Stanza by Example

Patrick S. Li

August 25, 2016

Foreword

Stanza by Example is an introductory book for teaching readers how to program in the L.B. Stanza programming language. Readers are assumed to have basic programming experience, at about the level required to implement and understand a basic sorting algorithm. This book is not a reference book, and is meant to be read in order from front to back. The material is written expecting readers to follow along with the coding examples and to the suggested exercises. By following the book, readers will gain a thorough understanding of Stanza's fundamental mechanisms and coding style.

For absolute beginners to programming, the pace of the first chapter will feel a bit fast, and readers are encouraged to take their time to understand and *experiment* with the examples. When I was young, I taught myself to program by reading *Beginning Java 2* by Ivor Horton, and one of the goals of this book is to help beginners get started with programming in the same way that Mr. Horton's book has helped me. Once you get the hang of it, programming is an extremely creative and satisfying endeavor.

I hope you enjoy the book, and Stanza.

-Patrick

Contents

Chapter 1

Getting Started

This chapter explains how to download and install Stanza for your system, compile the example programs, and also write your own very first Stanza program.

1.1 Get Stanza

Download Stanza

Navigate to `www.lbstanza.org`, go to the Downloads section of the webpage, and download the zip file containing the Stanza compiler for your platform. Unzip the file contents to a folder called `mystanza`. This is the directory where Stanza will be installed.

The main Stanza compiler should be located at

```
mystanza/stanza
```

and the core and collections libraries should be located at

```
mystanza/core/core.stanza
mystanza/core/collections.stanza
```

Installing on Linux and OS-X

If you're on a linux platform, open the terminal and type

```
cd mystanza
./stanza install -platform linux
```

If you're on Mac OS-X, then type instead

```
cd mystanza
./stanza install -platform os-x
```

This creates a `.stanza` file in your home directory that contains the installation directory for Stanza.

Put Stanza in your Path

Type the following if you want to be able to call Stanza from any working directory.

```
sudo ln stanza /usr/local/bin/stanza
```

Installing on Windows

Open `cmd.exe` and type

```
cd mystanza
stanza install -platform windows -path .
```

This creates a `.stanza` file in the `mystanza` directory. Stanza will print out a message telling you to set the `STANZA_CONFIG` environment variable to the installation directory. Additionally, add the `mystanza` directory to the `PATH` environment variable to be able to run stanza from any directory.

Running Stanza on windows additionally requires the MinGW-w64 port of the gcc compiler. Download the `mingw-w64-install.exe` installer from https://sourceforge.net/projects/mingw-w64/ and run it. By default, it is installed in `C:\Program Files\mingw-w64`. Add the MinGW-w64 `bin` directory to the `PATH` environment variable.

At the time of writing, the `bin` directory corresponding to our MinGW-w64 installation was located at

```
C:\Program Files\mingw-w64\x86_64-5.3.0-posix-seh-rt_v4-rev0\mingw64\bin
```

Test

Type the following in the terminal

```
stanza version
```

It should print out the version of the Stanza compiler that you downloaded. If you don't see this, then double check that

1. you downloaded Stanza for the right platform.
2. you installed Stanza with the correct `-platform` flag.
3. you put Stanza on your path.

Compile an Example

Type the following in the terminal

```
cd mystanza
stanza examples/helloworld.stanza -o helloworld
```

This should compile the `helloworld` example that comes with Stanza and generate an executable called `helloworld`. If this does not work, then double check that

1. you are in the `mystanza` folder.
2. you installed Stanza with the correct `-platform` flag.
3. you have the Gnu C compiler installed and can call it by typing `cc` (or `gcc` for Windows) in the terminal.

Run the Example

Type the following to run the compiled executable. It should print out "hello world".

```
./helloworld
```

If you're running Windows, then type either

```
helloworld
```

or

```
helloworld.exe
```

Congratulations! You've successfully installed Stanza! Now try compiling and running the other examples in the `examples` directory.

1.2 Write a Program

Basic Skeleton

Create a folder called `stanzaprojects` and create a file called `hello.stanza` containing

```
defpackage mypackage :
   import core

defn main () :
   println("Timon")
   println("and")
   println("Pumbaa")
```

```
main()
```

Make sure you don't forget the space between the `main` and the `()`! We will explain later why this is important. Compile and run it by typing

```
stanza hello.stanza -o hello
./hello
```

It should print out

```
Timon
and
Pumbaa
```

More println Statements

Surround the call to `main` with the following print statements

```
println("Simba")
main()
println("and Nala")
```

Run the program again and it should print out

```
Simba
Timon
and
Pumbaa
and Nala
```

The program runs in the order that it sees the top-level statements.

Delete the Call to main

Delete the call to `main` entirely.

```
println("Simba")
println("and Nala")
```

Now the program prints out

```
Simba
and Nala
```

If you don't call `main` then it never runs.

Rename main

Rename the `main` function to `hakuna`.

```
defpackage mypackage :
   import core

defn hakuna () :
   println("Timon")
   println("and")
   println("Pumbaa")

hakuna()
```

The program still prints out

```
Timon
and
Pumbaa
```

There is nothing special about the `main` function. Name it whatever you like.

Chapter 2

The Very Basics

This chapter introduces the basic programming constructs in Stanza. After this chapter, you'll be able to write basic programs that do simple things.

2.1 Project Framework

Follow these steps to set up a project framework.

Create basics.stanza

In your `stanzaprojects` directory, create a file called `basics.stanza` containing

```
defpackage mypackage :
   import core

defn main () :
   println("Code goes here")

main()
```

Again, make sure you do not forget the space between the `main` and the `()`. We'll explain why this is necessary when we discuss Stanza's lexical structure.

Compile and Run

Compile and run the basic framework by typing the following in the terminal

```
stanza basics.stanza -o basics
./basics
```

The basic framework should print out

```
Code goes here
```

Follow the chapter and try out the examples by replacing the `println` command with the example code.

Indentation

Try changing the basic framework to

```
defpackage mypackage :
import core

defn main () :
   println("Code goes here")

main()
```

and try to compile it. It won't work. My Stanza installation says

```
Syntax Error: Import clause expected here.
```

Indentation is important in Stanza programs. Be careful when trying out the examples.

And don't use tabs. Stanza won't let you. We don't like tabs.

2.2 Printing Simple Messages

Printing is important. It's the only way to observe what your program is doing.

Strings

This is a *string*.

```
"Timon and Pumbaa"
```

It's a bunch of characters surrounded in double quotes.

Printing Strings

Use the `println` function to print strings.

```
println("Timon")
println("and")
println("Pumbaa")
```

prints out

```
Timon
and
Pumbaa
```

Print Without a New Line

Use the `print` function to print strings without starting a new line at the end.

```
print("Timon")
print(" and")
print(" Pumbaa")
```

prints out

```
Timon and Pumbaa
```

Ints

This is an *integer*.

```
42
```

It's a bunch of digits, and represents the integers that you were taught in school.

Printing Integers

The `print` and `println` function works on integers too.

```
print(1)
print(" and a ")
print(2)
print(" and a ")
println(1)
println(2)
println(3)
println(4)
```

prints out

```
1 and a 2 and a 1
2
```

```
3
4
```

Actually `print` and `println` works on a lot of things. We'll learn about that later.

Printing Multiple Things

Calling `println` repeatedly to print multiple things is tedious. Use `println-all` to print out multiple things.

```
println-all([1 " and a " 2 " and a "])
println-all([1 2 3 4 "."])
```

prints out

```
1 and a 2 and a
1234.
```

If you don't want to start a new line at the end, use `print-all` instead.

```
print-all([1 " and a " 2 " and a "])
println-all([1 2 3 4 "."])
```

prints out

```
1 and a 2 and a 1234.
```

Don't fret about the `[]` brackets for now. They create *tuples*. We'll learn about those later.

Formatted Printing

Sometimes it's tedious to print multiple things even with `println-all`. Here's how to print things according to a *format string*.

```
println("%_ and a %_ and a %_, %_, %_, %_!" % [1 2 1 2 3 4])
```

prints out

```
1 and a 2 and a 1, 2, 3, 4!
```

Notice that you're calling the same `println` function that you've already learned. The `%` operator is what's doing all the work. We'll learn other operators later.

Where's the Commas?

Some of you may have noticed the lack of commas in the examples. Try adding them back in.

```
println("%_ and a %_ and a %_, %_, %_, %_!" % [1, 2, 1, 2, 3, 4])
```

still prints out

```
1 and a 2 and a 1, 2, 3, 4!
```

Commas are treated identically to spaces in Stanza. (Unless they are part of a string.) Try going crazy!

```
println("%_ and a %_ and a %_, %_, %_, %_!" % [1 2,1,2,,,,3,,,,,4])
```

The above still prints out what it used to. But don't do that. That was just an example.

2.3 Lexical Structure

Before compilation, a Stanza program is *lexed* into individual identifiers, numbers, and lists. Here are the rules that you'll need to know.

Lexemes

The first thing that Stanza does is break down a program into a sequence of *lexemes*, where each lexeme is separated by either whitespace or one of the following characters.

```
, . : & | < > [ ] { } ( )
```

Numbers

A number is a lexeme that begins with a digit, or a hyphen followed by a digit. Here are some examples.

```
3
50
-13
100L
```

Operators

An operator is any lexeme that is made up of the following characters.

```
~ ! @ \# \$ % ^ * + - = / . : & | < >
```

Here are some example operators.

```
+
*
&
&&
```

```
+=
>>>
<+>
::=
<^.^>
```

Identifiers

An identifier is any lexeme that is not a number or operator. Here are some examples.

```
x
timon
timon_and_pumbaa
timon-and-pumbaa
\#timon
\$timon
timon?
x+one
x1
x+1-3
```

Opening Brackets

Syntactically, an identifier followed *immediately* by a opening bracket character is treated *differently* than if the two were separated by spaces. For example

```
f(x)
```

is syntactically different than

```
f (x)
```

The former calls the function `f` with `x`. The latter is simply the function `f` followed by the value `x`.

This is similar to how

```
ab
```

is syntactically different than

```
a b
```

The two mean different things. Please keep this in mind when following the examples in this book. For example, this is why it was stressed to you to remember the space after `main` in

```
defn main () :
   ...
```

Additionally,

`f[x]`

is syntactically different than

`f [x]`

and

`f{x}`

is syntactically different than

`f {x}`

and

`f<x>`

is syntactically different than

`f <x>`

2.4 Comments

Comments begin with the `;` character, and every following character in the line will be regarded as a comment and won't affect the behaviour of the code. Here's an example of using comments.

```
;My favorite characters
println("Timon")  ;The small one
println("and")
println("Pumbaa") ;and the smart one
```

Code without comments is very difficult to understand. Use comments often.

2.5 Operators

Basic Arithmetic

To add two numbers together, use the `+` operator.

`println(10 + 32)`

prints out

```
42
```

That means the result of adding `32` to `10` is `42`. We say that the expression `10 + 32` *returns* `42`. We'll learn later why we use the word *returns*.

Here are examples of using the other arithmetic operators.

```
68 + 32 ;Addition
68 - 32 ;Subtraction
68 * 32 ;Multiplication
68 / 32 ;Division
68 % 32 ;Modulus
```

Bitwise Arithmetic

Here are examples of using the bitwise operators.

```
24 << 2 ;Left Bit Shift
24 >> 2 ;Right Bit Shift
24 >>> 2 ;Arithmetic Right Bit Shift
24 & 2 ;Bitwise And
24 | 2 ;Bitwise Or
24 ^ 2 ;Bitwise Xor
```

Operator Precedence

All the operators are *left associative*. This means that in the following expression

```
1 + 2 - 3 + 4 - 5 + 6
```

the operators are applied left to right. The above is equivalent to

```
((((1 + 2) - 3) + 4) - 5) + 6
```

In expressions containing a mix of operators, the operators with highest precedence are grouped together first, followed by the operators with second highest precedence, until you reach the operators with lowest precedence. The shift operators (`<<`, `>>`, `>>>`) have precedence 3. The multiply, divide, modulo, bitwise and, and bitwise xor operators (`*`, `/`, `%`, `&`, `^`) have precedence 2. Addition, subtraction, and bitwise or, (`+`, `-`, `|`), have precedence 1.

The following expression

```
3 + 2 << 2 * 3 + 3 << 1
```

first groups the precedence 3 operators

```
3 + (2 << 2) * 3 + (3 << 1)
```

followed by the precedence 2 operators

```
3 + ((2 << 2) * 3) + (3 << 1)
```

followed by the precedence 1 operators

```
(3 + ((2 << 2) * 3)) + (3 << 1)
```

Unary Operators

Here's how to negate a number.

```
(- 3)
```

The parentheses are not optional!

Here's how to flip all the bits in a number.

```
(~ 3)
```

Again, the parentheses are not optional! This is different than most other languages. There's a good reason for this. But don't forget them!

2.6 Values

Syntax

The statement

```
val a:Int = 3 * 71
```

calculates the result of `3 * 71` and *stores* the result in the *value* `a`.

After the storing the result in `a`, you can then use that value afterwards by name.

```
println(a)
```

You cannot change what is stored in a value once it is initialized.

Breaking up Complicated Expressions

You can use values to break up complicated expressions into smaller ones.

```
println(1 + 30 * 2 * 3 - 30 / (3 << 1))
```

can be rewritten as

```
val a:Int = 30 * 2 * 3
val b:Int = 30 / (3 << 1)
println(1 + a - b)
```

Types

The `Int` in the previous example is called a *type annotation*. It says that only an *integer* can be stored in `a`.

Stanza won't let you store anything else into `a`. We can try to store a string

```
val a:Int = "Timon"
```

but attempting to compile it gives us this error.

```
Cannot assign expression of type String to value a with declared type Int.
```

If you want to store a string into `a`, then you have to declare it like this.

```
val a:String = "Timon"
```

Later, we will learn more about types and about types other than `Int` and `String`.

Type Inference

If you leave off the type annotation

```
val a = 3 * 71
```

then Stanza figures out the type based on the expression it's initialized with.

Most Stanza programmers leave off type declarations for values.

2.7 Variables

A variable is declared like a value, but using `var` instead of `val`.

```
var a:Int = 10 + 30
```

Just like a value, the result of calculating `10 + 30` is stored in the variable `a`.

And just like a value, you can refer to it by name afterwards.

```
println(a)
```

The difference is that, even after it is initialized, you can still store something *else* into a variable.

```
var a:Int = 3 * 71
println(a)
a = -10
println(a)
```

prints out

```
213
-10
```

After we print out `a` for the first time, we use the `=` operator to store `-10` into `a`. The second time we print out `a`, it prints out `-10`.

Types

Just like values, a variable's type annotation restricts what you can store inside it. Here's what happens when we attempt to store a string into `a`.

```
var a:Int = 3 * 71
a = "Timon"
```

Compiling the above gives us

```
Cannot assign expression of type String to variable a with declared type Int.
```

Type Inference

Just like values, you can leave off the type annotation for a variable.

```
var a = 3 * 71
println(a)
a = -10
println(a)
```

However, the inferred type for a variable depends upon *all* the values assigned to it, not just the initial one.

For various reasons, Stanza cannot always infer the type of a variable, as in this example. (Don't mind the functions you don't know. We'll learn them later.)

```
var a = 3 * 71
a = cons(a, List())
```

Attempting to compile the above gives us this error.

```
Could not infer type of variable a.
```

In these cases, you'll have to provide an explicit type annotation for the variable.

Uninitialized Variables

Variables don't have to be declared with an initial value. Here's a variable, declared to only hold integers, but with no initial value.

```
var x:Int
```

Attempting to read from an uninitialized variable will crash the program. The following program

```
var x:Int
println(x)
```

when compiled and ran crashes with this error.

```
FATAL ERROR: Variable is uninitialized.
```

2.8 Functions

Here's a function that subtracts forty two from its argument.

```
defn subtract-forty-two (x:Int) -> Int :
   x - 42
```

And here's how you *call* the function.

```
subtract-forty-two(43)
```

Here's the complete program.

```
defpackage mypackage :
   import core

defn subtract-forty-two (x:Int) -> Int :
   x - 42

println(subtract-forty-two(43))
```

It prints out:

```
1
```

This means that the result of calling `subtract-forty-two` with `43` is `1`. We say that `subtract-forty-two(43)` *returned* `1`.

Return Value

Here's a silly change we can make to `subtract-forty-two`.

```
defn subtract-forty-two (x:Int) -> Int :
   x + 43
   x - 42
```

`subtract-forty-two(43)` still returns `1` though. The result of the last expression in a function's body is the value returned by the function.

Side Effects

Here's another change we can make to `subtract-forty-two`.

```
defn subtract-forty-two (x:Int) -> Int :
   println("Subtracting 42 from %_." % [x])
   x - 42
```

Now the following code

```
println(subtract-forty-two(43))
```

prints

```
Subtracting 42 from 43.
1
```

The expressions in a function body are evaluated one at a time, but only the result of the last one is returned.

Return Type

The `Int` following the `->` in `subtract-forty-two` is the function's *return type*. It says that the function must return an integer.

Stanza won't let you return anything else. If we try to return a string

```
defn subtract-forty-two (x:Int) -> Int :
   println("Subtracting 42 from %_." % [x])
   "Timon"
```

then the Stanza compiler gives us this error.

```
Cannot return an expression of type String for function
subtract-forty-two with declared return type Int.
```

You can leave off the return type annotation

```
defn subtract-forty-two (x:Int) :
   println("Subtracting 42 from %_." % [x])
   x - 42
```

in which case, Stanza will figure out the return type automatically based on the last expression in the function body. In certain cases, Stanza will not be able to figure it out and you'll have to provide it explicitly.

Argument Types

The `:Int` following the `x` argument in `subtract-forty-two` is the type annotation for the argument. This type annotation does two things. The first is that it restricts what values you can call `subtract-forty-two` with. Compiling the following code

```
subtract-forty-two("Hello")
```

gives the error

```
Cannot call function subtract-forty-two of type Int -> Int
with arguments of type (String).
```

The second is that it restricts what you are allowed to do with `x`. Here is what happens if we try to use `x` as if it were a format string.

```
defn subtract-forty-two (x:Int) :
   println(x % [1, 2, 3])
   x - 42
```

Compiling it gives us this error.

```
No appropriate function modulo for arguments of type
(Int, [Int, Int, Int]). Possibilities are:
   modulo: (String, Seqable) -> Printable at core/core.stanza:1982.12
   modulo: (Byte, Byte) -> Byte at core/core.stanza:2444.21
   modulo: (Int, Int) -> Int at core/core.stanza:2575.12
   modulo: (Long, Long) -> Long at core/core.stanza:2644.21
```

Roughly, that error message tells us that there are four different functions called *modulo*, and none of them can called with `x` and `[1, 2, 3]`. We'll learn later how to interpret that error message more precisely.

Example: String Arguments

Here is an example of a function that accepts a string as an argument.

```
defn timon-and-pumbaa-says (format:String) :
   println(format % ["Timon", "Pumbaa"])

timon-and-pumbaa-says(
   "%_ says they're fireflies, while %_ says they're big balls of gas.")

timon-and-pumbaa-says(
   "When the world turns their back on %_, %_ turns their back on the world.")
```

Compiling and running the above prints

```
Timon says they're fireflies, while Pumbaa says they're big balls of gas.
```

```
When the world turns their back on Timon, Pumbaa turns their back on the world.
```

We did not provide an explicit return type for `timon-and-pumbaa-says`. Thus Stanza figures it out automatically from the result of the last expression, the `println`. It turns out that `println` returns the value `false` which has type `False`. We could explicitly provide the return type as well.

```
defn timon-and-pumbaa-says (format:String) -> False :
   println(format % ["Timon", "Pumbaa"])
```

Leaving off the Argument Type

Beware. Argument types are *not* inferred automatically. You can leave off the argument types

```
defn subtract-forty-two (x) :
   x - 42
```

but this is *not* equivalent to declaring `x` as an `Int`.

