Introduction to Stanza

Patrick S. Li

L.B. Stanza Language

- www.lbstanza.org
- Productivity Language for Turning Prototypes into Products
- Optionally-Typed
- Functional
- Easy to Learn
- Looks Python-ish

Installing Stanza

- www.lbstanza.org
- Documentation > StanzaByExample > Getting Started
- On Mac / Linux:

```
./stanza install -platform os-x
./stanza install -platform linux
./stanza install -platform windows -path .
```

- On Windows:

```
./stanza install -platform windows -path .
```

Absolute Basics

Hello World

- helloworld.stanza:

```
println(“Hello World”)
```

- Terminal:

```
stanza helloworld.stanza -o helloworld
./helloworld
```

Basics

- Basic Harness:

```
defpackage mypackage :
  import core

defn mymain () :
  println("Hello World")

mymain()
```

Strings

```
defpackage mypackage :
  import core

defn mymain () :
  println(“Hello World”)
  println-all([1 “ and “ 2 “ and “ “Timon”])
  println(“%_ and %_ and %_” % [1, 2, “Timon”])

mymain()
```

Strings are surrounded with “s

Operators

```
defpackage mypackage :
  import core

defn mymain () :
  val x = 1
  val y = 2
  println(x + y)
  println(x - y)
  println(x * y)
  println(x / y)
  println((- x))

mymain()
```

Don’t forget these parentheses!

Comments

```
defpackage mypackage :
  import core

defn mymain () :
  val x = 1
  val y = 2
  println(x + y)   ;Add two numbers
  println(x - y)   ;Subtract two numbers
  println(x * y)   ;Multiply two numbers
  println(x / y)   ;Divide two numbers
  println((- x))   ;Negate a number

mymain()
```

Comments begin with ;

Values

```
defpackage mypackage :
  import core

defn mymain () :
  val a = 2
  val b = 2
  val c = 3
  val n = b * b - 4 * a * c
  println(n)

mymain()
```

Values are declared using “val”

Variables

```
defpackage mypackage :
  import core

defn mymain () :
  var x = 2
  x = x + 10
  x = x + 30
  println(x)

mymain()
```

Variables are declared using “var”

Variables and Explicit Types

```
defpackage mypackage :
  import core

defn mymain () :
  var x:Int = 2
  x = x + 10
  x = x + 30
  println(x)

mymain()
```

Now x can only hold integers

Functions

Space!

```
defn myfunction (a, b) :
  val c = a - b
  println("%_ - %_ = %_" % [a, b, c])
  c

myfunction(2, 3)
myfunction(20, 42)
myfunction("hello", 42)
```

The last expression is returned.

Runtime Error!

Functions with Types

Argument Types

```
defn myfunction (a:Int, b:Int) -> Int :
  val c = a - b
  println("%_ - %_ = %_" % [a, b, c])
  c

myfunction(2, 3)
myfunction(20, 42)
myfunction("hello", 42)
```

Return Type

Compile-time Error

The ? Type

myfunction now accepts anything

```
defn myfunction (a:?, b:?) :
  val c = a - b
  println("%_ - %_ = %_" % [a, b, c])
  c

myfunction(2, 3)
myfunction(20, 42)
myfunction("hello", 42)
```

No longer caught

Examples of Types

```
Int: 42, 33, 10
Float: 10.0f, 11.0f, -11.42f
Char: ‘a’, ‘c’, ‘e’
String: “Timon”, “Pumbaa”
True: true
False: false
```

If Expressions

Predicate

```
defn test (x) :
  if x < 3 :
    println("x is less than three")
  else :
    println("x is not less than three")
```

Consequent

Alternate

If Expressions

```
defn test (x) :
  if x < 3 :
    println("x is less than three")
  else if x < 5 :
    println("x is less than five")
  else :
    println("x is not less than five")
```

Chained If

If Expressions

```
defn test (x) :
  if x > 3 and x < 5 :
    println(“x is more than three and less than five”)
  else :
    println(“x is not”)
```

