructs a sequence by taking items successively from the `xs` sequence as long as calling `f` upon the item returns `false`. The resulting sequence ends as soon as `xs` is empty or `f` returns `true`. The item for which `f` returns `true` is included in the resulting sequence.

```
defn take-until<?T> (f: T -> True|False, xs:Seqable<?T>) -> Seq<T>
```

seq-cat

This function constructs a sequence by calling `f` upon each item on the `xs` sequence. `seq-cat` then returns the concatenation of all sequences returned by `f`.

```
defn seq-cat<?T,?R> (f:T -> Seqable<?R>, xs:Seqable<?T>) -> Seq<R>
```

These functions construct a sequence by iterating through the given sequences in parallel and calling `f` upon each pair (or triplet) of items from each sequence. `seq-cat` then returns the concatenation of all sequences returned by `f`.

```
defn seq-cat<?T,?S,?R> (f:(T,S) -> Seqable<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> Seq<R>
```

```
defn seq-cat<?T,?S,?U,?R> (f:(T,S,U) -> Seqable<?R>,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> Seq<R>
```

all?

This function iterates through the items in the sequence `xs` and calls `pred?` on each one. If `pred?` returns `true` for every item in the sequence `all?` returns true. If `pred?` returns `false` for any item, then `all?` immediately returns `false`.

```
defn all?<?T> (pred?: T -> True|False, xs:Seqable<?T>) -> True|False
```

These functions iterate through the given sequences in parallel and calls `pred?` on each pair (or triplet) of items from each sequence. Iteration stops as soon as any sequence is empty. `all?` returns `true` if all calls to `pred?` returns `true`. `all?` returns `false` immediately after a call to `pred?` returns `false`.

```
defn all?<?T,?S> (pred?: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> True|False

defn all?<?T,?S,?U> (pred?: (T,S,U) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> True|False
```

none?

This function iterates through the items in the sequence `xs` and calls `pred?` on each one. If `pred?` returns `true` for no item in the sequence then `none?` returns `true`. If `pred?` returns `true` for any item, then `none?` immediately returns `false`.

```
defn none?<?T> (pred?: T -> True|False, xs:Seqable<?T>) -> True|False
```

These functions iterate through the given sequences in parallel and calls `pred?` on each pair (or triplet) of items from each sequence. Iteration stops as soon as any sequence is empty. `none?` returns `true` if no calls to `pred?` returns `true`. `none?` returns `false` immediately after a call to `pred?` returns `true`.

```
defn none?<?T,?S> (pred?: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> True|False

defn none?<?T,?S,?U> (pred?: (T,S,U) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> True|False
```

any?

This function iterates through the items in the sequence `xs` and calls `pred?` on each one. If `pred?` returns `true` for no item in the sequence then `any?` returns `false`. If `pred?` returns `true` for any item, then `any?` immediately returns `true`.

```
defn any?<?T> (pred?: T -> True|False, xs:Seqable<?T>) -> True|False
```

These functions iterate through the given sequences in parallel and calls `pred?` on each pair (or triplet) of items from each sequence. Iteration stops as soon as any sequence is empty. `any?` returns `false` if no calls to `pred?` returns `true`. `any?` returns `true` immediately after a call to `pred?` returns `true`.

```
defn any?<?T,?S> (pred?: (T,S) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>) -> True|False

defn any?<?T,?S,?U> (pred?: (T,S,U) -> True|False,
  xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> True|False
```

count

This function iterates through the items in the sequence `xs` and calls `pred?` on each one. `count` returns the number of times that `pred?` returned `true`.

```
defn count<?T> (pred?: T -> True|False, xs:Seqable<?T>) -> Int
```

Sequence Constructors

Sequence constructors are functions that take non-Seq objects and construct and return Seq objects from them.

repeat

This function creates an infinite sequence resulting from repeating `x` indefinitely.

```
defn repeat<?T> (x:?T) -> Seq<T>
```

This function creates a sequence resulting from repeating `x` for `n` number of times.

```
defn repeat<?T> (x:?T, n:Int) -> Seq<T> & Lengthable
```

repeatedly

This function creates an infinite sequence from the results of calling `f` repeatedly.

```
defn repeatedly<?T> (f:() -> ?T) -> Seq<T>
```

This function creates a sequence resulting from the results of calling `f` for `n` number of times.

```
defn repeatedly<?T> (f:() -> ?T, n:Int) -> Seq<T> & Lengthable
```

repeat-while

This function creates a sequence resulting from calling `f` repeatedly and including, in the result sequence, the unwrapped values of every `One` object returned by `f`. The sequence ends as soon as `f` returns a `None` object.

```
defn repeat-while<?T> (f: () -> Maybe<?T>) -> Seq<T>
```