If you leave off the type annotation for an argument, it means that the argument can be *anything*. You can call the function with whatever you want, and within the body of the function Stanza will let you do whatever you wish with the argument. If you do something wrong then the program will crash when ran.

Here is an *incorrect* program that compiles correctly.

```
defn subtract-forty-two (x) :
   println(x % ["Timon", "Pumbaa"])
   x - 42

subtract-forty-two("%_ and %_ say Hakuna Matata!")
```

But when the program is ran, it crashes with this error.

```
Timon and Pumbaa say Hakuna Matata!
FATAL ERROR: Cannot cast value to type.
   at core/core.stanza:2566.12
   at stanzaprojects/basics.stanza:6.3
   at stanzaprojects/basics.stanza:8.0
```

The error message is saying that in the expression `x - 42` it could not convert `x` into the appropriate type needed by the `-` operator (`Int`).

The Unknown Type

More precisely, leaving off the type annotation for an argument is equivalent to declaring the argument with the `?` type. So the above program can be written equivalently as

```
defn subtract-forty-two (x:?) :
   println(x % ["Timon", "Pumbaa"])
   x - 42

subtract-forty-two("%_ and %_ say Hakuna Matata!")
```

The `?` type is very special and forms the foundation of Stanza's optional type system. You can pass *any* value to a location where a `?` is expected. *And*, you can use a value of `?` type anywhere.

You can use the `?` type in variable and value declarations too. Here is an example of using them with variables.

```
var x:? = "%_ says Hakuna."
println(x % ["Timon"])
x = 10 + 32
println("There are about %_ fireflies in the universe." % [x])
```

It prints out

```
Timon says Hakuna.
There are about 42 fireflies in the universe.
```

Notice that we're using `x` as a format string in one case, and a number in the other case. The `?` type allows us to do this.

2.9 Comparisons

Comparison Operators

To test whether one integer is smaller than another number, you can use the `<` operator. Here's an example.

```
println(10 < 32)
```

It prints out

```
true
```

This means that `10` is less than `32`. The expression `10 < 32` returned the value `true`. Conversely,

```
println(10 > 32)
```

prints out

```
false
```

This means that `10` is not greater than `32`. The expression `10 > 32` returned the value `false`.

Here's all the other comparison operators you can use.

```
10 < 32     ;Less Than
10 <= 32    ;Less Than or Equal to
10 > 32     ;Greater Than
10 >= 32    ;Greater Than or Equal to
10 == 32    ;Equal
10 != 32    ;Not Equal
```

Logical Operators

You can use the `not`, `and`, and `or` operators to combine the results of multiple comparisons. Here is how to test whether `1` is less than `3` *and* greater than `5`.

```
println(1 < 3 and 1 > 5)
```

Here is how to test whether `1` is less than `3` *or* greater than `5`.

```
println(1 < 3 or 1 > 5)
```

Here is how to test whether `1` is *not* less than `3`.

```
println(not 1 < 3)
```

2.10 If Expressions

If expressions let you test whether a value is `true` or `false` and do different things depending on the result.

```
val x = 10 < 32
if x :
   println("Timon is better!")
else :
   println("Pumbaa is better!")
```

prints out

```
Timon is better!
```

The result of `10 < 32` (`true`) is stored in `x`. Then because `x` (the *predicate*) is `true`, the *consequent* branch of the if expression is evaluated instead of the *alternate* branch.

Change the `<` operator to a `>` operator to root for Pumbaa instead.

Result of an If Expression

If expressions evaluate to a result. The *result* of an if expression is the result of the last expression in the consequent branch if the predicate is `true`. Otherwise it is the result of the last expression in the alternate branch.

Here's an example of using the result of an if expression.

```
val x =
   if 10 < 32 :
      "Timon"
   else :
      "Pumbaa"
println("%_ is better!" % [x])
```

It prints out the same message as the last example.

Default Else Branch

If you leave off the `else` branch in an if expression, then the if expression simply evaluates to `false` if the predicate is not `true`. In the following code

```
if 10 > 32 :
   println("Timon is better!")
```

nothing is ever printed.

Nested If Expressions

You can nest if expressions inside other if expressions. The following code prints different messages when x falls in different ranges.

```
val x = 32
if x < 0 :
   println("x is negative!")
else :
   if x < 10 :
      println("x is between 0 and 10")
   else :
      if x < 30 :
         println("x is between 20 and 30")
      else :
         println("x is really big!")
```

It uses nested if expressions to test a series of conditions.

Because nested if expressions are so common, you are allowed to omit the colon after the `else` keyword if it is followed by an `if` expression. The above can be rewritten equivalently as

```
val x = 32
if x < 0 :
   println("x is negative!")
else if x < 10 :
   println("x is between 0 and 10")
else if x < 30 :
   println("x is between 20 and 30")
else :
   println("x is really big!")
```

Here's another example. `sign` is a function that computes the sign of its argument.

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

True and False

The `true` and `false` values can be created directly simply by referring to them by name.

The following

```
val worries? = true
if worries? :
   println("Chill out!")
else :
   println("Hakuna Matata!")
```

prints out

```
Chill out!
```

To print `Hakuna Matata!` instead, change the `true` to `false`.

The value `true` has type `True`, and the value `false` has type `False`.

2.11 Expression Sequences

Multiple expressions can be grouped together as an *expression sequence* by surrounding them with parentheses.

```
val x = (println("A"), 42)
println(x)
```

prints out

```
A
42
```

The expressions in an expression sequence are evaluated one at a a time, and the result of the last expression is the result of the expression sequence.

In the above example, the first expression in the sequence, `println("A")`, is evaluated, and then the last expression, `42`, is the result of the sequence and is stored in `x`. `x` is then printed to the screen.

2.12 Structure Through Indentation

Some of you may be concerned about Stanza's use of structure through indentation due to how this system has been implemented in the past. Don't worry. Stanza's indentation structuring mechanism is very simple and *predictable*.

The indentation structuring mechanism is governed by a single rule: a line ending colon automatically inserts parentheses around the following indented block.

Thus, after the implicit parentheses have been added, the previous `sign` example looks like this.

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

A program with no line ending colons can even be written on a single line if desired.

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

Here is one more example.

```
defn hakuna () :
   println("Timon")
   println("Pumbaa")
```

becomes the following after implicit parentheses are added.

```
defn hakuna () : (
   println("Timon")
   println("Pumbaa"))
```

Here is `hakuna` written out on a single line.

```
defn hakuna () : (println("Timon") println("Pumbaa"))
```

As you may have noticed, the indentation mechanism is simply used as a shorthand for creating expression sequences out of indented blocks.

2.13 While Loops

The following

```
var x = 1
while x < 1000 :
   println("x is %_" % [x])
   x = x * 2
```

prints out

```
x is 1
x is 2
x is 4
x is 8
x is 16
x is 32
x is 64
x is 128
x is 256
x is 512
```

Here is the general form.

```
while predicate : body
```

A while loop repeatedly evaluates a block of code so long as the predicate expression evaluates to `true`.

Here is the order in which the while loop does things.

1. Evaluate the predicate.
2. If the predicate evaluates to `false`, then the loop is done.
3. Otherwise, evaluate the body and then repeat from step 1.

2.14 For "Loops"

Stanza's for construct is *extremely* powerful. "Loops" is in double quotes because, strictly speaking, the for construct is not a looping mechanism. But it is often used as one, so we'll explain it here as if it were. Later, we'll learn the general form of the for construct.

Counting Loops

The following

```
for i in 0 to 4 do :
   println("Pumbaa is Better!")
```

prints out `Pumbaa is Better!` four times.

The following

```
for i in 0 to 4 do :
   println("i is %_" % [i])
```

prints out

```
i is 0
i is 1
i is 2
i is 3
```

A counting loop has this general form.

```
for x in start to end do : body
```

For each integer between `start` (inclusive) and `end` (exclusive), the body is evaluated once with `x` *bound* to that integer.

Range Expressions

In the previous example, the expression `0 to 4` creates a `Range` object. A `Range` object represents a sequence of integers between some starting index and optional ending index.

Here's how to create a `Range` object that counts up in steps of 2.

```
0 to 10 by 2
```

It represents the numbers `0, 2, 4, 6, 8`.

The following

```
for i in 0 to 10 by 2 do :
   println("i is %_" % [i])
```

prints out

```
i is 0
i is 2
i is 4
i is 6
i is 8
```

To make the ending index inclusive rather than exclusive, use the `through` keyword rather than the `to` keyword.

```
0 through 10 by 2
```

represents the numbers `0, 2, 4, 6, 8, 10`.

If you use `false` for the ending index, then the range represents an *infinite* sequence of numbers.

```
0 to false by 3
```

represents the numbers `0, 3, 6, 9, ....`

2.15 Labeled Scopes

For Returning Early

As you've learned so far, functions return the result of the last expression in its body. But what if you want to return earlier?

As an example, here's a function that computes the n'th fibonacci number.

```
defn fibonacci (n:Int) -> Int :
   var a:Int = 0
   var b:Int = 1
   var i = 0
   while i < n :
      val c = a + b
      a = b
      b = c
      i = i + 1
   b
```

Let's use a labeled scope to change `fibonacci` to return `-1` immediately if the argument `n` is negative.

```
defn fibonacci (n:Int) -> Int :
   label<Int> myreturn :
      if n < 0 : myreturn(-1)
```

```
      var a:Int = 0
      var b:Int = 1
      var i = 0
      while i < n :
         val c = a + b
         a = b
         b = c
         i = i + 1
      b
```

The first line within the labeled scope

```
if n < 0 : myreturn(-1)
```

checks to see whether `n` is negative, and if it is, it *immediately* returns the value `-1` from the function by calling the *exit function* `myreturn`.

For Breaking From Loops

Labeled scopes are also useful for breaking early out of loops.

Here's how the while loop in `fibonacci` could have been written.

```
defn fibonacci (n:Int) -> Int :
   var a:Int = 0
   var b:Int = 1
   var i = 0
   label<False> break :
      while true :
         if i == n : break(false)
         val c = a + b
         a = b
         b = c
         i = i + 1
   b
```

The code above starts an infinite loop, but breaks out of it when `i` is equal to `n`.

General Form

The general form of a labeled scope is

```
label<Type> exit :
   body
```

`Type` is the type of the value returned by the labeled scope and `exit` is the name of the exit function.

You can name the exit function whatever you like. When used to return early from a function, `return` is a popular name for the exit function. When used to break early from a loop, `break` is a popular name.

The label construct simply executes the given body. If the exit function is never called then the result of the body expression is the result of the label construct. If the exit function is called, then we *immediately* stop evaluation of the body, and the argument to the exit function is the result of the label construct.

Well-Typed Labeled Scopes

The type annotation on the label construct enforces two properties.

1. The argument to the exit function must be of the specified type.
2. The result of the body itself must be of the specified type, as that is the value that is returned by the label construct if the exit function is never called.

The first restriction is fairly obvious. If you pass an argument of the wrong type to the exit function

```
defn fibonacci (n:Int) -> Int :
   label<Int> myreturn :
      if n < 0 : myreturn("Timon")
      var a:Int = 0
      var b:Int = 1
      var i = 0
      while i < n :
         val c = a + b
         a = b
         b = c
         i = i + 1
      b
```

then Stanza will issue an error.

```
Cannot call function myreturn of type Int -> Void with arguments of type (String).
```

The second restriction sometimes arises in more subtle situations. The following function computes the first integer whose square is greater than 1000.

```
defn first-big-square () :
   label<Int> return :
      for i in 0 to false do :
         if i * i > 1000 :
            return(i)
```

But compiling it gives us this error.

```
Cannot return an expression of type False for anonymous function
with declared return type Int.
```

This message says that the body of the labeled scope returns `False` but it's declared to return `Int`. This arises because the for construct with the `do` operating function returns `false`, but the type annotation on the label construct was `Int`.

Since we *know* that the `return` exit function is guaranteed to be called, we can explicitly handle this case by causing the program to fail if the loop ever finished without calling `return`.

```
defn first-big-square () :
   label<Int> return :
      for i in 0 to false do :
         if i * i > 1000 :
            return(i)
      fatal("Unreachable Statement")
```

2.16 Scopes and the Let Expression

We have now seen a number of expressions that introduce a new *scope*: functions, while loops, for loops, if expressions, and labeled scopes. Values and variables defined within a scope are only visible within that scope. For example, in the following code

```
val x = 3
if x < 5 :
   val y = 10
   println(y)
else :
   val z = 12
   println(z)
```

`y` is in the scope of the consequent branch of the if expression, and it is *only* visible from within the consequent branch of the if expression. And `z` is only visible from within the alternate branch of the if expression. Referencing `y` and `z` from outside the scope in which they were declared

```
val x = 3
if x < 5 :
   val y = 10
   println(y)
else :
   val z = 12
   println(z)

println(y)
```

```
println(z)
```

is illegal and would not pass the Stanza compiler.

Scopes may themselves contain other nested scopes. In the above example, `x`'s scope, contains both the scope of the consequent branch, and the scope of the alternate branch of the if statement. At any point in the program, you may only refer to a value or variable defined in a containing scope. The following *is* legal.

```
val x = 3
if x < 5 :
   val y = 10
   println(y)
   println(x)
else :
   val z = 12
   println(z)
   println(x)
```

If there are multiple values with the same name that are visible, you automatically refer to the one in the nearest scope. Thus the following code prints `11`, not `3`.

```
val x = 3
if x < 10 :
   val x = 11
   println(x)
```

This feature is called *shadowing*.

Sometimes it is useful to artificially introduce a new scope, simply because you will define a number of values that you only want visible within the scope. You can do this using the let expression.

```
val x = 3
let :
   val y = 4
   println(y)
```

In the above code, the let expression introduces a new scope where `y` is defined. After the let expression, `y` will no longer be visible.

2.17 Arrays

Arrays are one of Stanza's most fundamental datastructures. The following

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

creates an array of length 10 and gives it the name `a`. You can imagine an array to look like a row of boxes, into which you can put and retrieve objects. So `a` is a row of ten boxes, each capable of holding an integer.

Putting Things In

You can put things into the boxes like this.

```
a[0] = 42
a[1] = 13
```

The first box is numbered box 0. The next box is numbered box 1. The last box in `a` is box 9 because `a` has only ten boxes in total.

Getting Things Out

You can retrieve the contents of boxes like this.

```
println(a[0])
println(a[1])
```

prints out

```
42
13
```

Asking For Its Length

You can call the `length` function to ask for the length of an array.

```
val l = length(a)
println("a has %_ boxes." % [l])
```

prints out

```
a has 10 boxes.
```

The type of `a` is `Array<Int>` indicating that it is an array for holding integers. An array for holding strings would have type `Array<String>`. And an array that can hold anything would have type `Array<?>`.

Arrays and Loops

Arrays are most powerful when combined with loops. This function

```
defn array-sum (xs:Array<Int>) :
   var sum = 0
   for i in 0 to length(xs) do :
      sum = sum + xs[i]
   sum
```

computes the sum of every integer in an array. Let's use it to compute the sum of 10, 11, 7, and 8.

```
val a = Array<Int>(4)
a[0] = 10
a[1] = 11
a[2] = 7
a[3] = 8
println(array-sum(a))
```

prints out

```
36
```

2.18 Tuples

Tuples represent an immutable collection of items. The following creates a two-element type.

```
val t:[Int, String] = [42, "Hello"]
```

It contains an `Int` and a `String`. To extract the elements of a tuple, type

```
val [x, y] = t
```

The above code checks that `t` is a two-element tuple, and then puts the first element of `t` in `x` and the second element of `t` in `y`.

Notice that the type of the expression `[42, "Hello"]` is `[Int, String]`. That type says its a two-element tuple containing an `Int` and a `String`.

Returning Multiple Values

Tuples are often used to return multiple values from a function. The following function takes an argument, `n`, and a distance, `d`, and returns both `n - d` and `n + d`.

```
defn bracket (n:Int, d:Int) :
   [n - d, n + d]
```

We can call and receive both return values from `bracket` like this.

```
val [lo, hi] = bracket(5, 3)
println("Bracket around %_ and %_" % [lo, hi])
```

2.19 Basic Types

At this point, we have seen a couple of different types now. Here is a listing of the other basic types in Stanza.

```
Byte :     e.g. 1Y, 42Y, 255Y
Int :      e.g. 10, 42
Long :     e.g. 10L, 420020020L
Float :    e.g. 1.0f, 42.0f
Double :   e.g. 1.0, 42.0
String :   e.g. "Timon", "Pumbaa"
Char :     e.g. 'a', 'Z'
True :     e.g. true
False :    e.g. false
```

As we've said already, the `?` type is special and any value can be passed to a place expecting a `?`.

All of the types listed in the previous table are examples of *ground* types. It means that you refer to them simply by their name and they don't take any parameters. With the introduction of arrays, you have now also been introduced to your first *parametric* type.

```
Array<Int> :          Arrays of Ints
Array<String> :       Arrays of Strings
Array<Array<Int>> :   Arrays of Arrays of Ints
Array<?> :            Arrays of anything
```

Unlike ground types, parametric types take additional *type parameters*. An array needs to know what type of objects it holds, so it has one type parameter for specifying that.

Note that for parametric types, if you leave off its parameters, then it is equivalent to specifying `?` for all of its type parameters. Thus

```
Array
```

is equivalent to

```
Array<?>
```

And

```
Array<Array>
```

is equivalent to

```
Array<Array<?>>
```

Tuple types have their own syntax, and consists of surrounding the types of all of its elements with the `[]` brackets. Here is a tuple containing an integer and a string.

```
[Int, String]
```

Here is a tuple containing an integer, a string, and a tuple of a single integer.

```
[Int, String, [Int]]
```

2.20 Structs

For convenience, Stanza provides a simple way to create compound types out of existing types using structs. Here is how to define a new `Dog` type with two fields, a name, and a breed.

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

Once `Dog` is defined, you can create new `Dog` objects by calling the `Dog` function.

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

Getter Functions

To retrieve the values of the fields it was constructed from, you may call the `name` and `breed` getter functions.

```
println("%_ is a %_." % [name(d1), breed(d1)])
```

prints out

```
Chance is a Pitbull.
```

Struct Type

Additionally, once a struct is defined, you may now also use it as the name of a type. Here is a function that prints out the contents of an array of `Dog` objects.

```
defn kennel-contents (dogs:Array<Dog>) :
   println("There are %_ dogs in the kennel." % [length(dogs)])
   for i in 0 to length(dogs) do :
      println("%_ the %_" % [name(dogs[i]), breed(dogs[i])])
```

Let's try calling `kennel-contents`.

```
val kennel = Array<Dog>(3)
kennel[0] = Dog("Chance", "Pitbull")
kennel[1] = Dog("Shadow", "Golden Retriever")
kennel[2] = Dog("Bud", "Basketball Player")
kennel-contents(kennel)
```

prints out

```
There are 3 dogs in the kennel.
Chance the Pitbull
Shadow the Golden Retriever
Bud the Basketball Player
```

Structs are just a convenient form for quickly declaring a new type, constructor function, and getter functions. Later we'll learn the general method for creating new types.