Indentation Structure

```
defn test (x) :
  if x > 3 and x < 5 :
    println("x is more than three and less than five")
  else :
    println("x is not")
```

equivalent

```
defn test (x) : (
  if x > 3 and x < 5 : (
    println("x is more than three and less than five")
  )
  else : (
    println("x is not")
  )
)
```

If Expressions. Not If Statements.

```
defn sign (x) :
  val result =
    if x < 0 : -1
    else if x == 0 : 0
    else : 1
  result

println(sign(-42))
println(sign(101))
println(sign(0))
```

While Loops

```
var i = 0
while i < 10 :
  println(“i = %_” % [i])
  i = i + 1
```

For Loops

Don't forget "do"

```
for i in 0 to 10 do :
  println("i = %_" % [i])
```

For Loops

Ranges

```
for i in 0 to 10 do :
  println(“i = %_” % [i])

0 to 10 : Zero to ten (exclusive)
0 through 10 : Zero to ten (inclusive)
0 to false : Zero to infinity
0 to 10 by 2 : Zero to ten (exclusive) counting in two’s
0 through 10 by 2 : Zero to ten (inclusive) counting in two’s
10 through 0 by -1 : Ten to zero (inclusive) counting down
```

Arrays

```
val xs = Array<Int>(10, 42)
```

10-slot integer array initialized to 42

```
val xs = Array<Int>(10)
```

10-slot uninitialized integer array

```
val xs = Array<Int|String>(10)
```

10-slot array for holding integers or strings

```
val xs = Array<?>(10)
```

10-slot array for holding anything

Arrays

```
val xs = Array<Int>(10)

xs[5] = 42

val y = xs[5]

length(xs)
```

Storing into an array

Loading from an array

Length of an array

Arrays

```
defn print-array (xs:Array<Int>) :
  for i in 0 to length(xs) do :
    println(xs[i])
```

Arrays

```
defn print-array (xs:Array<Int>) :
  for x in xs do :
    println(x)
```

Automatic iteration through array

Labeled Scopes

```
defn first-negative (xs:Array<Int>) :
  val result =
    label<Int|False> myexit :
      for x in xs do :
        if x < 0 :
          myexit(x)
      false
  result
```

Exit block early with x

Labeled Scopes

```
defn first-negative (xs:Array<Int>) :
  label<Int|False> myexit :
    for x in xs do :
      if x < 0 :
        myexit(x)
    false
```

Exit block early with x

Structs

```
defstruct Dog :
  name: String
  breed: String

val d1 = Dog(“Chance”, “Pitbull”)
val d2 = Dog(“Shadow”, “Golden Retriever”)

println(name(d1))
println(breed(d1))
```

Fields

Getter Functions

Structs

```
defstruct Dog :
  name: String
  breed: String
  mood: String with: (setter => set-mood)

val d1 = Dog("Chance", "Pitbull", "Desperate")
val d2 = Dog("Shadow", "Golden Retriever", "Calm")

println(mood(d2))
set-mood(d2, "Exasperated")
println(mood(d2))
```

Mutable Field

Setter Function

Structs

```
defstruct Dog :
  name: String
  breed: String
  mood: String with: (setter => set-mood)

defn retriever? (d:Dog) :
  breed(d) == “Golden Retriever”

val d1 = Dog(“Chance”, “Pitbull”, “Desperate”)
val d2 = Dog(“Shadow”, “Golden Retriever”, “Calm”)

println(retriever?(d1))
println(retriever?(d2))
```

User Function

Structs with Custom Printing Behaviour

Don't forget this argument

```
defstruct Dog :
  name: String
  breed: String
  mood: String with: (setter => set-mood)

defmethod print (o:OutputStream, d:Dog) :
  print(o, "%_ the %_ is %_" % [name(d), breed(d), mood(d)])

val d1 = Dog("Chance", "Pitbull", "Desperate")
val d2 = Dog("Shadow", "Golden Retriever", "Calm")
println(d1)
println(d2)
```