Sequence Operators

Sequence operators are functions that take `Seq` objects as arguments and return a new `Seq` object.

filter

This function constructs a sequence by iterating through the `xs` and `sel` sequences in parallel and including items from the `xs` sequence in the result sequence only if the corresponding item from the `sel` sequence is `true`.

```
defn filter<?T> (xs:Seqable<?T>, sel:Seqable<True|False>) -> Seq<T>
```

take-n

This function constructs a sequence by taking the first `n` items from the `xs` sequence. The `xs` sequence must contain at least `n` items.

```
defn take-n<?T> (n:Int, xs:Seqable<?T>) -> Seq<T>
```

take-up-to-n

This function constructs a sequence by taking up to the first `n` items from the `xs` sequence.

```
defn take-up-to-n<?T> (n:Int, xs:Seqable<?T>) -> Seq<T>
```

cat

This function constructs a new sequence by concatenating the `a` sequence with the `b` sequence.

```
defn cat<?T> (a:Seqable<?T>, b:Seqable<?T>) -> Seq<T>
```

cat-all

This function constructs a new sequence by concatenating together all sequences in the `xss` sequence.

```
defn cat-all<?T> (xss: Seqable<Seqable<?T>>) -> Seq<T>
```

join

This function constructs a new sequence by including the item `y` in between each item in `xs`.

```
defn join<?T,?S> (xs:Seqable<?T>, y:?S) -> Seq<T|S>
```

zip

This function constructs a new sequence by iterating through the sequences `xs` and `ys` in parallel and including the 2 element tuples formed from the items of each sequence. Iteration stops as soon as either sequence is empty.

```
defn zip<?T,?S> (xs:Seqable<?T>, ys:Seqable<?S>) -> Seq<[T,S]>
```

This function constructs a new sequence by iterating through the sequences `xs`, `ys`, and `zs`, in parallel and including the 3 element tuples formed from the items of each sequence. Iteration stops as soon as any sequence is empty.

```
defn zip<?T,?S,?U> (xs:Seqable<?T>,
  ys:Seqable<?S>,
  zs:Seqable<?U>) -> Seq<[T,S,U]>
```

zip-all

This function constructs a new sequence by iterating through all sequences in xss in parallel, and including the tuples formed from the items of each sequence. Iteration stops as soon as any sequence is empty.

```
defn zip-all<?T> (xss:Seqable<Seqable<?T>>) -> Seq<Tuple<T>>
```

Sequence Reducers

Sequence reducers are functions that take Seq objects as arguments and compute and return a non-Seq object.

contains?

This function returns true if the xs sequence contains the item y. Otherwise it returns false.

```
defn contains? (xs:Seqable<Equalable>, y:Equalable) -> True|False
```

index-of

This function returns the first index at which the item in the xs sequence is equal to the item y. If no item in xs is equal to y then index-of returns false.

```
defn index-of (xs:Seqable<Equalable>, y:Equalable) -> Int|False
```

index-of!

This function behaves identically to index-of except that it assumes there exists an item in xs that is equal to y.

```
defn index-of! (xs:Seqable<Equalable>, y:Equalable) -> Int
```

split

This function iterates through the items in the xs and ss sequence in parallel, and separates the items from xs into two collections depending on whether the corresponding item from ss is true or false. The function returns a tuple of two collections, the first of which contains all items in xs for which the corresponding item in ss is true, and the second of which contains the other items in xs.

```
defn split<?T> (xs:Seqable<?T>, ss:Seqable<True|False>) ->
  [Collection<T>&Lengthable, Collection<T>&Lengthable]
```

count

This function returns the number of items in the given sequence. If the sequence is a subtype of `Lengthable`, then the length is returned directly. Otherwise, the length is calculated by iterating through the entire sequence.

```
defn count (xs:Seqable) -> Int
```

reduce

This function computes the left oriented reduction using function `f` on the sequence `xs` starting with initial value `x0`. If `xs` is empty then `x0` is returned. Otherwise, the function `f` is applied on `x0` and the first element in `xs`, then `f` is applied again on that result and the second element in `xs`, then `f` is applied on that result and the third element in `xs`, and so forth until `xs` is empty. `reduce` returns the final result returned by `f`.

```
defn reduce<?T,?S> (f: (T, S) -> ?T, x0: ?T, xs:Seqable<?S>) -> T
```