2.21 Exercises

1. Write a function called `nearest-pow-2` that takes a single positive integer `n` and returns the closest number to `n` that can be represented as a power of 2.
2. Write a program that prints out different messages depending on a value called `temperature`. If `temperature` is below 20, then print `Too Cold!`. If it is between 20 and 40 then print `Getting There!`, and print `Pumbaa Approves!` if it is over 40.
3. Use a for loop and a variable to compute the sum of all the integers between 100 (inclusive) to 500 (also inclusive).
4. Use a labeled scope and for loops to compute the minimum number, n, for which the sum of the integers between 0 and n (inclusive) is above 10000.
5. Use a while loop and variables to calculate the great common divisor of two variables, `x` and `y`. The Euclidean algorithm is the simplest one to use. Look online for a description about how the algorithm works.
6. Write a function to print out the following pattern of stars. Allow the width and height of the rectangle to be indicated by the arguments `width` and `height` respectively.

```
**********
*        *
*        *
*        *
*        *
*        *
*        *
**********
```

7. Write a program to print out the following pattern of stars. Allow the height of the triangle to be indicated by an argument named `height`.

```
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```

8. For the following code, predict what will be printed out and then test your prediction.

```
val x = 42
println(x)
let :
   println(x)
   val x = "Hi"
   println(x)
   let :
      val x = 42
      println(x)
   let :
      val x = "There"
      println(x)
println(x)
let :
   val x = x + 1
   println(x)
println(x)
```

9. For the following code, write it out completely on a single line, grouping expressions with parentheses where necessary. Run both versions and ensure that they behave identically.

```
val x = (42 10)
for i in 0 to 10 do :
   println(x)
      println("B")
println("C")
for i in 0 to 10 do : println("D")
   println("E")
```

Chapter 3

The Less Basic

3.1 More about Structs

Mutable Fields

In the last chapter, you were taught how to define structs and create objects. But the structs you created were *immutable*. You couldn't change the objects at all after you created them.

Here was our original definition of `Dog`.

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

We can create a dog by calling the `Dog` function with a provided name and breed. But once created, you cannot change a dog's name or breed. Here's how to define `Dog` with a *setter function* for changing its name.

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

Now we can use the `set-name` function to change a dog's name.

```
val d = Dog("Shadow", "Golden Retriever")
println("I used to be called %_." % [name(d)])
set-name(d, "Sir Shadow the Wise")
println("But now I am called %_." % [name(d)])
```

prints out

```
I used to be called Shadow.
But now I am called Sir Shadow the Wise.
```

With the above definition of `Dog`, we can change a dog's name but not its breed. If we want to be able to change the breed as well then we need to similarly give it a setter function.

The convention is to call the setter function the same name as the field it's setting but with a `set-` prefix. Follow this convention unless you have a good reason not to.

Providing Custom Printing Behaviour

We are used to using the `print` and `println` functions for printing things. Almost all of Stanza's core types can be printed using `print`. But `print` doesn't yet know how to print `Dog` objects. So the following

```
val d = Dog("Shadow", "Golden Retriever")
println("They call me %_." % [d])
```

prints out the fairly useless message

```
They call me [Unprintable Object].
```

Here is how to provide custom printing behaviour for `Dog` objects.

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

Now the same code

```
val d = Dog("Shadow", "Golden Retriever")
println("They call me %_." % [d])
```

prints out

```
They call me Shadow the Golden Retriever.
```

The `defmethod` keyword *extends* a defined *multi* with a new *method*. We'll learn what that all means later. This gives you small taste of Stanza's multimethod functionality and is the basis for Stanza's class-less object system.

In the body of the `print` method

```
print(o, "%_ the %_." % [name(d), breed(d)])
```

be especially mindful of the `o` argument to `print`. This argument says to print the message to the `o` *output stream*.

3.2 The Match Expression

Now that you are familiar with a number of different types and know how to create objects of each one, you'll have to learn how to differentiate between them. Here's how to write a

function that does different things depending on whether its argument is an integer or a string.

```
defn what-am-i (x) :
   match(x) :
      (i:Int) : println("I am %_. I am an integer." % [i])
      (s:String) : println("I am %_. I am a string." % [s])
```

If we call it with an integer

```
what-am-i(42)
```

then it prints out

```
I am 42. I am an integer.
```

But if we call it with a string

```
what-am-i("Timon")
```

then it prints out

```
I am Timon. I am a string.
```

If we call it with neither an integer or a string

```
what-am-i(false)
```

then the program crashes.

```
FATAL ERROR: No matching branch.
   at stanzaproject/lessbasic.stanza:5.9
   at stanzaproject/lessbasic.stanza:9.0
```

General Form

Here's the general form of a match expression.

```
match(argument expressions ...) :
   (argname:ArgType ...) : body
   ...
```

A match expression

1. computes the result of evaluating all the argument expressions,
2. then tests to see whether the results match the argument types indicated in the first branch.
3. If the types match, then the branch argument names are *bound* to the results, and the branch body is evaluated. The result of the branch is the result of the entire match expression.

4. If the types do not match, then the subsequent branch is tried. This continues either until a branch finally matches, or no branch matches and the program crashes.

Name Shadowing

The match expression branches each start a new scope, and the branch arguments are only visible from within that scope. To avoid confusing you we gave new names (`i` and `s`) to the branch arguments in our example

```
defn what-am-i (x) :
   match(x) :
      (i:Int) : println("I am %_. I am an integer." % [i])
      (s:String) : println("I am %_. I am a string." % [s])
```

but you can really use any name for the branch arguments. In fact, it is common to use the same name as the value that you are matching on.

```
defn what-am-i (x) :
   match(x) :
      (x:Int) : println("I am %_. I am an integer." % [x])
      (x:String) : println("I am %_. I am a string." % [x])
```

Matching Multiple Arguments

The match expression supports matching against *multiple* arguments. Here's a function that returns different things depending on the types of both of its arguments.

```
defn differentiate (x, y) :
   match(x, y) :
      (x:Int, y:Int) : 0
      (x:Int, y:String) : 1
      (x:String, y:Int) : 2
      (x:String, y:String) : 3
```

If we call it with different combinations of integers and strings

```
println(differentiate(42, 42))
println(differentiate(42, "Timon"))
println(differentiate("Pumbaa", 42))
println(differentiate("Timon", "Pumbaa"))
```

it returns different results for each. The above prints out

```
0
1
2
3
```

Example: Cats and Dogs

Here's a definition of two structs, `Cat` and `Dog`.

```
defstruct Dog : (name:String)
defstruct Cat : (name:String)
```

Here's the definition of a `say-hi` function that prints different messages depending on whether `x` is a `Cat` or `Dog`.

```
defn say-hi (x) :
   match(x) :
      (x:Dog) : println("Woof says %_ the dog." % [name(x)])
      (x:Cat) : println("Meow says %_ the cat." % [name(x)])
```

Let's call it a few times.

```
say-hi(Dog("Shadow"))
say-hi(Dog("Chance"))
say-hi(Cat("Sassy"))
```

prints out

```
Woof says Shadow the dog.
Woof says Chance the dog.
Meow says Sassy the cat.
```

Introducing Union Types

One problem with the `say-hi` function is that it allows us to pass obviously incorrect arguments to it, which crashes the program.

```
say-hi(42)
```

results in

```
FATAL ERROR: No matching branch.
   at stanzaproject/lessbasic.stanza:5.9
   at stanzaproject/lessbasic.stanza:9.0
```

This is because we didn't give `x` a type annotation

```
defn say-hi (x)
```

which we've said is equivalent to declaring it with the `?` type.

```
defn say-hi (x:?)
```

The `?` type, by definition, allows us to pass anything to it, so Stanza is doing what it should, even though it's not what we want.

We would like to give `x` a type annotation that prevents us from passing `42` to `say-hi`, but what should it be? It's neither `Dog` nor `Cat` because `say-hi` has to accept them both. The solution is to annotate `say-hi` to take *either* a `Dog` *or* a `Cat`.

```
defn say-hi (x:Dog|Cat) :
   match(x) :
      (x:Dog) : println("Woof says %_ the dog." % [name(x)])
      (x:Cat) : println("Meow says %_ the cat." % [name(x)])
```

You can verify that calling `say-hi` with dogs and cats continue to work, but more importantly, that calling `say-hi` with `42` *doesn't* work. Attempting to compile

```
say-hi(42)
```

gives the error

```
Cannot call function say-hi of type Cat|Dog -> False with arguments of type (Int).
```

`Cat`Dog| is an example of a *union type*. Union types allow us to specify the concept of "either this type or that type".

Branches with Unspecified Types

If you leave off the type annotation for an argument in a match expression branch, then Stanza will automatically infer it to have the same type as the match argument expression. The following

```
defn f (x:Int|String) :
   match(x) :
      (x:Int) : body
      (x) : body2
```

is equivalent to

```
defn f (x:Int|String) :
   match(x) :
      (x:Int) : body
      (x:Int|String) : body2
```

This is often used to provide a default branch to run when none of the preceeding branches match.

Revisiting the If Expression

Now that you've been introduced to the match expression, it's time to unveil the inner workings of the if expression. It turns out that the if expression is just a slightly decorated match expression.

```
if x < 4 :
   println("Do this")
else :
   println("Do that")
```

is completely equivalent to

```
match(x < 4) :
   (p:True) : println("Do this")
   (p:False) : println("Do that")
```

The if expression is an example of a simple *macro*. Macros are very powerful tools for simplifying the syntax of commonly used patterns. Stanza includes many constructs that are simply decorated versions of other constructs, each implemented as a macro. The defstruct statement is another example. Later, we'll learn how to write our own macros to provide custom syntax for common patterns.

3.3 The Is Expression

Often you simply want to determine whether an object is of a certain type. Here is a long-winded method for checking whether `x` is a `Dog` object or not.

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

Because this operation is so common, Stanza provides a shorthand for it. The above can be rewritten equivalently as

```
val dog? = x is Dog
```

Here is the general form.

```
exp is Type
```

It first evaluates `exp` and then returns `true` if the result is of type `Type`. Otherwise it returns `false`. The is expression is another example of a convenience syntax implemented using a macro. As you've noticed by now, Stanza's core library makes heavy use of macros.

The negative form of the is expression is the is-not expression. The following determines whether `x` is *not* a type of `Dog`.

```
val not-dog? = x is-not Dog
```

3.4 Casts

Stanza's type system is designed primarily to be *predictable*, not necessarily smart. This means that, as the programmer, you will often be able to infer a more specific type for an object than Stanza. Here is an example.

```
defn meow (x:Cat) :
   println("Meow!!!")

defn f (x:Cat|Dog) :
   val catness = if x is Cat : 1 else : -1
   if catness > 0 :
      meow(x)
```

Attempting to compile the above gives the error

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

Stanza believes that `x` is a `Dog`Cat|, but from our reasoning, the only way that `meow` can be called is if `catness` is positive. And `catness` is only positive if `x` is a `Cat`. Therefore `x` must be a `Cat` in the call to `meow` and the code should be fine.

To force Stanza to accept `x` as a `Cat`, we can explicitly *cast* `x`.

```
defn f (x:Cat|Dog) :
   val catness = if x is Cat : 1 else : -1
   if catness > 0 :
      meow(x as Cat)
```

The cast tells Stanza to trust your assertion that `x` is indeed a `Cat`. If, for some reason, your reasoning is faulty and `x` turns out not to be a `Cat`, then the incorrect cast will cause the program to crash at that point.

3.5 Deep Casts

Stanza's cast mechanism is much more flexible than many other languages, and, in particular, supports the notion of a *deep cast*. Here is a function that takes an array of integers or strings, and replaces each string in the array with its length.

```
defn compute-lengths (xs:Array<Int|String>) :
   for i in 0 to length(xs) do :
      match(xs[i]) :
         (x:String) : xs[i] = length(x)
         (x:Int) : false
```

And here is a function that computes the sum of an array of integers.

```
defn sum-integers (xs:Array<Int>) :
   var sum = 0
   for i in 0 to length(xs) do :
      sum = sum + xs[i]
   sum
```

Now, given an array containing both integers and strings, we want to first replace each string with its length, and then compute the sum of the integers in the array.

```
val xs = Array<Int|String>(4)
xs[0] = 42
xs[1] = 7
xs[2] = "Timon"
xs[3] = "Pumbaa"

compute-lengths(xs)
sum-integers(xs)
```

Attempting to compile the above gives us the error

```
Cannot call function sum-integers of type Array<Int> -> Int with arguments
of type (Array<String|Int>).
```

Stanza is complaining that `sum-integers` requires an array of integers, so `xs` is an illegal argument as it might contain strings.

But *we* know that `xs` will not contain any strings at that point because `compute-lengths` replaced all of them with their lengths. So we can use a cast to force Stanza to trust this assertion.

```
sum-integers(xs as Array<Int>)
```

With the above correction, the program now compiles and runs correctly.

Types as Contracts

The above was an example of a *deep* cast, because it wasn't a direct assertion about the type of `xs`, but about the types of the objects it *contains*. You might be wondering, then, what exactly does that cast do? Does it iterate through the array and check every element to see if it is an `Int`? You'll be relieved to hear that it does not. That would be hopelessly inefficient, and also impossible in general.

To answer the question, let's investigate what the cast does in the case that we're wrong. Change the definition of `compute-lengths` to this.

```
defn compute-lengths (xs:Array<Int|String>) :
   for i in 0 to length(xs) - 1 do :
      match(xs[i]) :
         (x:String) : xs[i] = length(x)
```

```
        (x:Int) : false
```

It now forgets to check the last element. So even after the call to `compute-lengths`, `xs` still contains one last string (`"Pumbaa"`) at the end, and thus our cast is incorrect.

Compile and run the program. It should crash with this error.

```
FATAL ERROR: Cannot cast value to type.
   at core/core.stanza:3062.16
   at stanzaprojects/lessbasic.stanza:13.18
   at core/core.stanza:2292.9
   at core/core.stanza:4042.16
   at stanzaprojects/lessbasic.stanza:12.28
   at stanzaprojects/lessbasic.stanza:23.0
```

The file position `stanzaprojects/lessbasic.stanza:13.18` tells us that the error occurred in the reference to `xs[i]` in `sum-integers`. Stanza is saying that it was expecting `xs[i]` to be an `Int` because you promised that `xs` is an `Array<Int>`. But `xs[i]` is *not* an `Int`, and so your program is wrong.

In general, a value's type in Stanza does not directly say what it *is*. Instead, a value's type is a *contract* on how it should behave. Part of the contract for an `Array<Int>` is that it should only contain `Int` objects. The above program crashed as soon as Stanza determined that `xs` does not satisfy its contract.

3.6 Operations on Strings

There are many useful operations on `String` objects available in the core library. We'll show a few of them here.

Length

Here's how to obtain the length of a string.

```
val s = "Hello World"
length(s)
```

Retrieve Character

Here's how to retrieve a given character in a string.

```
val s = "Hello World"
s[4]
```

The first character has index 0, and the last character is indexed one less than the length of the string. There is no function for setting the character in a string because strings are *immutable* in Stanza.

Convert to String

Here's how to convert any object into a string.

```
to-string(42)
```

Append

Here's how to form a longer string from appending two strings together.

```
val s1 = "Hello "
val s2 = "World"
append(s1, s2)
```

Substring

Here's how to retrieve a range of characters within a string.

```
val str = "Hello World"
println(str[4 to 9])
```

prints out

```
o Wor
```

It's all the characters between index 4 (inclusive) and index 9 (exclusive) in the string.

If we wanted to include the ending index, then we can use the `through` keyword, just as we've learned from the previous chapter.

```
println(str[4 through 9])
```

prints out

```
o Worl
```

If we wanted to extract all characters from index 4 until the end of the string, we can use `false` as the ending index.

```
println(str[4 to false])
```

prints out

```
o World
```

Check out the reference documentation for a listing of operations supported by `String` objects.

3.7 Operations on Tuples

Tuples support a few additional operations for querying its properties.

Length

Here is how to retrieve the length of a tuple.

```
val t = [4, 42, "Hello"]
length(t)
```

Retrieve an Element

Here is how to retrieve an element in a tuple at a *dynamically* calculated index.

```
val t = [4, 42, "Hello"]
val i = 1 + 1
println(t[i])
```

prints out

```
Hello
```

Note that, in general, a dynamically calculated index is not known until the program actually runs. This means that Stanza does not try to determine a precise type for the result of `t[i]`. The resulting type of `t[i]` is the union of all the element types in the tuple.

Attempting to compile this

```
val t = [4, 42, "Hello"]
val x:Int = t[0 + 1]
```

results in the error

```
Cannot assign expression of type Int|String to value x with declared type Int.
```

The tuple `t` has type `[Int, Int, String]`, and so an arbitrary element at an unknown index has type `Int`String|.

To overcome this, you may explicitly cast the result to an `Int` yourself.

```
val t = [4, 42, "Hello"]
val x:Int = t[0 + 1] as Int
```

Check out the reference documentation for a listing of operations supported by tuples.

Tuples of Unknown Length

The type `[Int]` is a tuple containing one integer, and the type `[Int, Int]` is a tuple containing two integers, et cetera. But what if we want to write a function that takes a tuple of *any* number of integers?

Here is a function that prints out every number in a tuple of integers.

```
defn print-tuple (t:Tuple<Int>) :
   for i in 0 to length(t) do :
      println(t[i])
```

The following

```
print-tuple([1, 2, 3])
```

prints out

```
1
2
3
```

But the following

```
print-tuple([1, "Timon"])
```

fails to compile with the error

```
Cannot call function print-tuple of type Tuple<Int> -> False with arguments
of type ([Int, String]).
```

In general, the type `Tuple<Type>` represents a tuple of unknown length where each element type is of type `Type`.

3.8 Packages

Thus far, all of your code has been contained in a single package. When your projects get larger, you'll start to feel the need to split up the entire program into smaller isolated components. In Stanza, you would do this by partitioning your program into multiple *packages*.

Create a separate file called `animals.stanza` containing

```
defpackage animals :
   import core

defstruct Dog :
   name: String
defstruct Cat :
```

```
   name: String

defn sound (x:Dog|Cat) :
   match(x) :
      (x:Dog) : "woof"
      (x:Cat) : "meow"
```

The `animals` package contains all of our code for handling dogs and cats. It contains the struct definitions for `Dog` and `Cat`, as well as the `sound` function that returns the sound made by each animal.

Now create a file called `mainprogram.stanza` containing

```
defpackage animal-main :
   import core

defn main () :
   val d = Dog("Shadow")
   val c = Cat("Sassy")
   println("My dog %_ goes %_!" % [name(d), sound(d)])
   println("My cat %_ goes %_!" % [name(c), sound(c)])

main()
```

The `animal-main` package contains the main code of the program and it will use the `animals` package as a library.

Importing Packages

Now compile both of your source files by typing in the terminal

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

Oops! Something's wrong! Stanza reports these errors.

```
mainprogram.stanza:5.11: Could not resolve Dog.
mainprogram.stanza:6.11: Could not resolve Cat.
mainprogram.stanza:7.44: Could not resolve sound.
mainprogram.stanza:8.44: Could not resolve sound.
```

The problem is that our `animal-main` package never *imported* the `animals` package. Packages must be imported before they can be used. So change

```
defpackage animal-main :
   import core
```

to

```
defpackage animal-main :
```

```
   import core
   import animals
```

and try compiling again. Stanza *still* reports the same errors.