The Match Expression

```
defn what-am-i (x) :
  match(x) :
    (i:Int) : println(“I am an integer”)
    (s:String) : println(“I am a string”)
    (d:Dog) : println(“I am a dog”)
    (c:Char) : println(“I am a character”)
    (y) : println(“<EXISTENTIAL CRISIS>”)

what-am-i(42)
what-am-i(“Pumbaa”)
what-am-i(Dog(“Shadow”, “Golden Retriever”, “Calm”))
what-am-i(‘Z’)
what-am-i(true)
```

Matching Multiple Types

```
defn what-are-these (x, y) :
  match(x, y) :
    (i1:Int, i2:Int) : println("Two integers.")
    (i:Int, s:String) : println("An integer and a string")
    (s:String, i:Int) : println("A string and an integer")
    (s1:String, s2:String) : println("Two strings")
    (x, y) : println("Something else")

what-are-these(10, 42)
what-are-these(10, "Pumbaa")
what-are-these("Timon", 42)
what-are-these("Timon", "Pumbaa")
what-are-these(10, 'Z')
```

Match is Fundamental

```
if 10 < 20 :
  println("Ten is less than twenty")
else :
  println("Ten is not less than twenty")
```


```
match(10 < 20) :
  (r:True) :
    println("Ten is less than twenty")
  (r:False) :
    println("Ten is not less than twenty")
```

Match is Fundamental

```
val dog? = x is Dog
println(dog?)
```


```
val dog? =
  match(x) :
    (x:Dog) : true
    (x) : false
println(dog?)
```

Casts

```
defn bark (x:Dog) :
  println("Woof")

defn cuteness (x) :
  if x is Dog : 100
  else : -100

defn test (x:Cat|Dog) :
  if cuteness(x) > 0 :
    bark(x)
```

Cannot call function bark of type Dog -> False
with arguments of type (Cat|Dog).

Casts

```
defn bark (x:Dog) :
  println("Woof")

defn cuteness (x) :
  if x is Dog : 100
  else : -100

defn test (x:Cat|Dog) :
  if cuteness(x) > 0 :
    bark(x as Dog)
```

Explicit Type Cast

Function Overloading

```
defn length (xs:Array) :
  ...
defn length (str:String) :
  ...
```

```
val xs = Array<Int>(10)
val str = “Hello World”

println(length(xs))
println(length(str))
```

Stanza will figure out which one you’re calling

Function Overloading

```
defstruct Dog
defstruct Tree
defstruct Captain

defn bark (d:Dog) -> False :
   println("Woof!")

defn bark (t:Tree) -> String :
   "Furrowed Cork"

defn bark (c:Captain) -> False :
   println("A teeeen-hut!")
```

Multiple overloaded definitions

Tuples

```
val xs = [1, 2, “Hello World”]
```

Tuple Creation

```
val [a, b, c] = xs
```

Tuple Destructuring

```
length(xs)
```

Tuple Length

```
val i = 1
xs[i]
```

Tuple Element

```
to-array<Int|String>(xs)
```

Tuple Conversion

Tuples are Stanza’s *only* variable-arity expression.

Tuples

Tuple Type

```
defn rectangle (dim: [Int, Int, Int, Int]) :
  val [x, y, w, h] = dim
  println("Rectangle at (%_, %_)" % [x, y])
  println("Size is %_ x %_" % [w, h])

rectangle([10, 10, 42, 72])
```

Tuples

Tuple Type with Unknown Length

```
defn sum-of-nums (nums: Tuple<Int>) :
  var accum = 0
  for x in nums do :
    accum = accum + x
  accum

sum-of-nums([1 2 3])
sum-of-nums([1 2 3 4 5 6 1 42 1])
```

Packages

animals.stanza

```
defpackage animals :
  import core

public defstruct Dog :
  name: String

public defn walk (d:Dog) :
  println("Let's take %_ for a walk." % [d])
```

Start of animals package

Import the core definitions

Only public definitions can be seen outside the package

mainprogram.stanza

```
defpackage mainprogram :
  import core
  import animals

for dog in [Dog("Chance"), Dog("Shadow")] do :
  walk(dog)
```