If no initial value is provided, then the first element in `xs` is used as the initial value of the reduction. `xs` must not be empty.

```
defn reduce<?T,?S> (f: (T|S, T) -> ?S, xs:Seqable<?T>) -> T|S
```

reduce-right

This function computes the right oriented reduction using function `f` on the sequence `xs` with final value `xn`. If `xs` is empty then `xn` is returned. If `xs` has a single element, then `f` is applied on that element and `xn`. If `xs` has two elements, then f is applied on the first element and the result of applying `f` on the second element and `xn`. If `xs` has three elements, then `f` is applied on the first element and the result of applying `f` on the second element and the result of applying `f` on the third element and `xn`. Et cetera.

```
defn reduce-right<?T,?S> (f: (S, T) -> ?T, xs:Seqable<?S>, xn:?T) -> T
```

If no final value is provided, then the last element in `xs` is used as the final value of the reduction. `xs` must not be empty.

```
defn reduce-right<?T,?S> (f: (T, T|S) -> ?S, xs:Seqable<?T>) -> T|S
```

unique

This function returns a list containing all unique items in the sequence xs.

```
defn unique<?T> (xs:Seqable<?T&Equalable>) -> List<T>
```

lookup?

This function iterates through the xs sequence, and looks for the first KeyValue pair whose key is equal to k. If such a pair is found, then its value is returned. If not then default is returned.

```
defn lookup?<?K,?V,?D> (xs:Seqable<KeyValue<?K&Equalable,?V>>,
  k:K&Equalable,
  default:?D) -> D|V
```

If no default value is provided, then false is returned if no appropriate KeyValue pair is found.

```
defn lookup?<?K,?V> (xs:Seqable<KeyValue<?K&Equalable,?V>>,
  k:K&Equalable) -> False|V
```

lookup

This function behaves identically to lookup? except that it assumes that an appropriate KeyValue pair exists and returns its value.

```
defn lookup<?K,?V> (xs: Seqable<KeyValue<?K&Equalable,?V>>, k:K&
   Equalable) -> V
```

fork-on-seq

This function takes a sequence xs, and returns a tuple containing the results of calling the functions f and g with xs. The functions f and g are computed concurrently such that xs is iterated through only once. This function allows multiple values to be computed from a sequence that can only be iterated through once.

```
defn fork-on-seq<?T,?X,?Y> (xs:Seqable<?T>,
  f:Seq<T> -> ?X,
  g:Seq<T> -> ?Y) -> [X,Y]
```

This function takes a sequence xs, and returns a tuple containing the results of calling the functions f, g, and h with xs. The functions are computed concurrently such that xs is iterated through only once.

```
defn fork-on-seq<?T,?X,?Y,?Z> (xs:Seqable<?T>,
  f:Seq<T> -> ?X,
```

```
  g:Seq<T> -> ?Y,
  h:Seq<T> -> ?Z) -> [X,Y,Z]
```

This function takes a sequence `xs`, and returns a tuple containing the results of calling every function in `fs` with `xs`. The functions are computed concurrently such that `xs` is iterated through only once.

```
defn fork-on-seq<?T,?S> (xs:Seqable<?T>,
  fs:Seqable<(Seq<T> -> ?S)>) -> Tuple<S>
```

Sorting

Stanza's core library provides convenience functions for sorting collections.

qsort!

This function sorts the collection using the given comparison function and the quick sort algorithm.

```
defn qsort!<?T> (xs:IndexedCollection<?T>, is-less?:(T,T) -> True|False
   ) -> False
```

This function sorts the collection using the `less?` multi and using the quick sort algorithm.

```
defn qsort!<?T> (xs:IndexedCollection<?T&Comparable>) -> False
```

This function sorts the collection by comparing the keys extracted using the `key` function and using the quick sort algorithm.

```
defn qsort!<?T> (key:T -> Comparable, xs:IndexedCollection<?T>) ->
   False
```

lazy-qsort

This function returns a lazily sorted sequence of the given sequence `xs` using the given comparison function and the quick sort algorithm.

```
defn lazy-qsort<?T> (xs:Seqable<?T>,
  is-less?:(T,T) -> True|False) -> Collection<T> & Lengthable
```

This function returns a lazily sorted collection of the given sequence `xs` using the `less?` multi and the quick sort algorithm.

```
defn lazy-qsort<?T> (xs:Seqable<?T&Comparable>) -> Collection<T> &
   Lengthable
```

This function returns a lazily sorted collection of the given sequence `xs` by comparing keys extracted using the `key` function and using the quick sort algorithm.

```
defn lazy-qsort<?T> (key:T -> Comparable,
  xs:Seqable<?T>) -> Collection<T> & Lengthable
```

LivenessTracker

Stanza's support for automatic garbage collection means that users do not need to manually delete objects after their use. However, it is often useful to know when an object has been reclaimed to perform additional cleanup operations. `LivenessTracker` and `LivenessMarker` objects are used to track whether an object is still live.

```
deftype LivenessTracker<T>
deftype LivenessMarker
```