```
mainprogram.stanza:5.11: Could not resolve Dog.
mainprogram.stanza:6.11: Could not resolve Cat.
mainprogram.stanza:7.44: Could not resolve sound.
mainprogram.stanza:8.44: Could not resolve sound.
```

Public Visibility

What's going on? The problem now is that our `animals` package did not make any of its definitions *public*. By default, definitions are not visible from outside the package it is declared in. To make a definition visible, you must prefix the definition with the `public` keyword.

Let's declare our `Dog` and `Cat` structs, and the `sound` function to be publicly visible.

```
defpackage animals :
   import core

public defstruct Dog :
   name: String
public defstruct Cat :
   name: String

public defn sound (x:Dog|Cat) :
   match(x) :
      (x:Dog) : "woof"
      (x:Cat) : "meow"
```

Now the program compiles successfully and prints out

```
My dog Shadow goes woof!
My cat Sassy goes meow!
```

Private Visibility

By default, all definitions are *private* to the package that they are defined in. There is *no* way to refer to a private definition from outside the package. This is a very powerful guarantee as it also means that there is no way for any outside code to depend upon the existence of a private definition.

For example, suppose we rely on a helper function called `dog?` to help us define the `sound` function.

```
defpackage animals :
   import core

public defstruct Dog :
   name: String
public defstruct Cat :
   name: String

defn dog? (x:Dog|Cat) :
   match(x) :
      (x:Dog) : true
      (x:Cat) : false

public defn sound (x:Dog|Cat) :
   if dog?(x) : "woof"
   else : "meow"
```

`dog?` is private to the `animals` package, so at any time in the future, if we wanted to rename `dog?` or remove it, we can safely do so without affecting other code.

3.9 Function Overloading

By this point, we've learned about arrays, tuples, strings, and how to retrieve the length of each of them.

```
val a = Array<Int>(4)
val b = "Timon and Pumbaa"
val c = [1, 2, 3, 4]
length(a) ;Retrieve length of a array
length(b) ;Retrieve length of a string
length(c) ;Retrieve length of a tuple
```

You simply call the `length` function. Here is what is happening behind the scenes. The `core` package actually contains *many* functions called `length`, but they differ in the type of the argument that they accept.

```
defn length (x:Array) -> Int
defn length (x:String) -> Int
defn length (x:Tuple) -> Int
```

When you call `length(a)`, Stanza automatically figures out which `length` function you are trying to call based on the type of its argument. `a` is an array, and so you're obviously trying to call the `length` function that accepts an `Array`. No other `length` function would be legal to call! Similarly, `b` is a string, so the call to `length(b)` is obviously a call to the `length` function that accepts a `String`. This is a feature called *function overloading* and is a key part of Stanza's object system.

Functions can be overloaded based on the number of arguments that they take, and the types of each argument. Let's write our own overloaded function.

```
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!")
```

Now let's try calling each of them. The following

```
val d = Dog()
val t = Tree()
val c = Captain()
bark(d)
println(bark(t))
bark(c)
```

prints out

```
Woof!
Furrowed Cork
A teeeen-hut!
```

Notice that the `bark` function for `Tree` returns a `String`, while the `bark` functions for `Dog` and `Captain` return `False`. There is no requirement for any of the `bark` functions to be related or aware of each other. They can even be declared in separate packages!

3.10 Operator Mapping

In the previous chapter, you were introduced to the basic arithmetic operators. Here we'll show you a bit about how they work underneath. The following

```
val a = 13
val b = 24
a + b
a - b
a * b
a / b
a % b
(- a)
```

can be rewritten equivalently as

```
val a = 13
val b = 24
plus(a, b)
minus(a, b)
times(a, b)
divide(a, b)
modulo(a, b)
negate(a)
```

Thus you can see here that all operators in Stanza are simply syntactic shorthands for specific function calls. Here is a listing of what each operator expands to.

```
a + b     expands to   plus(a, b)
a - b     expands to   minus(a, b)
a * b     expands to   times(a, b)
a / b     expands to   divide(a, b)
a % b     expands to   modulo(a, b)
(- x)     expands to   negate(x)

a << b    expands to   shift-left(a, b)
a >> b    expands to   shift-right(a, b)
a >>> b   expands to   arithmetic-shift-right(a, b)
a & b     expands to   bit-and(a, b)
a | b     expands to   bit-or(a, b)
a ^ b     expands to   bit-xor(a, b)
(~ x)     expands to   bit-not(x)

a == b    expands to   equal?(a, b)
a != b    expands to   not-equal?(a, b)
a < b     expands to   less?(a, b)
a <= b    expands to   less-eq?(a, b)
a > b     expands to   greater?(a, b)
a >= b    expands to   greater-eq?(a, b)
not x     expands to   complement(x)
```

Operator Overloading

The benefit to mapping each operator to a function call is that you can very easily reuse these operators for your own objects. Here is an example struct definition for modeling points on the cartesian plane.

```
defstruct Point :
   x: Double
   y: Double
```

Next let's define a function called `plus` that can add together two `Point` objects.

```
defn plus (a:Point, b:Point) :
   Point(x(a) + x(b), y(a) + y(b))
```

Let's try out our function.

```
defn main () :
   val a = plus(Point(1.0,3.0), Point(4.0,5.0))
   val b = plus(a, Point(7.0,1.0))
   println("b is (%_, %_)" % [x(b), y(b)])

main()
```

The above prints out

```
b is (12.000000000000000, 9.000000000000000)
```

But, as mentioned, the `+` operator is a shorthand for calling the `plus` function. So our `main` function can be written more naturally as

```
defn main () :
   val a = Point(1.0,3.0) + Point(4.0,5.0)
   val b = a + Point(7.0,1.0)
   println("b is (%_, %_)" % [x(b), y(b)])
```

Get and Set

Two other operators that we have been using without being aware of it are the get and set operators. The following code

```
val a = Array<Int>(4)
a[0] = 42
a[0]
```

is equivalent to

```
val a = Array<Int>(4)
set(a, 0, 42)
get(a, 0)
```

Thus the `a[i]` form expands to calls to the `get` function.

```
a[i]         expands to   get(a, i)
a[i, j]      expands to   get(a, i, j)
a[i, j, k]   expands to   get(a, i, j, k)
etc ...
```

And the `a[i] = v` form expands to calls to the `set` function.

```
a[i] = v          expands to   set(a, i, v)
a[i, j] = v       expands to   set(a, i, j, v)
a[i, j, k] = v    expands to   set(a, i, j, k, v)
etc ...
```

3.11 Vectors

So far we've only called the library functions in the `core` package. The `collections` package contains commonly used datastructures useful for daily programming.

Here is a program that imports the `collections` package and creates and prints a `Vector` object.

```
defpackage mypackage :
   import core
   import collections

defn main () :
   val v = Vector<Int>()
   add(v, 1)
   add(v, 2)
   add(v, 3)
   println(v)

main()
```

It prints out

```
[1 2 3]
```

A `Vector` object is similar to an array and represents a mutable collection of items where each item is associated with an integer index. However, whereas arrays are of fixed length, a vector can grow and shrink to accomodate more or less items.

The type of the `v` vector in the example above is

```
Vector<Int>
```

indicating that it is a vector for storing integers.

Here is how to add additional elements to the end of a vector.

```
add(v, 42)
```

Here is how to retrieve and remove the element at the end of the vector.

```
pop(v)
```

Identical to the case of arrays, here is how to retrieve the length of a vector, retrieve a value at a particular index, and assign a value to a particular index.

```
length(v) ;Retrieve a vector's length
v[0] = 42 ;Assign a value to index 0
v[0] ;Retrieve the value at index 0
```

3.12 HashTables

Hash tables are another commonly used datastructure in the `collections` package. A table associates a value object with a particular key object. It can be imagined as a two-column table (hence the name) where the left column is named *keys* and the right column is named *values*. Each entry in the table is recorded as a new row. The key object is recorded in the keys column, and its corresponding value object is recorded in the values column.

Here is how to create a HashTable where strings are used as keys, and integers are used as values.

```
val num-pets = HashTable<String,Int>()
num-pets["Luca"] = 2
num-pets["Patrick"] = 1
num-pets["Emmy"] = 3
println(num-pets)
```

The above prints out

```
["Patrick" => 1 "Luca" => 2 "Emmy" => 3]
```

Creation

The function

```
HashTable<String,Int>()
```

creates a new hash table that associates integer values with string keys. The type of the table created by the above function is

```
HashTable<String,Int>
```

which indicates that it is a hash table whose keys have type `String` and whose values have type `Int`.

Set

The calls to `set`

```
num-pets["Luca"] = 2
num-pets["Patrick"] = 1
```

```
num-pets["Emmy"] = 3
```

associates the value 2 with the key "Luca" in the table, the value 1 with "Patrick", and the value 3 with "Emmy".

Get

Here's how to retrieve the value associated with a key.

```
println("Emmy has %_ pets." % [num-pets["Emmy"]])
```

which prints out

```
Emmy has 3 pets.
```

Does a Key Exist?

Attempting to retrieve the value in a table corresponding to a key that doesn't exist is a fatal error. Use the `key?` function to check whether a key exists in the table.

```
if key?(num-pets, "George") :
   println("George has %_ pets." % [num-pets["George"]])
else :
   println("I don't know how many pets George has.")
```

Default Values

A hash table can also be created with a *default* value. If a hash table has a default value, then this default value is returned when retrieving the corresponding value for a key that does not exist in the table. Change the definition of num-pets to

```
val num-pets = HashTable<String,Int>(0)
```

Now when we retrieve the number of pets owned by George,

```
println("George has %_ pets." % [num-pets["George"]])
```

it prints out

```
George has 0 pets.
```

3.13 KeyValue Pairs

A `KeyValue` object represents an association between a key object and a value object. It can be created using the `KeyValue` function.

```
val kv = KeyValue(4, "Hello")
```

creates a `KeyValue` object that represents the mapping from the key `4` to the value `"Hello"`. This is done very commonly, so Stanza also provides a convenient operator. The above can be written equivalently as

```
val kv = 4 => "Hello"
```

The type of the `kv` object created above is

```
KeyValue<Int,String>
```

which indicates that it represents an association between a key of type `Int` and a value of type `String`.

The key and the value objects in a `KeyValue` object can be retrieved using the `key` and `value` functions respectively.

```
key(kv) ;Retrieve the key
value(kv) ;Retrieve the value
```

3.14 For Loops over Sequences

Thus far you've only been shown how to use the for construct for simple counting loops. Here you'll see how the for construct generalizes to all types of collections.

The for loop can be used to iterate directly through the items of an array like so.

```
val xs = Array<Int>(4)
xs[0] = 2
xs[1] = 42
xs[2] = 7
xs[3] = 1

for x in xs do :
   println(x)
```

which prints out

```
2
42
7
1
```

General Form

Here is the general form.

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

For each item in the *collection* `xs`, the for loop executes `body` once with `x` bound to the next item in the collection. In our example, `xs` contains the numbers `2`, `42`, `7`, and `1`, and thus `body` is executed once each with `x` bound to `2`, `42`, `7`, and finally `1`.

Examples of Collections

We will more precisely specify what constitutes a *collection* later. For now, just accept that arrays, vectors, and tuples are collections, and strings are collections of characters. For example,

```
for c in "Timon" do :
   print("Next char is ")
   println(c)
```

prints out

```
Next char is T
Next char is i
Next char is m
Next char is o
Next char is n
```

And similarly,

```
for x in [1, 3, "Timon"] do :
   print("Next item is ")
   println(x)
```

prints out

```
Next item is 1
Next item is 3
Next item is Timon
```

In fact, `Range` objects are collections of integers, so the counting loops we saw before are actually just a special case of iterating through the items in a `Range`.

```
val r = 0 to 4
for x in r do :
   print("Next number is ")
   println(x)
```

prints out

```
Next number is 0
Next number is 1
Next number is 2
```

```
Next number is 3
```

Tables are also collections, but they are collections of `KeyValue` objects, each representing one of the entries in the table. The following

```
val num-pets = HashTable<String,Int>()
num-pets["Luca"] = 2
num-pets["Patrick"] = 1
num-pets["Emmy"] = 3

for entry in num-pets do :
   println("%_ has %_ pets." % [key(entry), value(entry)])
```

prints out

```
Patrick has 1 pets.
Luca has 2 pets.
Emmy has 3 pets.
```

As you can see, Stanza's for construct is extremely powerful. In truth, the form shown here is *still* not the most general form of the for construct. We'll learn about that after we've covered first class functions.

3.15 Extended Example: Complex Number Package

In this extended example, we will implement a package for creating and performing arithmetic with complex numbers.

The Complex Package

Create a file called `complex.stanza` with the following content.

```
defpackage complex :
   import core

public defstruct Cplx :
   real: Double
   imag: Double
```

This struct will be our representation for complex numbers. It is stored in cartesian form and has real and imaginary components.

Printing Complex Numbers

To be able to print `Cplx` objects, we provide a custom print method.

```
defmethod print (o:OutputStream, x:Cplx) :
   if imag(x) >= 0.0 :
      print(o, "%_ + %_i" % [real(x), imag(x)])
   else :
      print(o, "%_ - %_i" % [real(x), (- imag(x))])
```

Main Driver

To test our program thus far, create a file called `complexmain.stanza` with the following content.

```
defpackage complex/main :
   import core
   import complex

defn main () :
   val a = Cplx(1.0, 5.0)
   val b = Cplx(3.0, -4.0)
   println(a)
   println(b)

main()
```

Compile and run the program by typing the following in the terminal.

```
stanza complex.stanza complexmain.stanza -o cplx
./cplx
```

It should print out

```
1.000000000000000 + 5.000000000000000i
3.000000000000000 - 4.000000000000000i
```

Great! So now we can create and print out complex numbers. If you're an electrical engineer, you may substitute `i` for `j` in the `print` method.

Arithmetic Operations

The next step is to implement the standard arithmetic operations for complex numbers. Pull out your old algebra textbooks and look up the formulas. Or pick up a pencil and derive them yourself.

```
public defn plus (a:Cplx, b:Cplx) :
   Cplx(real(a) + real(b), imag(a) + imag(b))

public defn minus (a:Cplx, b:Cplx) :
```

```
   Cplx(real(a) - real(b), imag(a) - imag(b))

public defn times (a:Cplx, b:Cplx) :
   val x = real(a)
   val y = imag(a)
   val u = real(b)
   val v = imag(b)
   Cplx(x * u - y * v, x * v + y * u)

public defn divide (a:Cplx, b:Cplx) :
   val x = real(a)
   val y = imag(a)
   val u = real(b)
   val v = imag(b)
   val den = u * u + v * v
   Cplx((x * u + y * v) / den, (y * u - x * v) / den)
```

Let's test out our operators.

```
defn main () :
   val a = Cplx(1.0, 5.0)
   val b = Cplx(3.0, -4.0)
   println("(%_) + (%_) = %_" % [a, b, a + b])
   println("(%_) - (%_) = %_" % [a, b, a - b])
   println("(%_) * (%_) = %_" % [a, b, a * b])
   println("(%_) / (%_) = %_" % [a, b, a / b])

main()
```

The program prints out

```
(1.000000000000000 + 5.000000000000000i) + (3.000000000000000 - 4.000000000000000i)
    = 4.000000000000000 + 1.000000000000000i
(1.000000000000000 + 5.000000000000000i) - (3.000000000000000 - 4.000000000000000i)
    = -2.000000000000000 + 9.000000000000000i
(1.000000000000000 + 5.000000000000000i) * (3.000000000000000 - 4.000000000000000i)
    = 23.000000000000000 + 11.000000000000000i
(1.000000000000000 + 5.000000000000000i) / (3.000000000000000 - 4.000000000000000i)
    = -0.680000000000000 + 0.760000000000000i
```

which looks right to me!

Root Finding

Armed with our new complex number package, let's now put it to good use and solve an equation. We will use the Newton-Raphson method to solve the following equation.

```
x ^ 3 - 1 = 0
```

Here is our numerical solver which takes an initial guess, `x0`, and the number of iterations, `num-iter`, and performs `num-iter` number of Newton-Raphson iterations to find the root of the equation.

```
defn newton-raphson (x0:Cplx, num-iter:Int) :
   var xn = x0
   for i in 0 to num-iter do :
      xn = xn - (xn * xn * xn - Cplx(1.0,0.0)) / (Cplx(3.0,0.0) * xn * xn)
   xn
```

Let's test it!

```
defn main () :
   println(newton-raphson(Cplx(1.0,1.0), 100))
```

The program prints out

```
1.000000000000000 + 0.000000000000000i
```

which is indeed one of the solutions to the equation! Fantastic!

Find all the Roots!

But according to the Fundamental Theorem of Algebra, the equation should have two more solutions. Different initial guesses will converge to different solutions so let's try a whole bunch of different guesses and try to find them all.

Here is a function that takes in a tuple of initial guesses and tries them all.

```
defn guess (xs:Tuple<Cplx>) :
   for x in xs do :
      val r = newton-raphson(x, 100)
      println("Initial guess %_ gave us solution %_." % [x, r])
```

And let's call it with a bunch of random guesses.

```
defn main () :
   guess([Cplx(1.0,1.0), Cplx(2.0,2.0), Cplx(-1.0,3.0), Cplx(-1.0,-1.0)])
```

The program prints out

```
Initial guess 1.000000000000000 + 1.000000000000000i gave
   us solution 1.000000000000000 + 0.000000000000000i.
Initial guess 2.000000000000000 + 2.000000000000000i gave
   us solution 1.000000000000000 + 0.000000000000000i.
Initial guess -1.000000000000000 + 3.000000000000000i gave
   us solution -0.500000000000000 + 0.866025403784439i.
Initial guess -1.000000000000000 - 1.000000000000000i gave
```

```
us solution -0.500000000000000 - 0.866025403784439i.
```

Thus the three solutions to the equation are `1`, `-0.5 + 0.866i`, and `-0.5 - 0.866i`. Problem solved!

Chapter 4

Architecting Programs

Stanza's object system differs significantly from most other programming languages. Most other languages (e.g. Java, C#, Python, Ruby, Objective-C, C++, Swift, Scala, OCaml, etc.) employ a *class* based object system. In a class based object system, each *thing* in the program is represented using a class. For each *ability* that a thing has, the user adds another method to its class. Classes have all the power in a class-based object system. Methods live inside classes.

In contrast, Stanza employs a *class-less* object system. In Stanza, each *thing* is represented as a type. There is a minimal set of fundamental operations that defines the behaviour of a type. After that, everything that can be *done* with each thing is implemented as a simple function. Both types and functions have equal standing. Types do not live inside functions, nor do functions live inside types. The careful balance between these constructs is what gives Stanza its flexibility and architectural power.

4.1 A Shape Library

In `shapes.stanza`, let's create a package for creating and manipulating two-dimensional shapes.