Start of mainprogram package

Import the core and animals definitions

```
stanza animals.stanza mainprogram.stanza -o animalprogram
```

Vectors

```
defpackage mypackage :
  import core
  import collections

val xs = Vector<String>()

add(xs, “Timon”)

val x = pop(xs)

length(xs)

xs[4]

xs[4] = “Pumbaa”
```

Don’t forget to import collections

Creating a Vector for holding strings

Add a new item to a Vector

Pop off the last item in a Vector

Current length of a Vector

Retrieve a value at a specific index

Set a value at a specific index

Vectors are essentially resizable arrays.

HashTable

```
defpackage mypackage :
  import core
  import collections

val age = HashTable<String, Int>()

age[“Patrick”] = 28

age[“Patrick”]

key?(age, “Luca”)

for entry in age do :
  val name = key(entry)
  val i = value(entry)
  println(i)
```

Don’t forget to import collections

A table that associates strings with integers

Associate “Patrick” with 28

Retrieve the number associated with “Patrick”

Is there anything associated with “Luca”?

Iterate through all entries in the table

The key of the entry

The value of the entry

HashTables associate keys with values

Architecting Programs

Architecting Programs

```
defpackage shapes :
  import core
  import math

public defstruct Point :
  x: Double
  y: Double

public defstruct Circle :
  x: Double
  y: Double
  radius: Double

defmethod print (o:OutputStream, p:Point) :
  ...
defmethod print (o:OutputStream, c:Circle) :
  ...
```

shapes package

Two different shapes

Custom printing behaviour

Architecting Programs

```
defpackage shapes :
  import core
  import math

...

public defn area (s:Point|Circle) -> Double :
   match(s) :
      (s:Point) : 0.0
      (s:Circle) : PI * radius(s) * radius(s)
```

shapes package

Function for computing area

Architecting Programs

```
defpackage shapes/main :
  import core
  import collections
  import shapes

public defn total-area (ss:Vector<Point|Circle>) -> Double :
  var total = 0.0
  for s in ss do :
    total = total + area(s)
  total

defn main () :
  val ss = Vector<Point|Circle>()
  add(ss, Point(1.0, 1.0))
  add(ss, Circle(2.0, 2.0, 3.0))
  add(ss, Circle(3.0, 0.0, 1.0))

  println(total-area(ss))

main()
```

shapes / main package

Function for computing total area

Vector of shapes

Defining a New Shape

Let's define a new kind of shape: Rectangles.
What do we need to change to support Rectangles?

Architecting Programs

```
defpackage shapes :
  import core
  import math

...

public defstruct Rectangle :
  x: Double
  y: Double
  width: Double
  height: Double

defmethod print (o:OutputStream, r:Rectangle) :
  ...
```

1. Struct for Rectangles

2. Custom printing behaviour

Architecting Programs

```
defpackage shapes :
  import core
  import math

...

public defn area (s:Point|Circle|Rectangle) -> Double :
  match(s) :
    (s:Point) : 0.0
    (s:Circle) : PI * radius(s) * radius(s)
    (s:Rectangle) : width(s) * height(s)
```

3. Change type signature to accept Rectangle

4. Add branch to support Rectangle

Architecting Programs

```
defpackage shapes/main :
  import core
  import collections
  import shapes

public defn total-area (ss:Vector<Point|Circle|Rectangle>) -> Double :
  var total = 0.0
  for s in ss do :
    total = total + area(s)
  total

defn main () :
  val ss = Vector<Point|Circle|Rectangle>()
  add(ss, Point(1.0, 1.0))
  add(ss, Circle(2.0, 2.0, 3.0))
  add(ss, Circle(3.0, 0.0, 1.0))
  add(ss, Rectangle(0.0, 0.0, 10.0, 11.0))

  println(total-area(ss))

main()
```

5. Change type signature to accept Rectangles

6. Allow vector to contain Rectangles

Subtyping

```
defpackage shapes :
  import core
  import math

...

public deftype Shape

public defstruct Point <: Shape :
  ...
public defstruct Circle <: Shape :
  ...
public defstruct Rectangle <: Shape :
  ...
```