LivenessTracker

This function creates a `LivenessTracker` with an associated value to indicate the identity of the tracker.

```
defn LivenessTracker<?T> (value:?T) -> LivenessTracker<T>
```

value

This function can be used to retrieve the value associated with a tracker.

```
defn value<?T> (tracker:LivenessTracker<?T>) -> T
```

marker

This function is used to create or check upon a `LivenessMarker`.

```
defn marker (tracker:LivenessTracker) -> False|LivenessMarker
```

The first time this function is called on a `LivenessTracker` it is guaranteed to return its associated `LivenessMarker`. This marker may then be stored with an object whose liveness the user wishes to track. On subsequent calls to `marker`, the function will either return the created marker, which indicates that the marker is still live. Or it will return `false`, which indicates that the marker is now longer live, and hence neither is the object in which it is stored.

For example, consider tracking the liveness of values of type `Car`. Intuitively, it is helpful to imagine a `LivenessTracker` as a futuristic pistol capable of shooting a tracking beacon (its `LivenessMarker`) that sticks to the `Car`. Periodically, you may ask the tracker whether it is still receiving signal from the beacon, in which case the marker and hence the `Car` is still live. If the beacon is no longer sending a signal (`marker` returns `false`), then the marker and hence the `Car` is no longer live.

marker!

This function assumes that `marker` will return a `LivenessMarker` and returns it.

```
defn marker! (tracker:LivenessTracker) -> LivenessMarker
```

add-gc-notifier

This function registers a new garbage collection notifier with Stanza. Garbage collection notifiers are run after every garbage collection phase.

```
defn add-gc-notifier (f: () -> ?) -> False
```

System Utilities

Stanza's core library provides convenience functions for manipulating the system environment.

command-line-arguments

The command line arguments used to invoke the program can be retrieved as an array of strings using the following function.

```
defn command-line-arguments () -> Array<String>
```

file-exists?

This function checks whether the given file exists given its name.

```
defn file-exists? (filename:String) -> True|False
```

delete-file

This function deletes the given file.

```
defn delete-file (path:String) -> False
```

resolve-path

This function expands all symbolic links in the given path and returns an absolute path that points to the same file. If such a file does not exist then `false` is returned.

```
defn resolve-path (path:String) -> String|False
```

current-time-ms

This function returns the number of milliseconds elapsed since January 1, 1970.

```
defn current-time-ms () -> Long
```

current-time-us

This function returns the number of microseconds elapsed since January 1, 1970.

```
defn current-time-us () -> Long
```

get-env

This function returns the value associated with the environment variable with the given name. `false` is returned if there is no such variable.

```
defn get-env (name:String) -> String|False
```

set-env

These functions associate the environment variable of the given name with the given value. If the environment variable already exists then its value is overwritten if the `overwrite` flag is `true`, otherwise its old value is kept. If no `overwrite` flag is given then its default value is `true`.

```
defn set-env (name:String, value:String, overwrite:True|False) -> False
defn set-env (name:String, value:String) -> False
```

call-system

This function calls the system shell with the given command. If the call fails with an error then the function throws a `SystemCallException`, otherwise it returns the integer status code of the call.

```
defn call-system (cmd:String) -> Int
```

Timer

`Timers` are used for measuring elapsed time at various granularities.

```
deftype Timer
```

MillisecondTimer

This function creates a timer with millisecond granularity and an associated name.

```
defn MillisecondTimer (name:String) -> Timer
```

MicrosecondTimer

This function creates a timer with microsecond granularity and an associated name.

```
defn MicrosecondTimer (name:String) -> Timer
```

PiggybackTimer

This function creates a timer that piggybacks off of the time measured by another timer. Time only elapses for a piggyback timer if the timer it is piggybacking is running. A `PiggybackTimer` returns time in the same granularity as the timer it is piggybacking.

```
defn PiggybackTimer (name:String, t:Timer) -> Timer
```

start

This function starts the timer. It is an error to start a `Timer` that is already running.

```
defn start (t:Timer) -> False
```

stop

This function stops the timer. It is an error to stop a `Timer` that is not running.

```
defn stop (t:Timer) -> False
```

time

This function retrieves the current time elapsed by the timer. The unit of the number returned is dependent upon how the timer is created.

```
defn time (t:Timer) -> Long
```

Chapter 3

Math Package

The `math` package contains basic mathematical quantities and operations.