```
defpackage shapes :
   import core
   import math

public defstruct Point :
   x: Double
   y: Double

public defstruct Circle :
   x: Double
   y: Double
```

```
   radius: Double

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

defmethod print (o:OutputStream, p:Point) :
   print(o, "Point(%_, %_)" % [x(p), y(p)])

defmethod print (o:OutputStream, c:Circle) :
   print(o, "Circle(%_, %_, radius = %_)" % [x(c), y(c), radius(c)])
```

The `shapes` package contains struct definitions for points and circles, methods for printing them, as well as an `area` function for computing the areas of these shapes. It imports the `math` package to access the definition of the mathematical constant `PI`.

In `shapes-main.stanza`, as our main program, let's compute the total area of a bunch of shapes.

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

defn total-area (ss:Vector<Point|Circle>) :
   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("Shapes:")
   println(ss)
   println("Total area = %_" % [total-area(ss)])

main()
```

Compile and run the program by typing the following in the terminal.

```
stanza shapes.stanza shapes-main.stanza -o shapes
./shapes
```

It should print out

```
Shapes:
[Point(1.000000000000000, 1.000000000000000)
 Circle(2.000000000000000, 2.000000000000000, radius = 3.000000000000000)
 Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)]
Total area = 31.415926535897931
```

4.2 Creating a New Shape

Our `shapes` package supports points and circles, but let's now extend it to support rectangles. What do we need to change in order to support another shape?

1. First we need to define a `Rectangle` struct for representing rectangles.
2. Next we need to provide custom printing behavior for rectangles.
3. We need to change `area`'s type signature to now accept a `Point`Circle|Rectangle|.
4. We need to add another branch to the implementation of `area` to support rectangles.
5. We need to change `total-area`'s type signature to now accept a `Vector<Point`Circle|Rectangle>|.
6. We need to change how `ss` is created to allow it to also hold rectangles.

The first two items are straightforward so let's do that immediately.

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

defmethod print (o:OutputStream, r:Rectangle) :
   print(o, "Rectangle(%_, %_, size = %_ x %_)" %
         [x(r), y(r), width(r), height(r)])
```

A rectangle is defined by the coordinates of its bottom-left corner and its width and height.

The other items on the list are not hard to implement at present but it is clear that it is not a sustainable strategy. Here's how it would look. `area` in the `shapes` package is updated to

```
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)
```

`total-area` in the `shapes/main` package is updated to

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

And the creation of `ss` in the `main` function is updated to

```
val ss = Vector<Point|Circle|Rectangle>()
```

It is not a pretty solution. Imagine if we had ten more types shapes to define. `area`'s type signature would quickly become unwieldy as we tack on more and more types to the argument. Every new shape requires manually editing the internals of the `area` function. Currently `area` is the only function defined on shapes, but what if there were a dozen more? Would we have to manually edit the internals of each of them?

By far, the worst aspect of the solution is the need to update the definition of the user's `total-area` function and `ss` vector. The user simply wants `total-area` to accept a vector of shapes. Which shapes? Well, *any* shape! There must be a better way to express that than an explicit union containing the names of every single type of shape currently defined.

4.3 Subtyping

Here is the definition of a new *type* called `Shape`.

```
public deftype Shape
```

A `Shape` is a general representation of a two-dimensional shape. If an object has type `Shape`, then we know that it is definitely a shape, though we may not know which particular shape it is.

Here is how to annotate our `Point` struct as being a *subtype* of `Shape`.

```
public defstruct Point <: Shape :
   x: Double
   y: Double
```

This annotation tells Stanza that all points are shapes. Thus if we write a function that requires `Shape` objects,

```
defn its-a-shape (s:Shape) :
   println("%_ is a shape!" % [s])
```

then we are allowed to pass it `Point` objects. The following

```
its-a-shape(Point(1.0, 2.0))
```

compiles correctly and prints out

```
Point(1.000000000000000, 2.000000000000000) is a shape!
```

Note, however, that though all points are shapes, *not* all shapes are points. Thus if we write a function that requires `Point` objects,

```
defn its-a-point (p:Point) :
   println("%_ is a point!" % [p])
```

and try it to call it with a `Shape` object,

```
var s:Shape
its-a-point(s)
```

Stanza will give the following error.

```
Cannot call function its-a-point of type Point -> False with arguments
of type (Shape).
```

Thus the relationship

```
Point <: Shape
```

says that `Point` is a subtype of `Shape`, meaning that all `Point` objects are also `Shape` objects (but not vice versa).

Code Cleanup

Now that we have a definition for `Shape`, let's indicate this relationship for all of our shape structs.

```
public defstruct Point <: Shape :
   x: Double
   y: Double

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

public defstruct Rectangle <: Shape :
   x: Double
   y: Double
   width: Double
   height: Double
```

Now all of our shape structs are also subtypes of `Shape`. This allows us to clean up many of the type signatures, both in the `shapes` package and in the `shapes/main` package.

The type signature for `area` is simplified.

```
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)
```

The type signature for `total-area` is simplified.

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

And the creation of `ss` is simplified.

```
val ss = Vector<Shape>()
```

Notice that with these simplifications, items 3, 5, and 6 on our checklist for creating new shapes are no longer necessary.

4.4 Multis and Methods

Our `shape` package is architecturally fairly complete at this point. It currently supports points, circles and rectangles, and we can calculate the area of each of them. If we need another shape, e.g. lines, then all we have to do is define a `Line` struct and edit `area` to support `Line` objects.

The Need for Extensibility

There remains one limitation to our shapes library however. Suppose that we are the authors and maintainers in charge of the shapes library, and that there are many users who use the `shapes` package for their daily work. What should a user do if our library does not support a shape that he needs? This is a likely scenario, because it is implausible for us to have fully considered the shape needs of every user. And often, some user's needs are so specific that we don't *want* to support it in the standard shapes library. It will just end up cluttering the library and confusing the rest of the users. For example, it seems inappropriate to support the `Salinon` shape in the standard library.

What we can do however, is to allow users to define their *own* shapes. Then typical users can stay content using the standard shapes in the library, and power users can define their own shapes for their own use.

Users can *almost* do this. They can create their own shape struct, and provide custom printing behaviour for it. Let us portray here a user working on greek architecture, and who has started defining his own extensions to the shape library in the file `greek-shapes.stanza`.

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

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

defmethod print (o:OutputStream, s:Salinon) :
   print(o, "Salinon(%_, %_, outer-radius = %_, inner-radius = %_)" % [
      x(s), y(s), outer-radius(s), inner-radius(s)])
```

The problem though is that the user has no way of extending `area` to support `Salinon` shapes, because that would require editing the code in the `shapes` package, which he does not have access to.

defmulti and defmethod

The solution is to declare `area` not as a function but as a *multi*.

```
public defmulti area (s:Shape) -> Double
```

Note that the definition of a multi does not include a body. It simply says that `area` is a multi that when called with a `Shape` returns a `Double`.

Here is how to *attach* a method to a multi.

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

This definition tells Stanza that when the `area` multi is called on a `Point` then simply return `0.0`.

Here are the methods for `area` for circles and rectangles.

```
defmethod area (c:Circle) :
   PI * radius(c) * radius(c)

defmethod area (r:Rectangle) :
   width(r) * height(r)
```

Notice that similar to functions, if the return type is not given, then it is inferred from the method body.

A multi can have any number of methods, and the methods can be distributed across any number of packages and source files. Thus our greek architect can now define a method for `area` for `Salinon` shapes in his own `greek-shapes` package.

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

...

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

You'll just have to trust me on the area of a salinon.

Let's go back to our main program now and include a couple of salinons in our `ss` vector.

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

...

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, Salinon(0.0, 0.0, 10.0, 5.0))
   add(ss, Salinon(5.0, -1.0, 8.0, 7.0))
   println("Shapes:")
   println(ss)
   println("Total area = %_" % [total-area(ss)])

...
```

Compile and run the program by typing

```
stanza shapes.stanza greek-shapes.stanza shapes-main.stanza -o shapes
./shapes
```

The program should print out

```
Shapes:
[Point(1.000000000000000, 1.000000000000000)
 Circle(2.000000000000000, 2.000000000000000, radius = 3.000000000000000)
 Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)
 Salinon(0.000000000000000, 0.000000000000000,
```

```
         outer-radius = 10.000000000000000, inner-radius = 5.000000000000000)
 Salinon(5.000000000000000, -1.000000000000000,
         outer-radius = 8.000000000000000, inner-radius = 7.000000000000000)]
Total area = 384.845100064749658
```

The following listing contains the complete program.

Program Listing

In `shapes.stanza`

```
defpackage shapes :
   import core
   import math

public deftype Shape

public defstruct Point <: Shape :
   x: Double
   y: Double

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

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

defmethod print (o:OutputStream, p:Point) :
   print(o, "Point(%_, %_)" % [x(p), y(p)])

defmethod print (o:OutputStream, c:Circle) :
   print(o, "Circle(%_, %_, radius = %_)" % [x(c), y(c), radius(c)])

defmethod print (o:OutputStream, r:Rectangle) :
   print(o, "Rectangle(%_, %_, size = %_ x %_)" %
         [x(r), y(r), width(r), height(r)])

public defmulti area (s:Shape) -> Double

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

```
   0.0

defmethod area (c:Circle) :
   PI * radius(c) * radius(c)

defmethod area (r:Rectangle) :
   width(r) * height(r)
```

In greek-shapes.stanza

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

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

defmethod print (o:OutputStream, s:Salinon) :
   print(o, "Salinon(%_, %_, outer-radius = %_, inner-radius = %_)" % [
      x(s), y(s), outer-radius(s), inner-radius(s)])

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

In shapes-main.stanza

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

defn total-area (ss:Vector<Shape>) :
   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, Salinon(0.0, 0.0, 10.0, 5.0))
   add(ss, Salinon(5.0, -1.0, 8.0, 7.0))
   println("Shapes:")
   println(ss)
   println("Total area = %_" % [total-area(ss)])

main()
```

4.5 Default Methods

Our current implementation of `area` does not have a *default* method. This means that if we call `area` with a shape that has no appropriate method, then the program will crash. Let's try this by commenting out the method for computing the areas of salinons and try running our program again.

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

It should print out

```
Shapes:
[Point(1.000000000000000, 1.000000000000000)
 Circle(2.000000000000000, 2.000000000000000, radius = 3.000000000000000)
 Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)
 Salinon(0.000000000000000, 0.000000000000000,
         outer-radius = 10.000000000000000, inner-radius = 5.000000000000000)
 Salinon(5.000000000000000, -1.000000000000000,
         outer-radius = 8.000000000000000, inner-radius = 7.000000000000000)]
FATAL ERROR: No matching branch.
   at shapes.stanza:31.16
   at shapes-main.stanza:10.22
   at core/collections.stanza:182.15
   at core/core.stanza:4042.16
   at shapes-main.stanza:9.15
   at shapes-main.stanza:22.32
   at shapes-main.stanza:24.0
```

Let's instead provide a *default* method that is called when no other method matches. Add the following method to the `shapes` package

```
defmethod area (s:Shape) :
   println("No appropriate area method for %_." % [s])
   println("Returning 0.0.")
   0.0
```

and run the program again. It should now print out

```
Shapes:
[Point(1.000000000000000, 1.000000000000000)
 Circle(2.000000000000000, 2.000000000000000, radius = 3.000000000000000)
 Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)
 Salinon(0.000000000000000, 0.000000000000000,
         outer-radius = 10.000000000000000, inner-radius = 5.000000000000000)
 Salinon(5.000000000000000, -1.000000000000000,
         outer-radius = 8.000000000000000, inner-radius = 7.000000000000000)]
No appropriate area method for
   Salinon(0.000000000000000, 0.000000000000000,
           outer-radius = 10.000000000000000, inner-radius = 5.000000000000000).
Returning 0.0.
No appropriate area method for
   Salinon(5.000000000000000, -1.000000000000000,
           outer-radius = 8.000000000000000, inner-radius = 7.000000000000000).
Returning 0.0.
Total area = 31.415926535897931
```

Default methods are often used to return a default value when no other method is appropriate. Another common use case for default methods is to provide a slow but general implementation of a certain function that works on any type in its domain, and then use methods to provide efficient implementations for specialized types.

4.6 Underneath the Hood

To help you better understand how the multi and method system works, here is what is happening underneath the hood. When a Stanza program is compiled it searches through all the packages and gathers up all the methods defined for each multi. In our `shapes` example, that gives us

```
public defmulti area (s:Shape) -> Double
defmethod area (s:Shape) :
   println("No appropriate area method for %_." % [s])
   println("Returning 0.0.")
   0.0
defmethod area (p:Point) -> Double :
   0.0
defmethod area (c:Circle) :
   PI * radius(c) * radius(c)
defmethod area (r:Rectangle) :
   width(r) * height(r)
defmethod area (s:Salinon) :
   val r = outer-radius(s) + inner-radius(s)
```

```
   PI * r * r / 4.0
```

These methods are then *sorted* from most specific to least specific, and the multi is transformed into a single function with a match expression for selecting which method to call.

```
public defn area (s:Shape) -> Double :
   match(s) :
      (p:Point) :
         0.0
      (c:Circle) :
         PI * radius(c) * radius(c)
      (r:Rectangle) :
         width(r) * height(r)
      (s:Salinon) :
         val r = outer-radius(s) + inner-radius(s)
         PI * r * r / 4.0
      (s:Shape) :
         println("No appropriate area method for %_." % [s])
         println("Returning 0.0.")
         0.0
```

Notice how the default method for `Shape` is positioned as the last branch in the match expression as it is the least specific method.

Thus this engine demonstrates that Stanza's multi and method system can simply be thought of as a way of writing match expressions but with its branches distributed across multiple packages.

4.7 Intersection Types

Multiple Parent Types

Suppose we have a type called `Rollable` with the following multi declared.

```
deftype Rollable
defmulti roll-distance (r:Rollable) -> Double
```

`roll-distance` computes the distance traveled by a `Rollable` object in one revolution.

We now wish to make `Circle` a subtype of `Rollable` and provide it an appropriate method for `roll-distance`, but `Circle` is already declared to be a subtype of `Shape`! The solution is to declare `Circle` as both a subtype of `Shape` *and* `Rollable`.

```
public defstruct Circle <: Shape & Rollable :
   x: Double
   y: Double
```

```
   radius: Double
```

This is an example of using an *intersection type*. Now we can provide a method for `roll-distance` on `Circle` objects.

```
defmethod roll-distance (c:Circle) :
   2.0 * PI * radius(c)
```

A circle rolls exactly the length of its diameter in one revolution.

Let's try it out!

```
defn try-rolling () :
   val c = Circle(0.0, 0.0, 10.0)
   println("The circle %_ rolls %_ units in one revolution." % [c, roll-distance(c)])

try-rolling()
```

Compiling and running the above prints out

```
The circle Circle(0.000000000000000, 0.000000000000000,
                  radius = 10.000000000000000)
rolls 62.831853071795862 units in one revolution.
```

Intersection Types as Arguments

Let's suppose that we're about to make a pizza, and we need to choose the shape of its base. Additionally, we're experimenting with new pizza delivery methods, and would also like the pizza to be able to roll.

Here's the function that makes our pizza.

```
defn make-pizza (base:Shape & Rollable) :
   println("Let's make a pizza!")
   println("The base will have shape %_." % [base])
   println("And it will have area %_." % [area(base)])
   println("Plus when we roll it, it travels %_ units!" % [roll-distance(base)])
```

Because we call `area` on `base` we know that the `base` needs to be a type of `Shape` object. But we also call `roll-distance` on `base`, and so `base` will *also* have to be a type of `Rollable` object. Thus `base` is declared to be both `Shape` and `Rollable`.

Let's try it out!

```
make-pizza(Circle(0.0, 0.0, 10.0))
```

Compiling and running the above prints out

```
Let's make a pizza!
The base will have shape Circle(0.000000000000000, 0.000000000000000,
                                radius = 10.000000000000000).
```

```
And it will have area 314.159265358979326.
Plus when we roll it, it travels 62.831853071795862 units!
```

So circular pizzas will be our first foray into rolling self-delivering pizzas!

The argument that `make-pizza` requires needs to be both a `Shape` and a `Rollable`. We do have other shapes available that are not `Rollable`. Here is what happens if we try to make a rectangular pizza for example.

```
make-pizza(Rectangle(0.0, 0.0, 5.0, 3.0))
```

Attempting to compile the above gives us the following error.

```
Cannot call function make-pizza of type Rollable&Shape -> False
with arguments of type (Rectangle).
```

Thus Stanza correctly prevents us from attempting to make pizzas out of shapes that don't roll.

4.8 The Flexibility of Functions

In the beginning of the chapter we said that Stanza's class-less object system gives you flexibility you wouldn't have in a typical class based language. Here is a demonstration of that.

The definition of the `Circle` struct in the `shapes` package defines circles using their x and y coordinates, and their radius. But what if, as a user, we don't like this convention and instead want to define circles given a `Point` to represent their center, and their diameter? Stanza allows us to easily do that.

Here is `shape-utils.stanza`, which contains a user defined package with his own utilities for managing shapes.

```
defpackage shape-utils :
   import core
   import shapes

public defn Circle (center:Point, diameter:Double) :
   Circle(x(center), y(center), diameter / 2.0)

public defn diameter (c:Circle) :
   radius(c) * 2.0

public defn center (c:Circle) :
   Point(x(c), y(c))
```

With this package, users can now use circles as if they were defined with a center point and a diameter instead of a pair of x, and y coordinates and a radius. Let's update our `main`

function to use this new convention.

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

...

defn main () :
   val ss = Vector<Shape>()
   add(ss, Point(1.0, 1.0))
   add(ss, Circle(Point(2.0, 2.0), 6.0))
   add(ss, Circle(Point(3.0, 0.0), 2.0))
   add(ss, Salinon(0.0, 0.0, 10.0, 5.0))
   add(ss, Salinon(5.0, -1.0, 8.0, 7.0))
   println("Shapes:")
   println(ss)
   println("Total area = %_" % [total-area(ss)])
```

Run the program to see that it continues to print out the same message as before.

```
Shapes:
[Point(1.000000000000000, 1.000000000000000)
 Circle(2.000000000000000, 2.000000000000000, radius = 3.000000000000000)
 Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)
 Salinon(0.000000000000000, 0.000000000000000,
         outer-radius = 10.000000000000000, inner-radius = 5.000000000000000)
 Salinon(5.000000000000000, -1.000000000000000,
         outer-radius = 8.000000000000000, inner-radius = 7.000000000000000)]
Total area = 384.845100064749658
```

Note that the `center` and `diameter` functions in the `shapes-utils` package are no less "special" or "fundamental" than the `x`, `y`, and `radius` functions in the `shapes` package. Users can use whichever representation they prefer. Most importantly, adding this functionality did not require the user to communicate with the authors of the `shapes` package at all.

We can similarly add support for a new representation of rectangles. Currently, a rectangle is represented using the x, and y coordinates of its bottom-left corner and its width and height. Let's add support for representing rectangles given its bottom-left point and its top-right point.

```
public defn Rectangle (bottom-left:Point, top-right:Point) :
   val w = x(top-right) - x(bottom-left)
   val h = y(top-right) - y(bottom-left)
```

```
   if (w < 0.0) or (h < 0.0) : fatal("Illegal Rectangle!")
   Rectangle(x(bottom-left), y(bottom-left), w, h)

public defn bottom-left (r:Rectangle) :
   Point(x(r), y(r))

public defn top-right (r:Rectangle) :
   Point(x(r) + width(r), y(r) + height(r))
```

Again, the `bottom-left` and `top-right` functions in the `shapes-utils` package are no less "fundamental" than the `x`, `y`, `width`, and `height` functions in the `shapes` package.

4.9 Fundamental and Derived Operations

Things in your program are modeled using *types* in Stanza. Anything than can be *done* using a type is implemented as a function (or multi). These operations are further categorized into *fundamental* and *derived* operations.