Abstract type for representing shapes

Points, Circles, and Rectangles are subtypes of Shape

X <: Y
roughly translates to
"an X is a type of Y"

Subtyping

```
defpackage shapes :
  import core
  import math

...

public defn area (s:Shape) -> Double :
  match(s) :
    (s:Point) : 0.0
    (s:Circle) : PI * radius(s) * radius(s)
    (s:Rectangle) : width(s) * height(s)
```

area works on any type of Shape

Subtyping

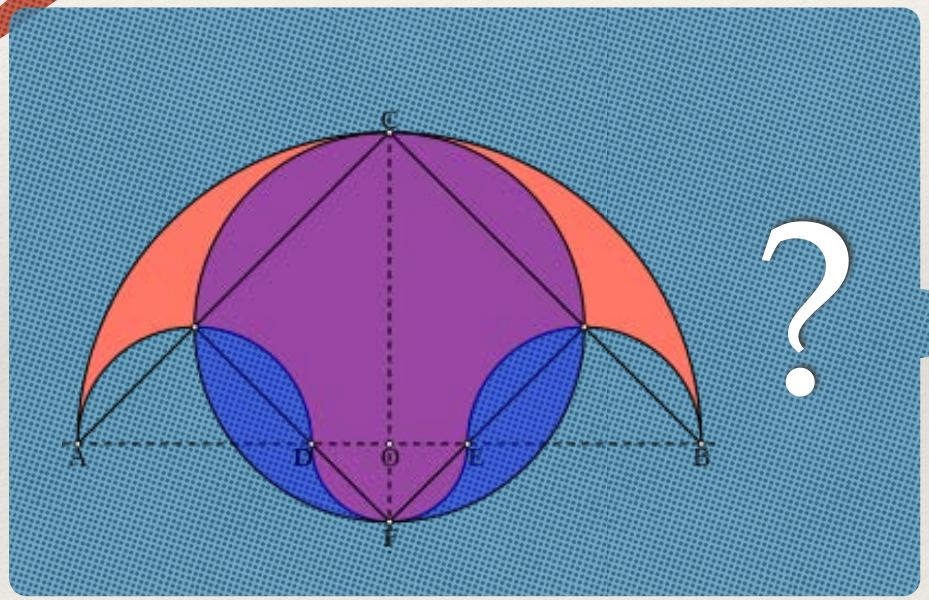
```
defpackage shapes/main :
  import core
  import collections
  import shapes

public defn total-area (ss:Vector<Shape>) -> Double :
  var total = 0.0
  for s in ss do :
    total = total + area(s)
  total

defn main () :
  val ss = Vector<Shape>()
  add(ss, Point(1.0, 1.0))
  add(ss, Circle(2.0, 2.0, 3.0))
  add(ss, Circle(3.0, 0.0, 1.0))
  add(ss, Rectangle(0.0, 0.0, 10.0, 11.0))

  println(total-area(ss))

main()
```

total-area works on vectors containing any type of Shape

Vector can contain any type of Shape

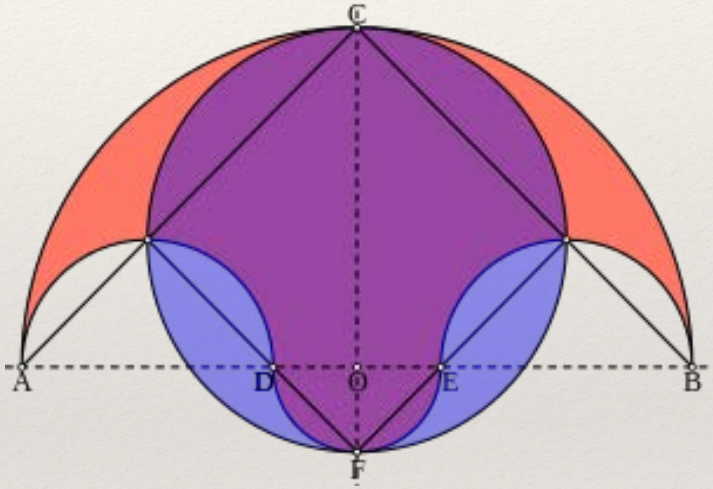
Extensibility

shapes.stanza!