Quantities

PI

This value contains a double-precision definition of the mathematical quantity pi.

```
val PI : Double
```

PI-F

This value contains a single-precision definition of the mathematical quantity pi.

```
val PI-F : Float
```

Basic Operations

exp

This function computes e to the power of `x`.

```
defn exp (x:Double) -> Double
defn exp (x:Float) -> Float
```

log

This function computes the natural logarithm of x.

```
defn log (x:Double) -> Double
defn log (x:Float) -> Float
```

log10

This function computes the base 10 logarithm of x.

```
defn log10 (x:Double) -> Double
defn log10 (x:Float) -> Float
```

pow

This function computes x to the power of y.

```
defn pow (x:Double, y:Double) -> Double
defn pow (x:Float, y:Float) -> Float
```

sin

This function computes the sine of x.

```
defn sin (x:Double) -> Double
defn sin (x:Float) -> Float
```

cos

This function computes the cosine of x.

```
defn cos (x:Double) -> Double
defn cos (x:Float) -> Float
```

tan

This function computes the tangent of x.

```
defn tan (x:Double) -> Double
defn tan (x:Float) -> Float
```

asin

This function computes the inverse sine of x.

```
defn asin (x:Double) -> Double
defn asin (x:Float) -> Float
```

acos

This function computes the inverse cosine of x.

```
defn acos (x:Double) -> Double
defn acos (x:Float) -> Float
```

atan

This function computes the inverse tangent of x.

```
defn atan (x:Double) -> Double
defn atan (x:Float) -> Float
```

atan2

This function computes the inverse tangent of y divided by x. The signs of both arguments are used to determine the quadrant of the result.

```
defn atan2 (y:Double, x:Double) -> Double
defn atan2 (y:Float, x:Float) -> Float
```

sinh

This function computes the hyperbolic sine of x.

```
defn sinh (x:Double) -> Double
defn sinh (x:Float) -> Float
```

cosh

This function computes the hyperbolic cosine of x.

```
defn cosh (x:Double) -> Double
defn cosh (x:Float) -> Float
```

tanh

This function computes the hyperbolic tangent of x.

```
defn tanh (x:Double) -> Double
defn tanh (x:Float) -> Float
```

ceil

This function computes the ceiling of x.

```
defn ceil (x:Double) -> Double
defn ceil (x:Float) -> Float
```

floor

This function computes the floor of x.

```
defn floor (x:Double) -> Double
defn floor (x:Float) -> Float
```

round

This function rounds x to the nearest integer.

```
defn round (x:Double) -> Double
defn round (x:Float) -> Float
```

to-radians

This function converts an angle expressed in degrees to radians.

```
defn to-radians (x:Double) -> Double
defn to-radians (x:Float) -> Float
```

to-degrees

This function converts an angle expressed in radians to degrees.

```
defn to-degrees (x:Double) -> Double
defn to-degrees (x:Float) -> Float
```

Chapter 4

Collections Package

The `collections` package consists of functions and types useful for managing collection datastructures.

Vector

A `Vector` object represents a dynamically growing array of items. Like arrays, each item can be accessed using an integer index. However, a `Vector` object can be resized after creation.

```
deftype Vector<T> <: IndexedCollection<T>
```

`Vector` is a subtype of `IndexedCollection` and thus implements the appropriate methods for retrieving elements, assigning elements, and retrieving its length. When assigning an item, `v`, to an index `i`, if `i` is less than the length of the vector then the item at that location is overwritten by `v`. If `i` is equal to the length of the vector, then `v` is added to the end of the vector.

Vector

This function creates a `Vector` for holding objects of type `T` initialized with a capacity of `cap`.

```
defn Vector<T> (cap:Int) -> Vector<T>
```

If no capacity is given, then the default capacity is `8`.

```
defn Vector<T> () -> Vector<T>
```

to-vector

This function creates a new `Vector` from the elements in the sequence `xs`.

```
defn to-vector<T> (xs:Seqable<T>) -> Vector<T>
```

add

This function adds the given value to the end of the vector.

```
defn add<?T> (v:Vector<?T>, value:T) -> False
```

add-all

This function adds all of the values in the sequence `vs` to the end of the vector.

```
defn add-all<?T> (v:Vector<?T>, vs:Seqable<T>) -> False
```

clear

This function removes every item in the vector.

```
defn clear (v:Vector) -> False
```

pop

This function removes the last item in the vector and returns it. The vector must not be empty.

```
defn pop<?T> (v:Vector<?T>) -> T
```

peek

This function returns the last item in the vector. The vector must not be empty.

```
defn peek<?T> (v:Vector<?T>) -> T
```

remove

This function removes the item at index `i` in the vector and returns the removed item. `i` must be less than the length of the vector. The operation takes time proportional to the difference between the length of the vector and `i`.

```
defn remove<?T> (v:Vector<?T>, i:Int) -> T
```

remove

This function removes the items within the range `r` from the vector. The range must be a *dense index range* with respect to the vector. The operation takes time proportional to the difference between the length of the vector and the end of the range.

```
defn remove (v:Vector, r:Range) -> False
```