The `area` function for `Shape` objects is an example of a fundamental operation. At least in our `shapes` package, a shape is *defined* by its property of having an area. In fact, the *only* thing that all shapes have in common is that you can compute their area. And when defining a new type of shape, users *must* also define an appropriate method for `area`.

Here is an example of a *derived* operation on shapes.

```
public defn sort-by-area (xs:Array<Shape>) :
   var sorted? = false
   while not sorted? :
      sorted? = true
      for i in 0 to (length(xs) - 1) do :
         if area(xs[i + 1]) < area(xs[i]) :
            val a = xs[i]
            val b = xs[i + 1]
            xs[i] = b
            xs[i + 1] = a
            sorted? = false
```

This operation allows you to sort an array of shapes by increasing area. By reading its definition, you can see that it will work on *all* shapes, because the only operation it requires is the ability to call `area`. A derived operation is a function implemented in terms of a type's fundamental operations.

When defining a new subtype of an existing type, users must implement a small set of fundamental operations to ensure correct operation of their subtype. In the core library documentation, this set is called the *mandatory minimal implementation*.

The typical architecture of a Stanza program is to define a *small* number of fundamental operations for each type, coupled with a *large* library of derived operations. Structuring your program in this way gives you the most flexibility and extensibility. Adding new derived operations is as simple as defining a new function and is very easy. Defining new types is also easy as their mandatory minimal implementations are small.

4.10 Multiple Dispatch

The `area` multi in the `shapes` package accepts only a single argument, and at runtime it *dispatches* to the appropriate method depending on the type of the argument. Stanza places no restriction on the number of arguments that a multi can take, so users can write multis that dispatches to the appropriate method depending on the types of *multiple* arguments. This feature is called *multiple dispatch*.

We will demonstrate the power of multiple dispatch by writing an *overlaps?* function that decides whether two shapes are overlapping. Here is the definition of the multi.

```
public defmulti overlaps? (a:Shape, b:Shape) -> True|False
```

It returns `true` if the shape `a` overlaps with shape `b`, or `false` otherwise.

Points have zero area, so two points overlap only if they are exactly equal to each other.

```
defmethod overlaps? (a:Point, b:Point) :
   (x(a) == x(b)) and (y(a) == y(b))
```

A circle overlaps with a point if the distance between the point and the center of the circle is less than or equal to the radius of the circle.

```
defmethod overlaps? (a:Point, b:Circle) :
   val dx = x(b) - x(a)
   val dy = y(b) - y(a)
   val r = radius(b)
   dx * dx + dy * dy <= r * r
```

Stanza makes no assumption that `overlaps?` is commutative. So we explicitly tell Stanza that a circle overlaps with a point if the point overlaps with the circle.

```
defmethod overlaps? (a:Circle, b:Point) :
   overlaps?(b, a)
```

Finally, two circles overlap if the center of circle `a` overlaps with a circle with the same center as circle `b` but with radius equal to the sum of both circles.

```
defmethod overlaps? (a:Circle, b:Circle) :
   overlaps?(Point(x(a), y(a)), Circle(x(b), y(b), radius(b) + radius(a)))
```

With these definitions, `overlaps?` completely handles points and circles. Let's try it out.

```
defn test-overlap (a:Shape, b:Shape) :
   println(a)
   if overlaps?(a, b) : println("overlaps with")
   else : println("does not overlap with")
   println(b)
   print("\n")

defn try-overlaps () :
   val pt-a = Point(0.0, 0.0)
   val pt-b = Point(0.0, 3.0)
   val circ-a = Circle(0.0, 3.0, 1.0)
   val circ-b = Circle(3.0, 0.0, 1.0)
   val circ-c = Circle(0.0, 0.0, 3.0)
   test-overlap(pt-a, pt-b)
   test-overlap(circ-a, circ-b)
   test-overlap(pt-b, circ-b)
   test-overlap(circ-a, pt-b)
   test-overlap(circ-c, circ-a)

try-overlaps()
```

The above prints out

```
Point(0.000000000000000, 0.000000000000000)
does not overlap with
Point(0.000000000000000, 3.000000000000000)

Circle(0.000000000000000, 3.000000000000000, radius = 1.000000000000000)
does not overlap with
Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)

Point(0.000000000000000, 3.000000000000000)
does not overlap with
Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)

Circle(0.000000000000000, 3.000000000000000, radius = 1.000000000000000)
overlaps with
Point(0.000000000000000, 3.000000000000000)

Circle(0.000000000000000, 0.000000000000000, radius = 3.000000000000000)
overlaps with
Circle(0.000000000000000, 3.000000000000000, radius = 1.000000000000000)
```

As an exercise, you may try to implement the rest of the methods required for `overlaps?` to also work on rectangles. The brave and adventurous amongst you can try supporting salinons as well.

4.11 Ambiguous Methods

A multi dispatches to the most *specific* method appropriate for the types of the arguments. However, there are sometimes multiple methods that are equally specific, and it is ambiguous which method should be called.

As an example, consider the very strange shape, the `Blob`. The blob has the very strange property that *it* overlaps with no shape, but *every* shape overlaps with *it*.

```
defstruct Blob <: Shape
defmethod print (o:OutputStream, b:Blob) : print(o, "Amorphous Blob")
defmethod overlaps? (a:Blob, b:Shape) : false
defmethod overlaps? (a:Shape, b:Blob) : true
```

Let's try it out.

```
defn try-overlaps () :
   val pt-a = Point(0.0, 0.0)
   val pt-b = Point(0.0, 3.0)
   val circ-a = Circle(0.0, 3.0, 1.0)
   val circ-b = Circle(3.0, 0.0, 1.0)
   val circ-c = Circle(0.0, 0.0, 3.0)
   val blob = Blob()
   test-overlap(pt-a, blob)
   test-overlap(circ-a, blob)
   test-overlap(blob, pt-b)
   test-overlap(blob, circ-b)
```

The program prints out

```
Point(0.000000000000000, 0.000000000000000)
overlaps with
Amorphous Blob

Circle(0.000000000000000, 3.000000000000000, radius = 1.000000000000000)
overlaps with
Amorphous Blob

Amorphous Blob
does not overlap with
Point(0.000000000000000, 3.000000000000000)

Amorphous Blob
does not overlap with
Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)
```

But the real question is: does a blob overlap with a blob? Let's see.

```
defn try-overlaps () :
   val pt-a = Point(0.0, 0.0)
   val pt-b = Point(0.0, 3.0)
   val circ-a = Circle(0.0, 3.0, 1.0)
   val circ-b = Circle(3.0, 0.0, 1.0)
   val circ-c = Circle(0.0, 0.0, 3.0)
   val blob = Blob()
   test-overlap(pt-a, blob)
   test-overlap(circ-a, blob)
   test-overlap(blob, pt-b)
   test-overlap(blob, circ-b)
   test-overlap(blob, blob)
```

prints out

```
Point(0.000000000000000, 0.000000000000000)
overlaps with
Amorphous Blob

Circle(0.000000000000000, 3.000000000000000, radius = 1.000000000000000)
overlaps with
Amorphous Blob

Amorphous Blob
does not overlap with
Point(0.000000000000000, 3.000000000000000)

Amorphous Blob
does not overlap with
Circle(3.000000000000000, 0.000000000000000, radius = 1.000000000000000)

Amorphous Blob
FATAL ERROR: Ambiguous branch.
   at shapes.stanza:47.16
   at shapes.stanza:71.6
   at shapes.stanza:87.3
   at shapes.stanza:89.0
```

Stanza is telling us that there are multiple `overlaps?` methods that are equally specific for arguments of type `Blob` and `Blob`, and it does not know which one to call. To resolve this, users would have to define an `overlaps?` method specifically comparing `Blob` against `Blob`.

4.12 Revisiting Print

Now that you've been introduced to multis and methods, we can remove some of the mysteries surrounding the `print` function. So far, you've been told to follow a specific pattern to provide custom printing behaviour for your custom structs. For example, here is the print method defined for circles.

```
defmethod print (o:OutputStream, c:Circle) :
   print(o, "Circle(%_, %_, radius = %_)" % [x(c), y(c), radius(c)])
```

But now you can see that it is simply attaching a new method to a multi called `print`. The `print` multi is defined in the `core` package

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

and takes two arguments. The first is an `OutputStream` object that represents the *target* that you're printing to. The most common target is the standard output stream, i.e. the user's terminal. The second argument is the object that you're printing.

Thus far, you've only provided `print` methods for more specific types of `x` in order to print different types of objects. But later, you'll see how you can provide `print` methods for more specific types of `o` in order to print to different targets. And all of this works seamlessly due to the power of multiple dispatch.

4.13 The New Expression

The new expression is Stanza's fundamental construct for creating objects. All objects in Stanza are either literals (e.g. 1, 'x', "Timon"), or are created (directly or indirectly) by the new expression.

Let's define a type called `Stack` that represents a stack into which we can push and pop strings. Start a new file called `stack.stanza`. Here's the type definition for `Stack` and also two multis for the push and pop operations to which we will later attach methods, and a third multi for checking whether the stack is empty.

```
deftype Stack
defmulti push (s:Stack, x:String) -> False
defmulti pop (s:Stack) -> String
defmulti empty? (s:Stack) -> True|False
```

Let's provide it with custom printing behaviour.

```
defmethod print (o:OutputStream, s:Stack) :
   print(o, "Stack")
```

Now in our main function we will create a single `Stack` object and print it out.

```
defn main () :
```

```
   val s = new Stack
   println("s is a %_." % [s])

main()
```

Compile the program and run it. It should print out

```
s is a Stack.
```

Thus the expression

```
new Stack
```

creates a new `Stack` object. We say that it creates a new *instance* of type `Stack`.

But this stack object thus far isn't terribly useful. The only thing it can do is print itself. Stanza does allow us to call `push` and `pop` on the stack, but it will just crash because we haven't attached any methods yet.

Instance Methods

The new expression allows us to define *instance* methods for the object being created. Here is an instance method for the `empty?` multi for the stack being created.

```
defn main () :
   val s = new Stack :
      defmethod empty? (this) :
         true
   println("s is a %_." % [s])
   println("stack is empty? %_." % [empty?(s)])

main()
```

We haven't defined any methods for pushing strings to the stack yet, so the `empty?` method simply returns `true` for now. Compile the program and run it. It should print out

```
s is a Stack.
stack is empty? true.
```

The instance method declaration looks similar to the standard method declarations that you've already learned except for one major difference. The *this* argument is very special. In an instance method declaration, `this` refers specifically to the object currently being created by new. In this case, the object being created is `s`. So the instance method is saying: if `empty?` is called on `s` then return `true`. Every instance method must have exactly one argument named `this`.

In fact, now that we've learned about instance methods, let's redefine the `print` method as an instance method for `s`. Delete the top-level print method, and add the following.

```
defn main () :
   val s = new Stack :
      defmethod empty? (this) :
         true
      defmethod print (o:OutputStream, this) :
         print(o, "Stack")
   println("s is a %_." % [s])
   println("stack is empty? %_." % [empty?(s)])

main()
```

Compile and run the program and verify that it prints the same message as before.

The Push and Pop Methods

We will now define the methods for `push` and `pop`. The stack contents will be held in an array, and we'll keep track of how many items are currently in the stack using a `size` variable. The array will be of length 10, so the maximum number of strings that the stack can hold is 10. Declare the following *within* the `main` function.

```
val items = Array<String>(10)
var size = 0
```

Next we'll declare the `push` method. Declare the following within the new expression.

```
defmethod push (this, x:String) :
   if size == 10 : fatal("Stack is full!")
   items[size] = x
   size = size + 1
```

The `push` method first ensures that the stack is not full. Then it stores `x` in the next slot in the array and increments the stack's size by one.

Here's the correponding `pop` method.

```
defmethod pop (this) :
   if size == 0 : fatal("Stack is empty!")
   size = size - 1
   items[size]
```

The `pop` method first ensures that the stack is not empty. Then it decrements the stack's size by one, and returns the top item in the stack.

Here's the revised `empty?` method.

```
defmethod empty? (this) :
   size == 0
```

The stack is empty if its size is zero.

And finally, here's the revised `print` method.

```
defmethod print (o:OutputStream, this) :
   print(o, "Stack containing [")
   for i in 0 to size do :
      print(o, items[i])
      if i < size - 1 :
         print(o, " ")
   print(o, "]")
```

It iterates through and prints all the strings currently in the stack.

Putting all the pieces together gives us the following `main` function. To test the stack, we try pushing and popping a few strings.

```
defn main () :
   val items = Array<String>(10)
   var size = 0
   val s = new Stack :
      defmethod push (this, x:String) :
         if size == 10 : fatal("Stack is full!")
         items[size] = x
         size = size + 1
      defmethod pop (this) :
         if size == 0 : fatal("Stack is empty!")
         size = size - 1
         items[size]
      defmethod empty? (this) :
         size == 0
      defmethod print (o:OutputStream, this) :
         print(o, "Stack containing [")
         for i in 0 to size do :
            print(o, items[i])
            if i < size - 1 :
               print(o, " ")
         print(o, "]")

   println("1.")
   println(s)

   println("2.")
   push(s, "Pumbaa")
   println(s)

   println("3.")
   push(s, "and")
   push(s, "Timon")
```

```
   println(s)

   println("4.")
   val x = pop(s)
   println("Popped %_ from stack." % [x])
   println(s)

   println("5.")
   val y = pop(s)
   println("Popped %_ from stack." % [y])
   println(s)

main()
```

Compile and run the program. It should print out

```
1.
Stack containing []
2.
Stack containing [Pumbaa]
3.
Stack containing [Pumbaa and Timon]
4.
Popped Timon from stack.
Stack containing [Pumbaa and]
5.
Popped and from stack.
Stack containing [Pumbaa]
```

4.14 Constructor Functions

In the above example, we created a stack called `s` directly in the `main` function. You may be thinking that this seems like a lot of work to create a single stack! What if we need to create multiple stacks?

The solution is to simply move the stack construction code into a new function and call it once for each stack we want to create. Here is a function called `make-stack` that accepts a `capacity` argument for specifying the maximum size supported by the stack.

```
defn make-stack (capacity:Int) -> Stack :
   val items = Array<String>(capacity)
   var size = 0
   new Stack :
      defmethod push (this, x:String) :
         if size == capacity : fatal("Stack is full!")
```

```
      items[size] = x
      size = size + 1
   defmethod pop (this) :
      if size == 0 : fatal("Stack is empty!")
      size = size - 1
      items[size]
   defmethod empty? (this) :
      size == 0
   defmethod print (o:OutputStream, this) :
      print(o, "Stack containing [")
      for i in 0 to size do :
         print(o, items[i])
         if i < size - 1 :
            print(o, " ")
      print(o, "]")
```

Let's change our main function to create two stacks and push different strings into them.

```
defn main () :
   val s1 = make-stack(10)
   val s2 = make-stack(10)

   println("1.")
   push(s1, "Timon")
   push(s1, "Pumbaa")
   push(s1, "Nala")
   push(s2, "Ryu")
   push(s2, "Ken")
   push(s2, "Makoto")
   println(s1)
   println(s2)

   println("2.")
   println("Popped %_ from s1." % [pop(s1)])
   println("Popped %_ from s2." % [pop(s2)])
   println(s1)
   println(s2)

   println("3.")
   println("Popped %_ from s1." % [pop(s1)])
   println("Popped %_ from s2." % [pop(s2)])
   println(s1)
   println(s2)

main()
```

Compile and run the program. It should print out

```
1.
Stack containing [Timon Pumbaa Nala]
Stack containing [Ryu Ken Makoto]
2.
Popped Nala from s1.
Popped Makoto from s2.
Stack containing [Timon Pumbaa]
Stack containing [Ryu Ken]
3.
Popped Pumbaa from s1.
Popped Ken from s2.
Stack containing [Timon]
Stack containing [Ryu]
```

Notice especially that the two stacks created by the separate calls to `make-stack` contain different strings and operate independently of each other.

We call `make-stack` a *constructor* function for `Stack` objects because it returns newly created `Stack` objects. If you are familiar with the object systems in other languages it might surprise you to see that there is nothing particularly special about constructor functions in Stanza. They're just regular functions. This lack of distinction between constructors and functions is another contributing factor to Stanza's flexibility. Constructors in class based languages are typically more "special" than regular functions, and while any user can define functions for a class, only the library's author can define more constructors for a class.

As a note on style, we named the constructor function for `Stack` objects `make-stack` in order to avoid confusing you. But the idiomatic Stanza style is to give the same name to the constructor function as the type of object it is constructing. So `make-stack` would simply be named `Stack`, and you will distinguish based on context whether a reference to `Stack` refers to the type or the function.

As a reminder, even with the new expression, you are still encouraged to keep the number of fundamental operations for a type small, and then implement as much functionality as derived operations as possible.

As an exercise, try implementing a function called `UnboundedStack` that constructs a `Stack` object with no maximum capacity. Then try it in place of `Stack`, and observe that there is no behavioural difference (save for capacity limitations) between stacks created with `UnboundedStack` and stacks created with `Stack`.

4.15 Revisiting Defstruct

`defmulti`, `defmethod`, `deftype` and `new` forms the fundamental constructs of Stanza's class-less object system. The `defstruct` construct that you have been using thus far is merely a syntactic shorthand for a specific usage pattern of `new`. Let's take a peek at its internals.

Here is a struct definition for a `Dog` object with a name field and a mutable breed field.

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

The above can be equivalently written as

```
deftype Dog <: Animal
defmulti name (d:Dog) -> String
defmulti breed (d:Dog) -> String
defmulti set-breed (d:Dog, breed:String) -> False

defn Dog (name:String, initial-breed:String) -> Dog :
   var breed = initial-breed
   new Dog :
      defmethod name (this) : name
      defmethod breed (this) : breed
      defmethod set-breed (this, b:String) : breed = b
```

Thus, the `defstruct` construct expands to

1. a type definition,
2. getter functions for each of its fields,
3. setter functions for each of its mutable fields, and
4. a constructor function for creating instances of the type.

Chapter 5

Programming with First-Class Functions

Stanza fully supports and encourages functional programming, however "Functional Programming" is intentionally not the title of this chapter. In the community, the term functional programming has been used to refer to two different concepts. The first is the concept of programming with *first-class* functions, where functions themselves are passed as arguments and stored in datastructures. This is the subject of this chapter.

The second concept refers to a style of programming revolving around the mathematical definition of functions; so called *pure* functions. A pure function is guaranteed to return the same result if called with the same arguments, and also not affect the environment in any way (e.g. by printing to the terminal). This style of programming is largely an exercise in manipulating immutable datastructures. It is also a powerful paradigm and will be the subject of a later chapter.

5.1 Nested Functions

As a gentle introduction to first-class functions we will start with *nested* functions. We hope the concept will seem straightforward, and then later we'll reveal that they are actually quite sophisticated underneath.

Here is a function that sorts an array of integers in increasing order.

```
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
```

Let's try it out on an array of random numbers.

```
defn main () :
   val xs = Array<Int>(10)
   xs[0] = 510
   xs[1] = 923
   xs[2] = 671
   xs[3] = 811
   xs[4] = -129
   xs[5] = -581
   xs[6] = 233
   xs[7] = -791
   xs[8] = 899
   xs[9] = 313

   selection-sort(xs)
   println(xs)

main()
```

It should print out

```
[-791 -581 -129 233 313 510 671 811 899 923]
```

By reading through the algorithm, you can see that the larger problem of sorting the array is actually composed of a number of smaller subproblems. For example, the lines

```
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]
```

compute the index of the minimum element between index `i + 1` and index `n`. The lines

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

swaps the element at index `i` with the element at index `min-idx`. `selection-sort` is short enough that we can still understand the main algorithm even without explicitly dividing the problem into smaller ones. But as programs get larger, the ability to break up a larger

problem into smaller ones is very important. Nested functions gives us a lot of power for doing this.