Extensibility

```
defpackage greek-shapes :
  import core
  import math
  import shapes

...
public defstruct Salinon <: Shape :
  x: Double
  y: Double
  outer-radius: Double
  inner-radius: Double

defmethod print (o:OutputStream, s:Salinon) :
  ...
```



Extensibility

```
defpackage shapes :
  import core
  import math

...

public defn area (s:Shape) -> Double :
  match(s) :
    (s:Point) : 0.0
    (s:Circle) : PI * radius(s) * radius(s)
    (s:Rectangle) : width(s) * height(s)

    (s:Salinon) : ????
```

What about this?

Extensibility

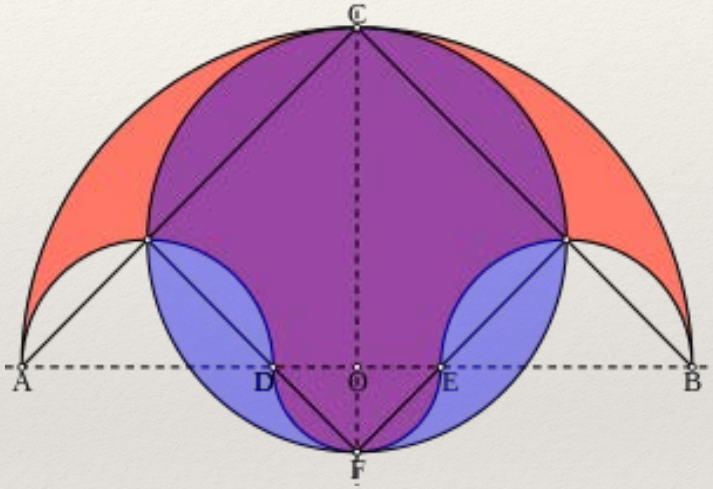
```
defpackage shapes :
  import core
  import math

...

public defmulti area (s:Shape) -> Double

defmethod area (p:Point) -> Double :
  0.0

defmethod area (c:Circle) -> Double :
  PI * radius(c) * radius(c)

defmethod area (r:Rectangle) -> Double :
  width(r) * height(r)
```

The multi definition for area

methods for area

Extensibility

```
defpackage greek-shapes :
  import core
  import math
  import shapes

...
public defstruct Salinon <: Shape :
  x: Double
  y: Double
  outer-radius: Double
  inner-radius: Double

defmethod print (o:OutputStream, s:Salinon) :
  ...

defmethod area (s:Salinon) :
  val r = outer-radius(s) + inner-radius(s)
  PI * r * r / 4.0
```



User-provided method for area

Subtyping

```
defpackage shapes/main :
  import core
  import collections
  import shapes
  import greek-shapes

public defn total-area (ss:Vector<Shape>) -> Double :
  ...

defn main () :
  val ss = Vector<Shape>()
  add(ss, Point(1.0, 1.0))
  add(ss, Circle(2.0, 2.0, 3.0))
  add(ss, Circle(3.0, 0.0, 1.0))
  add(ss, Rectangle(0.0, 0.0, 10.0, 11.0))
  add(ss, Salinon(5.0, -1.0, 8.0, 7.0))

  println(total-area(ss))

main()
```

total-area is unchanged

User-defined Shape

First-class Functions

Nested Functions

```
defn selection-sort (xs:Array<Int>) :
   val n = length(xs)
   for i in 0 to (n - 1) do :
      var min-idx = i
      var min-val = xs[i]
      for j in (i + 1) to n do :
         if xs[j] < min-val :
            min-idx = j
            min-val = xs[j]

      if i != min-idx :
         xs[min-idx] = xs[i]
         xs[i] = min-val
```