update

This function either overwrites or removes items from the vector by calling the function `f` on each item in the vector. For each item in the vector, if `f` returns a `One` object, then the item is overwritten with the wrapped value in the `One` object. If `f` returns a `None` object, then the item is removed from the vector. `update` is compatible with the `for` macro form.

```
defn update<?T> (f: T -> Maybe<T>, v:Vector<?T>) -> False
```

remove-item

This function removes the first occurrence of the item `x` in the vector `v` if it exists. The function returns `true` if an item was removed, or `false` otherwise.

```
defn remove-item<?T> (v:Vector<?T&Equalable>, x:T&Equalable) -> True|
   False
```

remove-when

This function calls the function `f` on every item in the vector `v`. Items for which `f` returned `true` are removed.

```
defn remove-when<?T> (f: T -> True|False, v:Vector<?T>) -> False
```

trim

This function sets the capacity of the vector to be equal to the size of the vector, thus removing storage for additional elements.

```
defn trim (v:Vector) -> False
```

shorten

This function truncates the given vector down to the specified size. The given size must be less than or equal to the length of the vector.

```
defn shorten (v:Vector, size:Int) -> False
```

lengthen

This function pads the given vector up to the specified size by adding `x` repeatedly to the end of the vector. The given size must be greater than or equal to the length of the vector.

```
defn lengthen<?T> (v:Vector<?T>, size:Int, x:T) -> False
```

set-length

This function sets the length of the vector to the new specified length. If the given length is shorter than the current length then the vector is truncated. Otherwise the vector is padded using the item `x`.

```
defn set-length<?T> (v:Vector<?T>, length:Int, x:T) -> False
```

map

This function creates a new `Vector` from the results of applying the function `f` repeatedly to each item in the given vector `v`. Note that the element type, `R`, of the result vector must be given explicitly.

```
defn map<R,?T> (f: T -> R, v:Vector<?T>) -> Vector<R>
```

Queue

A `Queue` represents an indexed collection with support for acting as a first in last out (FIFO) datastructure.

```
deftype Queue<T> <: IndexedCollection<T>
```

`Queue` is a subtype of `IndexedCollection` and thus implements the appropriate methods for retrieving elements, assigning elements, and retrieving its length. When assigning an item, `v`, to an index `i`, if `i` is not negative and less than the length of the queue then the item at that location is overwritten by `v`. If i is equal to `-1`, then `v` is added to the front of the queue.

Queue

This function creates a `Queue` for holding objects of type `T` initialized with a capacity of `cap`.

```
defn Queue<T> (cap:Int) -> Queue<T>
```

If no capacity is given, then the default capacity is `8`.

```
defn Queue<T> () -> Queue<T>
```

add

This function adds the given item to the front of the queue. The corresponding indices of all existing entries in the queue increase by `1`.

```
defn add<?T> (q:Queue<?T>, x:T) -> False
```

pop

This function removes and returns the item at the back of the queue. The queue must not be empty.

```
defn pop<?T> (q:Queue<?T>) -> T
```

peek

This function returns the item at the back of the queue. The queue must not be empty.

```
defn peek<?T> (q:Queue<?T>) -> T
```

clear

This function removes all items in the queue.

```
defn clear (q:Queue) -> False
```

Table

A `Table` represents a mapping between key objects of type `K` and value objects of type `V`.

```
deftype Table<K,V> <: Collection<KeyValue<K,V>> & Lengthable
```

`Table` is a subtype of `Collection` and must implement an appropriate method for the `to-seq` multi that allows for the table to be viewed as a sequence of `KeyValue` pairs.

`Table` is a subtype of `Lengthable` and must implement an appropriate method for the `length` multi that returns the number of `KeyValue` pairs currently in the table.

Mandatory minimal implementation: `to-seq`, `length`, `set`, `get?`, `default?`, `remove`, `clear`.

set

This multi associates the key object `k` with the value object `v` in the table.

```
defmulti set<?K,?V> (t:Table<?K,?V>, k:K, v:V) -> False
```

get?

This multi retrieves the value object associated with the key object `k` in the table. If there is no such key in the table then the default value `d` is returned.

```
defmulti get?<?K,?V,?D> (t:Table<?K,?V> k:K, d:?D) -> V|D
```

default?

This multi returns the optional default value associated with the table. If a default value is provided then it is returned when accessing keys that do not exist in the table.

```
defmulti default?<?V> (t:Table<?,?V>) -> Maybe<V>
```

remove

This multi removes the given key and its associated value from the table. If there is a key then the function returns `true`. Otherwise the function returns `false`.

```
defmulti remove<?K> (t:Table<?K,?>, k:K) -> True|False
```

clear

This multi removes all keys and all values in the table.

```
defmulti clear (t:Table) -> False
```

get?