Let's define a *nested* function, `index-of-min`, that takes two indices `start` and `end`, and returns the index of the minimum element between indices `start` (inclusive) and `end` (exclusive).

```
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
```

Let's define another nested function, `swap`, that swaps the element in index `i` with the element in index `j`.

```
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
```

And now let's clean up our `selection-sort` function using these 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

   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))
```

The code is slightly longer than before, but the *overall* algorithm is *much* clearer now.

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

In English, it says: iterate with index `i` starting from 0 and proceeding to the end of the array, and swap the element at `i` with the minimum element in the rest of the array.

Notice that the nested functions `index-of-min` and `swap` are not merely functions declared within the body of `selection-sort`. If you tried to declare them as top-level functions, the program would give you this error when you try to compile it,

```
Could not resolve xs.
```

indicating that `xs` is not in scope and is not visible to `index-of-min` or `swap`. Part of the power of nested functions rests in them being able to refer to values defined in the function they're nested in.

Example: Permutations

Here is another example of using nested functions to greatly simplify code. The `permutations` function accepts an array of strings and prints out all possible permutations of its contents.

```
defn permutations (xs:Array<String>) :
   val n = length(xs)

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

   defn permute (i:Int) :
      if i < n - 1 :
         for j in i to n do :
            swap(i, j)
            permute(i + 1)
            swap(i, j)
      else :
         println(xs)

   permute(0)
```

It internally relies upon the nested functions `swap` and `permute`.

Let's try it out with these strings.

```
defn main () :
   val xs = to-array<String>(["All" "Dogs" "Are" "Awesome"])
   permutations(xs)

main()
```

When compiled and ran, it prints out

```
["All" "Dogs" "Are" "Awesome"]
["All" "Dogs" "Awesome" "Are"]
["All" "Are" "Dogs" "Awesome"]
["All" "Are" "Awesome" "Dogs"]
["All" "Awesome" "Are" "Dogs"]
["All" "Awesome" "Dogs" "Are"]
["Dogs" "All" "Are" "Awesome"]
["Dogs" "All" "Awesome" "Are"]
["Dogs" "Are" "All" "Awesome"]
["Dogs" "Are" "Awesome" "All"]
["Dogs" "Awesome" "Are" "All"]
["Dogs" "Awesome" "All" "Are"]
["Are" "Dogs" "All" "Awesome"]
["Are" "Dogs" "Awesome" "All"]
["Are" "All" "Dogs" "Awesome"]
["Are" "All" "Awesome" "Dogs"]
["Are" "Awesome" "All" "Dogs"]
["Are" "Awesome" "Dogs" "All"]
["Awesome" "Dogs" "Are" "All"]
["Awesome" "Dogs" "All" "Are"]
["Awesome" "Are" "Dogs" "All"]
["Awesome" "Are" "All" "Dogs"]
["Awesome" "All" "Are" "Dogs"]
["Awesome" "All" "Dogs" "Are"]
```

As an exercise, try writing a function called `combinations` that prints out all *combinations* of an array of strings instead of all *permutations*.

5.2 Functions as Arguments

The `selection-sort` function in the previous example sorted the array in increasing order. But there are many ways to sort an array of integers. The following `sort-by-abs` function sorts the array by their *absolute* values.

```
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))
```

If you replace the call to `selection-sort` in the `main` function with `sort-by-abs` then it now prints out

```
[-129 233 313 510 -581 671 -791 811 899 923]
```

Here is yet another way of sorting an array. The following `sort-by-sum-of-digits` function sorts the array by the total sum of their individual digits.

```
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 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))
```

Replacing the call to `selection-sort` with `sort-by-sum-of-digits` prints out

```
[510 313 233 811 -129 671 -581 923 -791 899]
```

You'll have noticed by now that the implementation of each sorting function is almost entirely identical except for one line. Here are the three different comparison functions.

```
;Compare value directly
xs[i] < min-val

;Compare absolute values
abs(xs[i]) < abs(min-val)

;Compare the sum of their digits
sum-of-digits(xs[i]) < sum-of-digits(min-val)
```

Couldn't we somehow write a general sort function and give it a specific way to compare things? We can! And the solution is to accept a *key* function that, for each item in the array, computes the value you wish to sort by.

Here is the general sorting function, `sort-by`, that accepts a key function `key`.

```
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))
```

Notice especially the type of the `key` argument.

```
Int -> Int
```

This says that `key` is a *function* that accepts a single argument, an `Int`, and returns an `Int`.

We can update our `main` function to sort the array in three different ways by using three different key functions.

```
defn identity (x:Int) : x

defn main () :
   val xs = Array<Int>(10)
   xs[0] = 510
   xs[1] = 923
   xs[2] = 671
   xs[3] = 811
   xs[4] = -129
   xs[5] = -581
   xs[6] = 233
   xs[7] = -791
   xs[8] = 899
   xs[9] = 313

   println("Sort by value directly.")
   sort-by(identity, xs)
   println(xs)

   println("Sort by absolute value.")
   sort-by(abs, xs)
   println(xs)

   println("Sort by sum of digits.")
   sort-by(sum-of-digits, xs)
   println(xs)

main()
```

Compiling and running the program prints out

```
Sort by value directly.
[-791 -581 -129 233 313 510 671 811 899 923]
Sort by absolute value.
[-129 233 313 510 -581 671 -791 811 899 923]
Sort by sum of digits.
[510 313 233 811 -129 671 -581 923 -791 899]
```

Up until now, we have always referred to a function in *function call position*. For example,

```
abs( ... )
```

```
sum-of-digits( ... )
```

But now you see that you can actually refer to functions directly as values to be passed to other functions!

```
sort-by(abs, xs)
sort-by(sum-of-digits, xs)
```

Functions that take functions as arguments are called *higher-order functions*. They are an *extremely* powerful programming technique, and you'll soon see that you've already been using them everywhere without knowing it.

5.3 Functions as Return Values

When a language has *first-class* functions, it means that functions can be treated as values. In the previous section we saw how to pass functions as arguments. Now we'll see how to use functions as return values.

Here's a function called `digit` that accepts a single argument `n`, and returns a *function*. What the *returned* function does is extract and return the `n`'th significant digit from its argument.

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

Let's try it out on some numbers.

```
defn main () :
   val first-digit = digit(0)
   val third-digit = digit(2)

   defn test-first-digit (x:Int) :
      println("The first digit of %_ is %_." % [x, first-digit(x)])
   test-first-digit(413)
   test-first-digit(-313)
   test-first-digit(41)
   test-first-digit(137)
   test-first-digit(991)

   defn test-third-digit (x:Int) :
```

```
      println("The third digit of %_ is %_." % [x, third-digit(x)])
   test-third-digit(413)
   test-third-digit(-313)
   test-third-digit(41)
   test-third-digit(137)
   test-third-digit(991)

main()
```

Compiling and running the program prints out

```
The first digit of 413 is 3.
The first digit of -313 is 3.
The first digit of 41 is 1.
The first digit of 137 is 7.
The first digit of 991 is 1.
The third digit of 413 is 4.
The third digit of -313 is 3.
The third digit of 41 is 0.
The third digit of 137 is 1.
The third digit of 991 is 9.
```

The type signature of `digit` is daunting at first.

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

Let's decipher it piece by piece. `digit` is a function that takes a single `Int` argument, and returns a `(Int -> Int)`. And we learned previously that a `(Int -> Int)` is a one argument function that takes an `Int` and returns an `Int`. The parentheses around `Int -> Int` is not strictly necessary as `->` is a right-associative operator. Thus, `digit` can also be declared the following way.

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

Write it in the way that is most clear to you. As an exercise, think about what the type of `digit` is.

Sorting By Digit

Now that we have a function for creating functions that are compatible with what is expected by `sort-by`, let's use `sort-by` to sort by different digits. Update the `main` function in our previous example.

```
defn main () :
   val xs = Array<Int>(10)
   xs[0] = 510
   xs[1] = 923
   xs[2] = 671
```

```
xs[3] = 811
xs[4] = -129
xs[5] = -581
xs[6] = 233
xs[7] = -791
xs[8] = 899
xs[9] = 313

println("Sort by value directly.")
sort-by(identity, xs)
println(xs)

println("Sort by absolute value.")
sort-by(abs, xs)
println(xs)

println("Sort by sum of digits.")
sort-by(sum-of-digits, xs)
println(xs)

println("Sort by first digit.")
sort-by(digit(0), xs)
println(xs)

println("Sort by second digit.")
sort-by(digit(1), xs)
println(xs)

println("Sort by third digit.")
sort-by(digit(2), xs)
println(xs)
```

Compile and run the program. It should print out

```
Sort by value directly.
[-791 -581 -129 233 313 510 671 811 899 923]
Sort by absolute value.
[-129 233 313 510 -581 671 -791 811 899 923]
Sort by sum of digits.
[510 313 233 811 -129 671 -581 923 -791 899]
Sort by first digit.
[510 811 671 -581 -791 233 313 923 -129 899]
Sort by second digit.
[510 811 313 923 -129 233 671 -581 -791 899]
Sort by third digit.
[-129 233 313 510 -581 671 -791 811 899 923]
```

Isn't that elegant! This is but a small demonstration of the power of first-class functions.

5.4 Core Library Functions

The `sort-by` function is so general and useful that you might wonder whether it's already included in Stanza's core library. And it is, along with many other useful higher order functions. We'll show you a few of them here.

qsort!

The `qsort!` function is Stanza's included sorting function. It implements the quick sort algorithm, and you can use it sort collections in much the same way that you used the `sort-by` function. One big difference, though, is that `qsort!` works on many different kinds of objects whereas your `sort-by` function only worked on `Int` objects.

```
val xs = Vector<String>()
add(xs, "Patrick")
add(xs, "Luca")
add(xs, "Emmy")
add(xs, "Sunny")
add(xs, "Whiskey")
add(xs, "Rummy")
qsort!(xs)
println(xs)
```

The above is an example of sorting a vector of strings, and it prints out

```
["Emmy" "Luca" "Patrick" "Rummy" "Sunny" "Whiskey"]
```

`qsort!` can optionally take a key function as its first argument for computing the item with which to sort. Here's how to sort the `xs` vector by the second letter.

```
defn second-letter (s:String) : s[1]
qsort!(second-letter, xs)
println(xs)
```

Running the program prints out

```
["Patrick" "Whiskey" "Emmy" "Rummy" "Luca" "Sunny"]
```

The third form of `qsort!` takes, as its second argument, a *comparison* function with which to sort by. The comparison function is given two items from the collection and must return `true` if the first argument is less than the second argument, or `false` otherwise.

Here is an example of sorting a vector containing both integers and strings. Integers are less than other integers if their numeric value is smaller. Strings are compared against

other strings according to their lexicographic order. And integers are less than strings if the integer is less than the length of the string.

```
val xs = Vector<Int|String>()
add(xs, 1)
add(xs, 3)
add(xs, "A")
add(xs, "B")
add(xs, 4)
add(xs, -10)
add(xs, "Timon")
add(xs, "Pumbaa")
add(xs, 42)

defn compare-items (a:Int|String, b:Int|String) :
   match(a, b) :
      (a:Int, b:Int) : a < b
      (a:Int, b:String) : a < length(b)
      (a:String, b:Int) : length(a) < b
      (a:String, b:String) : a < b
qsort!(xs, compare-items)
println(xs)
```

Running the program prints out

```
[-10 1 "A" "B" 3 4 "Pumbaa" "Timon" 42]
```

find

The `find` function looks for the first item in a collection that satisfies a condition. The condition is given as a function, and takes a single argument representing an item from the collection. The condition function must return `true` if the item satisfies the condition, or `false` otherwise. `find` returns the item if it is found, or `false` otherwise.

Here is an example of looking for the first capitalized word in a vector of strings.

```
val xs = Vector<String>()
add(xs, "they")
add(xs, "call")
add(xs, "me")
add(xs, "Mr")
add(xs, "Pig")

defn capitalized? (x:String) : upper-case?(x[0])
println(find(capitalized?, xs))
```

Running the program prints out

```
Mr
```

index-when

The `index-when` function is similar to `find`, and looks for the first item in a collection that satisfies a condition. The difference is if the item is found, then `index-when` returns its *index*.

Calling `index-when` instead of `find` on the previous example

```
println(index-when(capitalized?, xs))
```

prints out

```
3
```

Maybe Objects and first

A `Maybe` is used to indicate the presence or absence of an object. A `None` object is a subtype of `Maybe` and indicates there is no object. A `One` object is a subtype of `Maybe` and contains a wrapped object. You can retrieve the wrapped object in a `One` object using the `value` function.

The `first` function takes an argument function, `f`, and a collection `xs`, and calls `f` repeatedly on each item in the collection. `f` must return a `Maybe` object. `first` returns the first `One` object that is returned by `f`, or a `None` object if no call to `f` returns a `One` object.

Here is an example of using first to find the first even sum of digits in a vector of integers.

```
val xs = Vector<Int>()
add(xs, 14)
add(xs, 78)
add(xs, 232)
add(xs, 787)
add(xs, 49)

defn even-sum? (x:Int) :
   val s = sum-of-digits(x)
   if s % 2 == 0 : One(s)
   else : None()
match(first(even-sum?, xs)) :
   (x:One<Int>) :
      println("The first even sum of digits is %_." % [value(x)])
   (x:None) :
      println("No number in xs has an even sum of digits.")
```

map!

The `map!` function takes a function `f` and an array (or vector) `xs`. It then iterates through the array and replaces each item in the array with a call to `f` on the item.

Here is how to capitalize each entry in a vector of strings using `map!`.

```
val xs = Vector<String>()
add(xs, "they")
add(xs, "call")
add(xs, "me")
add(xs, "Mr")
add(xs, "Pig")

defn capitalize (x:String) :
   append(upper-case(x[0 to 1]), x[1 to false])
map!(capitalize, xs)
println(xs)
```

When ran, it prints out

```
["They" "Call" "Me" "Mr" "Pig"]
```

all?, any?, none?

`all?` is used to determine whether *every* item in a collection satisfies some condition. The `all?` function takes a function `f` and a collection `xs`. It returns `true` if calling `f` on every item in `xs` returns `true`. If `f` returns `false` on any item then `all?` immediately returns `false`.

Here is how we can use `all?` to test whether all numbers in `xs` are positive.

```
val xs = Vector<Int>()
add(xs, 4)
add(xs, 2)
add(xs, 3)
add(xs, -8)
add(xs, 5)

defn positive? (x:Int) : x > 0
all?(positive?, xs)
```

The `any?` and `none?` functions work similarly. `any?` determines whether *any* item satisfies the condition, and `none?` determines whether *no* item satisfies the condition.

do

Finally we get to the *most* commonly used higher order function of them all: the `do` function. The `do` function takes a function `f` and a collection `xs` and calls `f` on each item in the collection.

Here is how to report the lengths of every string in a vector using `do`.

```
val xs = Vector<String>()
add(xs, "they")
add(xs, "call")
add(xs, "me")
add(xs, "Mr")
add(xs, "Pig")

defn report-length (x:String) :
   println("%_ has length %_." % [x, length(x)])
do(report-length, xs)
```

When ran, it prints out

```
they has length 4.
call has length 4.
me has length 2.
Mr has length 2.
Pig has length 3.
```

At this point, particularly precocious readers might start to suspect that they have already used `do` in their programs without knowing it.

5.5 Anonymous Functions

Before the introduction of higher-order functions it was natural for you to give every function in your program a name. After all, if a function has no name, then how would you call it? But after having been exposed to higher-order functions, you might now be wondering if it's possible to *avoid* giving functions a name. A lot of functions are now only ever used once, and only as an argument to another higher-order function.

Anonymous functions are functions without names. Here is `report-length` from the previous example written as an anonymous function.

```
fn (x:String) :
   println("%_ has length %_." % [x, length(x)])
```

Here is an example of rewriting the `do` example using an anonymous function.

```
val xs = Vector<String>()
```

```
add(xs, "they")
add(xs, "call")
add(xs, "me")
add(xs, "Mr")
add(xs, "Pig")
do(fn (x:String) :
      println("%_ has length %_." % [x, length(x)])
   xs)
```

Notice how the `report-length` function is now directly created using `fn` and passed immediately as an argument to `do`. The arguments to higher-order functions are often very short and anonymous functions provides a convenient syntax for using them.

Bidirectional Type Inference

The type inference rules for anonymous functions are different than those for named functions. For a named function, if a type annotation is left off of an argument, then the argument is assumed to have the `?` type, and can accept anything. For an anonymous function, if a type annotation is left off of an argument, then the argument's type is inferred from the context in which the function is used.

Thus the call to `do` in the above example could be more concisely written as

```
do(fn (x) : println("%_ has length %_." % [x, length(x)])
   xs)
```

From the context, the type of `xs` is `Vector<String>`, and since `do` calls the function on each item in `xs`, it is obvious that `x` must be of type `String`.

Idiomatic Stanza code rarely contains type annotations for anonymous functions, and instead relies upon type inference. In certain circumstances, Stanza will be unable to infer the argument types, in which case you'll have to provide them explicitly.

Anonymous Function Shorthand

For *extremely* short anonymous functions, Stanza provides a syntactic shorthand. The following function

```
fn (x) : x + 1
```

can be written equivalently as

```
{_ + 1}
```

As another example, the following function

```
fn (x, y) : x + 2 * y
```

can be written equivalently as

```
{_ + 2 * _}
```

The shorthand consists of surrounding the function body with the `{}` brackets, and using underscores to denote arguments. To create anonymous functions with explicit type annotations use the following form.

```
fn (x:Int, y:String) : x + length(y)
```

can be written equivalently as

```
{_:Int + 2 * _:String}
```

Curried Function Shorthand

For *extremely* short anonymous functions consisting of a single function call, Stanza provides another syntactic shorthand. The following function

```
fn (xs) : qsort!(abs, xs)
```

can be written equivalently as

```
qsort!{abs, _}
```

Similarly, to create anonymous functions with explicit type annotations use the following form.

```
fn (i:Int) : index-of-min(i, length(xs))
```

can be written equivalently as

```
index-of-min{_:Int, length(xs)}
```

The Application Operator

With the introduction of anonymous functions, you'll find that you can implement lots of functionality using single lines of code. To help with this programming style, Stanza provides the `\$` operator to help reduce the number of nested expressions. The expression

```
f \$ x
```

is equivalent to

```
f(x)
```

Thus the `\$` operator is just a shorthand for function application. Notice, however, that with the `\$` operator, the above expression did not require any parentheses.

There is a very common usage pattern involving both curried functions and the `\$` operator. Here is what it looks like.

```
f{x, _} \$ y
```

Let's remove the syntactic shorthands incrementally to figure out what the above means. First, we will write out the `\$` operator in full.

```
(f{x, _})(y)
```

Next, we will write out the curried function in full.

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

So the expression, when written in full, just creates an anonymous function and then immediately calls it with `y`. Calling the anonymous function is then equivalent to

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

Thus the expression

```
f{x, _} \$ y
```

is equivalent to

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

You can think of the `\$` operator as substituting the right hand side expression in for the underscores in the left hand side expression.

This usage pattern is often used to chain a long sequence of operations together.