Finds the index of the minimum smaller between i and n

Swaps the number at i with the number at min-idx

Nested Functions

```
defn selection-sort (xs:Array<Int>) :
   defn index-of-min (start:Int, end:Int) :
      var min-idx = start
      var min-val = xs[start]
      for i in (start + 1) to end do :
         if xs[i] < min-val :
            min-idx = i
            min-val = xs[i]
      min-idx
```

Finds the index of the minimum smaller between start and end

```
   defn swap (i:Int, j:Int) :
      if i != j :
         val xs-i = xs[i]
         val xs-j = xs[j]
         xs[i] = xs-j
         xs[j] = xs-i
```

Swaps the number at i with the number at j

```
   val n = length(xs)
   for i in 0 to (n - 1) do :
      swap(i, index-of-min(i, n))
```

Overall algorithm

Functions as Arguments

```
defn sort-by-abs (xs:Array<Int>) :
   defn index-of-min (start:Int, end:Int) :
      var min-idx = start
      var min-val = xs[start]
      for i in (start + 1) to end do :
         if abs(xs[i]) < abs(min-val) :
            min-idx = i
            min-val = xs[i]
      min-idx

   defn swap (i:Int, j:Int) :
      if i != j :
         val xs-i = xs[i]
         val xs-j = xs[j]
         xs[i] = xs-j
         xs[j] = xs-i

   val n = length(xs)
   for i in 0 to (n - 1) do :
      swap(i, index-of-min(i, n))
```

Functions as Arguments

```
defn sort-by-sum-of-digits (xs:Array<Int>) :
   defn index-of-min (start:Int, end:Int) :
      var min-idx = start
      var min-val = xs[start]
      for i in (start + 1) to end do :
         if sum-of-digits(xs[i]) < sum-of-digits(min-val) :
            min-idx = i
            min-val = xs[i]
      min-idx

   defn swap (i:Int, j:Int) :
      if i != j :
         val xs-i = xs[i]
         val xs-j = xs[j]
         xs[i] = xs-j
         xs[j] = xs-i

   val n = length(xs)
   for i in 0 to (n - 1) do :
      swap(i, index-of-min(i, n))
```

```
defn sum-of-digits (n:Int) :
   if n == 0 : 0
   else if n < 0 : sum-of-digits((- n))
   else : (n % 10) + sum-of-digits(n / 10)
```

Functions as Arguments

```
defn sort-by (key:Int -> Int, xs:Array<Int>) :
   defn index-of-min (start:Int, end:Int) :
      var min-idx = start
      var min-val = xs[start]
      for i in (start + 1) to end do :
         if key(xs[i]) < key(min-val) :
            min-idx = i
            min-val = xs[i]
      min-idx

   defn swap (i:Int, j:Int) :
      if i != j :
         val xs-i = xs[i]
         val xs-j = xs[j]
         xs[i] = xs-j
         xs[j] = xs-i

   val n = length(xs)
   for i in 0 to (n - 1) do :
      swap(i, index-of-min(i, n))
```

Functions as Arguments

```
defn sort-by (key:Int -> Int, xs:Array<Int>) :
  ...

defn identity (x:Int) :
  x

defn sum-of-digits (n:Int) :
  if n == 0 : 0
  else if n < 0 : sum-of-digits((- n))
  else : (n % 10) + sum-of-digits(n / 10)

defn absolute-value (n:Int) :
  if n < 0 : (- n)
  else : n
```

```
val xs = Array<Int>(10)
...
sort-by(identity, xs)
sort-by(absolute-value, xs)
sort-by(sum-of-digits, xs)
```