This function returns the value associated with the given key if it exists, otherwise it returns `false`.

```
defn get?<?K,?V> (t:Table<?K,?V>, k:K) -> V|False
```

get

This multi assumes that the given `k` exists in the table and returns its associated value, or that a default value is provided. A default method, defined in terms of `get?`, is provided but users may provide customized methods for subtypes of `Table` for efficiency purposes.

```
defmulti get<?K,?V> (t:Table<?K,?V>, k:K) -> V
```

key?

This multi returns `true` if the given key `k` exists in the table. A default method, defined in terms of `get?`, is provided but users may provide customized methods for subtypes of `Table` for efficiency purposes.

```
defmulti key?<?K> (t:Table<?K,?>, k:K) -> True|False
```

keys

This multi returns a sequence containing all keys in the table. A default method, defined in terms of `to-seq`, is provided but users may provide customized methods for subtypes of `Table` for efficiency purposes.

```
defmulti keys<?K> (t:Table<?K,?>) -> Seqable<K>
```

values

This multi returns a sequence containing all values in the table. A default method, defined in terms of `to-seq`, is provided but users may provide customized methods for subtypes of `Table` for efficiency purposes.

```
defmulti values<?V> (t:Table<?,?V>) -> Seqable<V>
```

empty?

This function returns `true` if the table contains no keys or values.

```
defn empty? (t:Table) -> True|False
```

HashTable

A `HashTable` provides an efficient implementation of tables for keys of type `Equalable`. Key type `K` must be a subtype of `Equalable`.

```
deftype HashTable<K,V> <: Table<K,V>
```

`HashTable` is a subtype of `Table` and support all necessary methods for correct table operations.

HashTable

This function creates a `HashTable` with key objects of type `K` and value objects of type `V`. A `hash` function that maps key objects to integers, an optional default value, and an initial capacity must be provided. If provided, the default value is returned for retrieval operations when the key does not exist.

```
defn HashTable<K,V> (hash: K -> Int,
  default:Maybe<V>,
  initial-capacity:Int) -> HashTable<K,V>
```

If unspecified, the default capacity is 32.

```
defn HashTable<K,V> (hash: K -> Int, default:V) -> HashTable<K,V>
```

If unspecified, then there is no default value.

```
defn HashTable<K,V> (hash: K -> Int) -> HashTable<K,V>
```

This function creates a `HashTable` that uses the `hash` multi to compute hashes for its keys, and a default value of `V`. `K` must be a subtype of both `Hashable` and `Equalable`.

```
defn HashTable<K,V> (default:V) -> HashTable<K,V>
```

If unspecified, then there is no default value.

```
defn HashTable<K,V> () -> HashTable<K,V>
```

Chapter 5

Reader Package

reader

The `reader` package contains the implementation of Stanza's s-expression reader.

read-file

This function reads in the given file and returns its contents as a list of s-expressions.

```
defn read-file (filename:String) -> List
```

read-all

This function converts the given string into a list of s-expressions.

```
defn read-all (text:String) -> List
```

read-all

This function reads in all characters from the given string input stream and returns a list of s-expressions.

```
defn read-all (stream:StringInputStream) -> List
```

read

This function reads a single s-expression from the string input stream.

```
defn read (stream:StringInputStream) -> ?
```

Chapter 6

Macro Utilities Package

The `macro-utils` package contains functions for manipulating s-expressions useful for macro writers.

Utility Functions

tagged-list?

This function returns `true` if the given form is a list (or list wrapped in a `Token`) whose first element is equal to the given symbol, or is a token whose wrapped value is equal to the given symbol.

```
defn tagged-list? (form, tag:Symbol) -> True|False
```

S-Expression Template Engine

The template engine can be thought of as an advanced search and replace utility that operates on s-expressions instead of on strings.

fill-template

This function returns the result of performing search and replace operations on a template form given a collection of desired replacements.

```
defn fill-template (template, replacements:Collection<KeyValue<Symbol
   ,?>>) -> ?
```

The replacements are expressed as `KeyValue` pairs where the key holds the symbol that should be replaced, and the value holds what the key should be replaced by. Values can be objects of any type, but there are a set of distinguished template forms with special replacement rules.

Simple Replacements

Consider the following template :

```
val form = `(
  defn myfunction (arg : argtype) :
    body)
```

and the desired replacements :

```
val replacements = [
  `myfunction => `print-int
  `arg => `x
  `argtype => `Int
  `body => `(println("Inside print-int"),
  println(x))]
```

Then the following call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x : Int) :
    (println("Inside print-int"),
    println(x)))
```

Observe that occurrences of the keys in `replacements` in the template form are systematically replaced with their associated values.

Splice Template

The splice template allows for a symbol to be replaced by multiple values. The following function creates a splice template. Note that the given item must be either a `Collection` or a `Token` wrapping over a `Collection`.