Function Types

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

Call with 2 integers and returns a string

```
() -> String
```

Call with no arguments and returns a string

```
() -> ?
```

Call with no arguments and can return anything

```
Int -> String
```

Call with an integer and returns a string

```
? -> String
```

Call with anything and returns a string

Functions as Return Values

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

  defn extract-digit (x:Int, n:Int) -> Int :
    if x < 0 : extract-digit((- x), n)
    else if n == 0 : x % 10
    else : extract-digit(x / 10, n - 1)

  defn extract-digit-n (x:Int) -> Int :
    extract-digit(x, n)

  extract-digit-n
```

```
val first-digit = digit(0)
first-digit(413)
first-digit(-8)
first-digit(19)
```

Functions as Return Values

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

  defn extract-digit (x:Int, n:Int) -> Int :
    if x < 0 : extract-digit((- x), n)
    else if n == 0 : x % 10
    else : extract-digit(x / 10, n - 1)

  defn extract-digit-n (x:Int) -> Int :
    extract-digit(x, n)

  extract-digit-n
```

```
val xs = Array<Int>(10)
...
sort-by(digit(0), xs)
sort-by(digit(1), xs)
sort-by(digit(2), xs)
```

Anonymous Functions

```
defn for-family (f: String -> ?) :
  for x in [“Patrick” “Luca” “Emmy” “Sunny” “Whiskey” “Rummy”] do :
    f(x)

defn say-hello (name:String) :
  println(“Hello %_” % [name])

defn say-bye (name:String) :
  println(“Bye %_” % [name])

for-family(say-hello)
for-family(say-bye)
```

Anonymous Functions

```
defn for-family (f: String -> ?) :
  for x in [“Patrick” “Luca” “Emmy” “Sunny” “Whiskey” “Rummy”] do :
    f(x)

for-family(
  fn (name) :
    println(“Hello %_” % [name]))

for-family(
  fn (name) :
    println(“Bye %_” % [name]))
```

Anonymous Functions

```
defn for-family (f: String -> ?) :
  for x in [“Patrick” “Luca” “Emmy” “Sunny” “Whiskey” “Rummy”] do :
    f(x)

for-family({println(“Hello %_” % [_])})

for-family({println(“Bye %_” % [_])})
```

Anonymous Functions

```
defn for-family (f: String -> ?) :
  for x in [“Patrick” “Luca” “Emmy” “Sunny” “Whiskey” “Rummy”] do :
    f(x)

for-family(println{“Hello %_” % [_]})

for-family(println{“Bye %_” % [_]})
```

Anonymous Functions

```
g{a, b, _, _}
```

equivalent

```
{g(a, b, _, _)}
```

equivalent

```
fn (x, y) :
  g(a, b, x, y)
```

Sequences

Seq<T>

```
val xs = to-seq([1 2 5 “hello” “world”])

empty?(xs)

peek(xs)

next(xs)
```

Convert a tuple into a sequence

Check if the sequence is empty

Peek at the next item in the sequence

Take the next item in the sequence

Collections

```
deftype Collection<T>

defmulti to-seq<?T> (c:Collection<?T>) -> Seq<T>
```

```
Tuple<T> <: Collection<T>
Array<T> <: Collection<T>
List<T> <: Collection<T>
Vector<T> <: Collection<T>
HashTable<K,V> <: Collection<KeyValue<K,V>>
String <: Collection<Char>
```

Collections

```
defn sum-of-numbers (xs:Collection<Int>) :
  val xs-seq = to-seq(xs)
  var accum = 0
  while not empty?(xs-seq) :
    accum = accum + next(xs-seq)
  accum
```

```
val xs = Vector<Int>()
val ys = Array<Int>(10)
val zs = [1 2 3 4 1 42 17]
...
sum-of-numbers(xs)
sum-of-numbers(ys)
sum-of-numbers(zs)
```

Collections

```
defn sum-of-numbers (xs:Collection<Int>) :
  var accum = 0
  for x in xs do :
    accum = accum + x
  accum
```

```
val xs = Vector<Int>()
val ys = Array<Int>(10)
val zs = [1 2 3 4 1 42 17]
...
sum-of-numbers(xs)
sum-of-numbers(ys)
sum-of-numbers(zs)
```

Exercises

- Read *Stanza By Example* and **follow along** with these chapters:
  - Getting Started
  - The Very Basics
  - Architecting Programs
  - Programming with First-class Functions
  - Programming with Sequences
- End of chapter exercises can be skipped if you feel like you understand. But work through the chapter examples.