```
defn splice (item:Collection|Token) -> SpliceTemplate
```

Consider again the previous example. Notice that, the key `` `body `` was associated with a 2 element list. And in the resulting form, the occurrence of `body` was replaced with the 2 element list.

By using the splice template we can replace occurrences of `` `body `` directly with the elements inside the list and avoid an extra pair of parenthesis.

Consider the following template :

```
val form = `(
  defn myfunction (arg : argtype) :
    body)
```

and the desired replacements :

```
val replacements = [
  `myfunction => `print-int
  `arg => `x
  `argtype => `Int
  `body => splice(`(println("Inside print-int"),
  println(x)))]
```

Then the following call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x : Int) :
    println("Inside print-int")
    println(x))
```

Nested Template

The nested template allows for a pattern to be repeatedly replaced using a different set of replacements each time. The following function creates a nested template.

```
defn nested (items:Collection<Collection<KeyValue<Symbol,?>>>) ->
   NestedTemplate
```

Consider the following template :

```
val form = `(
  defn myfunction (args{arg : argtype}) :
    body)
```

and the desired replacements :

```
val replacements = [
  `myfunction => `print-int
  `args => nested $ [
    [`arg => `x, `argtype => `Int]
    [`arg => `y, `argtype => `String]
    [`arg => `z, `argtype => `Char]]
  `body => `(println(x))]
```

Then the following call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x:Int, y:String, z:Char) :
    (println(x)))
```

Note, specially, the syntax in the template when using a nested template replacement. If the replacement is a nested template, then the template form key *must* have form :

```
key{pattern ...}
```

Note that the set of replacements used during each replacement of a nested template also includes all the entries from the parent replacements.

For example, if we use the following replacements on the same template form :

```
val replacements = [
  `myfunction => `print-int
  `argtype => `Int
  `args => nested $ [
    [`arg => `x]
    [`arg => `y]
    [`arg => `z, `argtype => `Char]]
  `body => `(println(x))]
```

Then the call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x:Int, y:Int, z:Char) :
    (println(x)))
```

Notice that the `` `argtype `` replacement value is inherited from the parent bindings.

Plural Template

The plural template is similar to the nested template and allows for a pattern to be repeatedly replaced using a different set of replacements each time. The difference is that the plural template allows the user to specify *for each symbol* what it should be replaced with each time, whereas the nested template requires the user to specify *for each replacement* what the set of replacements should be.

The following function creates a plural template given a sequence of replacements, each replacement being a sequence itself. Note that the length of each replacement value must be equal.

```
defn plural (entries: Seqable<KeyValue<Symbol, Seqable>>) ->
    PluralTemplate
```

Consider the following template :

```
val form = `(
  defn myfunction (args{arg : argtype}) :
    body)
```

and the desired replacements :

```
val replacements = [
  `myfunction => `print-int
  `args => plural $ [
    `arg => `(x y z)
    `argtype => `(Int String Char)]
  `body => `(println(x))]
```

Then the following call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x:Int, y:String, z:Char) :
    (println(x)))
```

Choice Template

The choice template allows for a symbol to be replaced with a pattern selected from amongst a list of patterns depending on the replacement value.

The following function creates a choice template given the choice `n`.

```
defn choice (n:Int) -> ChoiceTemplate
```

The following convenience function also allows for users to create a choice template given a boolean value, where `false` is equivalent to choice `0` and `true` is equivalent to choice `1`.

```
defn choice (b:True|False) -> ChoiceTemplate
```

Consider the following template :

```
val form = `(
  defn myfunction (x:Int) :
    body{
      println(x)
    }{
```

```
      x + 1
    }{
      fatal(x)
    })
```

and the desired replacements :

```
val replacements = [
  `myfunction => `print-int
  `body => choice(1)]
```

Then the following call to `fill-template` :

```
fill-template(form, replacements)
```

will return the following replaced form :

```
`(defn print-int (x:Int) :
    x + 1)
```

Note, specially, the syntax in the template when using a choice template replacement. If the replacement is a choice template, then the template form key *must* have form :

```
key{pattern0 ...}{pattern1 ...}{pattern2 ...}
```

We can select which pattern is produced by changing the index we use to create the choice template. The following replacements :

```
val replacements = [
  `myfunction => `print-int
  `body => choice(2)]
```

will result in the following replaced form :

```
`(defn print-int (x:Int) :
    fatal(x))
```