```
f{1, _} \$
   head \$
   g{2, _} \$
   xs[2]
```

is a shorthand for writing

```
val result1 = xs[2]
val result2 = g(2, result1)
val result3 = head(result2)
f(1, result3)
```

Here is a concrete example of an idiomatic usage of the `\$` operator. In the demonstration of the `qsort!` operator, we explicitly created a `compare-items` function to pass into `qsort!`.

```
defn compare-items (a:Int|String, b:Int|String) :
   match(a, b) :
      (a:Int, b:Int) : a < b
      (a:Int, b:String) : a < length(b)
      (a:String, b:Int) : length(a) < b
      (a:String, b:String) : a < b
qsort!(xs, compare-items)
```

But here is another way it can be (and often is) written.

```
qsort!{xs, _} \$ fn (a, b) :
   match(a, b) :
      (a:Int, b:Int) : a < b
      (a:Int, b:String) : a < length(b)
      (a:String, b:Int) : length(a) < b
      (a:String, b:String) : a < b
```

Since the types of the arguments of anonymous functions are inferred, there is also no need to provide explicit type annotations.

5.6 The For Construct

Now that you've been introduced to anonymous functions and higher-order functions, we are now ready to introduce the *full* for construct. The expression

```
for x in xs do :
   println(x)
```

is equivalent to

```
do(fn (x) :
      println(x)
   xs)
```

As mentioned before, the for construct is not a looping mechanism. It is a syntactic shorthand for calling higher-order functions of a certain form. As explained earlier, the `do` function is what's responsible for looping over each element in `xs`.

The for construct can be called with multiple bindings as well.

```
for (x in xs, y in ys) do :
   println(x + y)
```

is equivalent to

```
do(fn (x, y) :
      println(x + y)
   xs, ys)
```

Thus, in general, the for construct expands to a call to a higher order function where the first argument is an anonymous function followed by n remaining arguments. The anonymous function must take n arguments, one for each of the remaining arguments.

There are forms of the `do` function that accept multiple collections. The collections are iterated over in parallel, and iteration stops when it reaches the end of any one of them. The following example prints every item in a vector coupled with its index.

```
val xs = Vector<String>()
add(xs, "Patrick")
```

```
add(xs, "Luca")
add(xs, "Emmy")

for (x in xs, i in 0 to false) do :
   println("Element %_ is at index %_." % [x, i])
```

It prints out

```
Element Patrick is at index 0.
Element Luca is at index 1.
Element Emmy is at index 2.
```

Operating Functions

`find`, `all?`, `any?`, `none?`, and `index-when?` also have multiple collection versions that can be used with a for construct with multiple bindings.

Functions like `do`, with type signatures compatible with the for construct, are called *operating functions*. Stanza's core library includes a large number of commonly used ones. For more details, read the reference documentation.

Occasionally, it might also be appropriate to implement your own operating function. Here is an operating function that first iterates over each odd integer in a vector and then iterates over each even integer in the vector.

```
defn do-odd-then-even (f: Int -> ?, xs:Vector<Int>) :
   for x in xs do :
      if x % 2 != 0 : f(x)
   for x in xs do :
      if x % 2 == 0 : f(x)
```

Here is an example of using it in conjunction with the for construct.

```
val xs = Vector<Int>()
add(xs, 1)
add(xs, 3)
add(xs, 2)
add(xs, 6)
add(xs, 5)
add(xs, 2)
add(xs, 4)
add(xs, 3)

for x in xs do-odd-then-even :
   println(x)
```

Compiling and running the above prints out

```
1
3
5
3
2
6
2
4
```

5.7 Stanza Idioms

With our new knowledge of anonymous functions, curried functions, and the for construct, we can now revisit our examples of using the core library and write them using standard Stanza idioms and functions.

```
;A vector of strings
val strs = Vector<String>()
add-all(strs, ["patrick", "luca", "emmy", "Sunny", "whiskey", "Rummy"])

;Sort by the second letter
println("1.")
qsort!({_[1]}, strs)
println(strs)

;A vector of ints and strings
val int-strs = Vector<Int|String>()
add-all(int-strs, [1, 3, "A", "B", 4, -10, "Timon", "Pumbaa", 42])

;Sort using custom comparison function
println("\n2.")
qsort!{int-strs, _} \$ fn (a, b) :
   match(a, b) :
      (a:Int, b:Int) : a < b
      (a:Int, b:String) : a < length(b)
      (a:String, b:Int) : length(a) < b
      (a:String, b:String) : a < b
println(xs)

;Find first capitalized word
println("\n3.")
println \$ for s in strs find :
   upper-case?(x[0])

;Find index of first capitalized word
```

```
println("\n4.")
println \$ for s in strs index-when :
   upper-case?(x[0])

;Capitalize every word
println("\n5.")
for s in strs map! :
   append(upper-case(x[0 to 1]), x[1 to false])
println(strs)

;A vector of integers
val ints = Vector<Int>()
add-all(ints, [4, 2, 3, -8, 5])

;Are they all positive?
println("\n6.")
println \$ for x in ints all? :
   x > 0

;Report lengths of string along with their index
println("\n7.")
for (s in strs, i in 0 to false) do :
   println("strs[%_] = %_. Length = %_." % [i, strs[i], length(strs[i])])
```

Compiling and running all of the above print outs

```
1.
["patrick" "whiskey" "emmy" "Rummy" "luca" "Sunny"]

2.
[-10 1 "A" "B" 3 4 "Pumbaa" "Timon" 42]

3.
Rummy

4.
3

5.
["Patrick" "Whiskey" "Emmy" "Rummy" "Luca" "Sunny"]

6.
false

7.
strs[0] = Patrick. Length = 7.
```

```
strs[1] = Whiskey. Length = 7.
strs[2] = Emmy. Length = 4.
strs[3] = Rummy. Length = 5.
strs[4] = Luca. Length = 4.
strs[5] = Sunny. Length = 5.
```

5.8 Tail Calls

Consider the following function for computing the sum of all the positive integers less than or equal to `n`.

```
defn sum-of (n:Int) :
   if n > 0 : n + sum-of(n - 1)
   else : 0
```

Calling `sum-of(6)` returns `21`.

Here's a visualization of the *execution context* as that operation is being performed.

```
sum-of(6) =
6 + sum-of(5) =
6 + 5 + sum-of(4) =
6 + 5 + 4 + sum-of(3) =
6 + 5 + 4 + 3 + sum-of(2) =
6 + 5 + 4 + 3 + 2 + sum-of(1) =
6 + 5 + 4 + 3 + 2 + 1 + sum-of(0) =
6 + 5 + 4 + 3 + 2 + 1 + 0 =
6 + 5 + 4 + 3 + 2 + 1 =
6 + 5 + 4 + 3 + 3 =
6 + 5 + 4 + 6 =
6 + 5 + 10 =
6 + 15 =
21
```

Observe that the computation of `sum-of(6)` requires knowing the result of `sum-of(5)`. And so we then recursively compute the result of `sum-of(5)` *while remembering* that we should add `6` to the result to get the final answer. Similarly, the computation of `sum-of(5)` requires the result of `sum-of(4)`. Et cetera.

Each recursive invocation of `sum-of` requires us to remember what to do with the result. This is our execution context, and in Stanza, is saved in a stack of *activation records*. How big does this stack get? Well, for `sum-of`, it grows to contain exactly `n` *activation records*.

This can be verified by forcing a program failure when `n` is equal to `0` and looking at the *stack trace*.

```
defn sum-of (n:Int) :
```

```
   if n > 0 : n + sum-of(n - 1)
   else : fatal("n reached zero.")
```

When compiled and ran, the above prints out

```
FATAL ERROR: n reached zero.
   at test.stanza:7.10
   at test.stanza:6.18
   at test.stanza:6.18
   at test.stanza:6.18
   at test.stanza:6.18
   at test.stanza:6.18
   at test.stanza:6.18
   at test.stanza:9.0
```

Each `test.stanza:6.18` entry in the stack trace refers to a recursive call to `sum-of(n - 1)`. Thus you can see that there are `6` entries corresponding to calling `sum-of(6)`. *activation records* take up space. If `n` is too large, then eventually the stack of activation records will consume all of your program memory.

An Iterative Algorithm

Now consider this alternative implementation of `sum-of`.

```
defn sum-of (n:Int) :
   x+sum-of(0, n)

defn x+sum-of (x:Int, n:Int) :
   if n > 0 : x+sum-of(x + n, n - 1)
   else : x
```

Here's a visualization of the execution context of computing `sum-of(6)`.

```
sum-of(6) =
x+sum-of(0, 6) =
x+sum-of(6, 5) =
x+sum-of(11, 4) =
x+sum-of(15, 3) =
x+sum-of(18, 2) =
x+sum-of(20, 1) =
x+sum-of(21, 0) =
21
```

Similar to before, the computation of `x+sum-of(0, 6)` requires knowing the result of `x+sum-of(6, 5)`. And so we then recursively compute the result of `x+sum-of(6, 5)`. But this time, we don't have to remember what to do with the result! The result of

`x+sum-of(6, 5)` *is* the result of `x+sum-of(0, 6)`, so just return whatever `x+sum-of(6, 5)` returns.

In concrete terms, what this means is that Stanza can discard the activation record for `x+sum-of(0, 6)` immediately before calling `x+sum-of(6, 5)` since there's no context to remember. If that is done, then the number of activation records does not grow, no matter how large `n` is. And thus the program will not run out of memory. This optimization is called *tail call optimization*.

Tail Call Optimization

By default, Stanza does not optimize tail calls in functions. This is done for the purposes of debugging. It is useful to have a complete stack trace, even if not strictly necessary for correct operation of the algorithm. To tell Stanza to optimize tail calls in a function, we have to explicitly declare the function as being tail call optimized.

```
defn sum-of (n:Int) :
   x+sum-of(0, n)

defn* x+sum-of (x:Int, n:Int) :
   if n > 0 : x+sum-of(x + n, n - 1)
   else : x
```

`defn*` is the tail call optimized version of `defn`. Similarly, `defmethod*` is the tail call optimized version of `defmethod`, and `fn*` is the tail call optimized version of `fn`.

To verify that the stack frames are properly being discarded, we'll again force the program to fail when `n` is equal to `0` and examine the stack trace.

```
defn* x+sum-of (x:Int, n:Int) :
   if n > 0 : x+sum-of(x + n, n - 1)
   else : fatal("n reached zero.")
```

Compiling and running the program prints

```
FATAL ERROR: n reached zero.
   at tests/test.stanza:6.3
   at tests/test.stanza:20.10
```

There are now *no* stack frames corresponding to `x+sum-of`. They have all been discarded.

5.9 Revisiting While

With the introduction of tail calls, it is time for us to unveil the internals of the while loop construct. The expression

```
var i = 0
while i < 10 :
   println(i)
   i = i + 1
```

is equivalent to

```
var i = 0
defn* loop () :
   if i < 10 :
      println(i)
      i = i + 1
      loop()
loop()
```

Thus the while construct simply defines a local tail call optimized function and then calls it.

Directly expressing loops using recursive functions can often be much more natural than using a while loop. A while loop loops by default, and the programmer has to specify when it should *stop* looping. In constrast, a recursive function does *not* loop by default, and the programmer instead specifies when to perform another iteration.

Here is an example of using a tail call optimized function for finding the index of an integer `x` in a *sorted* vector `xs` using binary search.

```
defn bsearch (x:Int, xs:Vector<Int>) :
   label<Int|False> return :
      defn* loop (start:Int, end:Int) :
         if start < end :
            val mid = start + (end - start) / 2
            if x < xs[mid] : loop(start, mid)
            else if x > xs[mid] : loop(mid + 1, end)
            else : return(mid)
      loop(0, length(xs))
```

`loop` finds the index of `x`, assuming that it exists between the `start` index and `end` index (exclusive). It does this by computing a midpoint, `mid`, between the two indices. If `x` is less than the element at `mid` then it looks again in the first half of the range. If `x` is greater than the element at `mid` then it looks again in the second half of the range. Otherwise it has found `x`, and it is at index `mid`.

Let's try looking for the numbers `1`, `14`, and `13`.

```
defn main () :
   val xs = Vector<Int>()
   add-all(xs, [1,3,4,7,8,11,14,18,20,35])
   println(bsearch(1, xs))
   println(bsearch(14, xs))
   println(bsearch(13, xs))
```

```
main()
```

When compiled and ran, the above prints out

```
0
6
false
```

Chapter 6

Programming with Sequences

A sequence is a series of objects. At any point, you may ask whether a sequence is empty, and if it is not empty you may retrieve the next object. Many datastructures can represent their items as a sequence. For example, a sequence for representing the items in an array could begin with the item at index `0`. Subsequent items in the sequence would correspond to subsequent items in the array. When it reaches the end of the array then the sequence is empty.

While sequences are not a core language feature, they do play a fundamental part in the design of Stanza's core library. In this chapter we'll see how to fully exploit their power, and by doing so, avoid having to repeatedly reimplement many common programming patterns ourselves.

6.1 Fundamental Operations

A sequence is represented by the `Seq` type in Stanza. Let's first create a sequence containing all the strings in a tuple.

```
val xs = to-seq(["Timon" "and" "Pumbaa" "are" "good" "friends."])
```

This creates the sequence `xs`, which has type `Seq<String>` indicating that it is a sequence of strings.

Fundamentally, a sequence is defined by three operations.

1. You may ask whether a sequence is empty.
2. You may take a peek at the next item in the sequence.
3. You may take out the next item in the sequence.

Here is how to ask whether our `xs` sequence is empty.

```
if empty?(xs) :
```

```
   println("xs is empty.")
else :
   println("xs is not empty.")
```

which prints out

```
xs is not empty.
```

because we haven't taken anything out of the sequence yet.

If the sequence is not empty, then you can take a peek at the next item in the sequence like this.

```
val x0 = peek(xs)
println("The next item is %_." % [x0])
```

which prints out

```
The next item is Timon.
```

Peeking at an empty sequence is a fatal error.

Peeking at a sequence does not change the state of a sequence. If you peek again at the same sequence, it returns the same thing.

```
val x1 = peek(xs)
println("The next item is still %_." % [x1])
```

which prints out

```
The next item is still Timon.
```

Once you've determined that the sequence is not empty, you may take out the next item in the sequence.

```
val y0 = next(xs)
println("Took out item %_ from xs." % [y0])
```

which prints out

```
Took out item Timon from xs.
```

Calling `next` on a sequence *does* change the state of a sequence. If you call `next` again on the same sequence, it will return the following item in the sequence.

```
val y1 = next(xs)
println("Now took out item %_ from xs." % [y1])
```

which prints out

```
Now took out item and from xs.
```

Here is the standard pattern for printing out all the items in a sequence.

```
while not empty?(xs) :
   println("Next item is %_" % [next(xs)])
```

which prints out

```
Next item is Timon
Next item is and
Next item is Pumbaa
Next item is are
Next item is good
Next item is friends.
```

6.2 Writing a Sequence Function

Let's now write a function that takes a sequence argument. `cum-sum` takes a sequence of integers, `xs`, and returns a vector containing the cumulative sum of all the numbers in `xs`.

```
defn cum-sum (xs:Seq<Int>) :
   val ys = Vector<Int>()
   var accum = 0
   while not empty?(xs) :
      accum = accum + next(xs)
      add(ys, accum)
   ys
```

Let's try it out on some numbers.

```
defn main () :
   val xs = [1, 1, 3, 1, 5, 6, 2, 3, 8]
   println(cum-sum(to-seq(xs)))

main()
```

Compiling and running the above prints out

```
[1 2 5 6 11 17 19 22 30]
```

Seqable

Notice that in the call to `cum-sum` we have to explicitly convert our tuple into a `Seq` object using `to-seq`. Otherwise Stanza would issue a type error. For convenience, however, it would be better to move the call to `to-seq` inside the body of `cum-sum` and have `cum-sum` accept *any* object that supports `to-seq`.

This brings us to the type `Seqable`. Values of type `Seqable` support only a single operation: calling `to-seq` on a `Seqable` object returns a `Seq`. `Seqable` also accepts a *type parameter* that indicates the type of element it contains. Thus calling `to-seq` on a `Seqable<Int>` returns a `Seq<Int>`.

Let's change our `cum-sum` function to accept an object of type `Seqable<Int>`.

```
defn cum-sum (xs:Seqable<Int>) :
   val xs-seq = to-seq(xs)
   val ys = Vector<Int>()
   var accum = 0
   while not empty?(xs-seq) :
      accum = accum + next(xs-seq)
      add(ys, accum)
   ys
```

Now our `cum-sum` function is general enough to be called with any `Seqable` object. This includes ranges, tuples, arrays, vectors, lists (which we will cover later), and even other sequences. Let's try it out.

```
defn main () :
   val xs = [1, 1, 3, 1, 5, 6, 2, 3, 8]
   val ys = to-array<Int>([1, 1, 3, 1, 5, 6, 2, 3, 8])
   val zs = to-vector<Int>([1, 1, 3, 1, 5, 6, 2, 3, 8])
   val ws = to-list([1, 1, 3, 1, 5, 6, 2, 3, 8])

   println(cum-sum(xs))
   println(cum-sum(ys))
   println(cum-sum(zs))
   println(cum-sum(ws))

main()
```

which prints out

```
[1 2 5 6 11 17 19 22 30]
[1 2 5 6 11 17 19 22 30]
[1 2 5 6 11 17 19 22 30]
[1 2 5 6 11 17 19 22 30]
```

This is the mechanism that allows the core library functions (such as `do`) to operate on all sorts of collections. `do` just accepts a `Seqable` argument.

And since `do` accepts a `Seqable` argument, we can actually rewrite our `cum-sum` function more elegantly using `do`.

```
defn cum-sum (xs:Seqable<Int>) :
   val ys = Vector<Int>()
   var accum = 0
   for x in xs do :
      accum = accum + x
      add(ys, accum)
   ys
```

6.3 Lazy Sequences

Our `cum-sum` function takes a sequence as its argument and returns a vector. This works just fine if we want *all* of the cumulative sums, but what if we want only the first four? Then we're spending a lot of time computing results that we don't need.

To overcome this, we can rewrite `cum-sum` to return a `Seq<Int>` instead of a `Vector<Int>` where the elements in the returned sequence is computed *on-demand*.

```
defn cum-sum (xs:Seqable<Int>) :
   var accum = 0
   val xs-seq = to-seq(xs)
   new Seq<Int> :
      defmethod empty? (this) :
         empty?(xs-seq)
      defmethod peek (this) :
         accum + peek(xs-seq)
      defmethod next (this) :
         accum = peek(this)
         next(xs-seq)
         accum
```

Now `cum-sum` returns a *lazy* sequence where items are computed as they're needed. To demonstrate this, let's call `cum-sum` on an *infinite* range of numbers, and print out the first 10 elements.

```
defn main () :
   val xs = 1 to false by 3
   val ys = cum-sum(xs)
   for i in 0 to 10 do :
      println("Item %_ is %_." % [i, next(ys)])

main()
```

Compiling and running the above gives us

```
Item 0 is 1.
Item 1 is 5.
Item 2 is 12.
Item 3 is 22.
Item 4 is 35.
Item 5 is 51.
Item 6 is 70.
Item 7 is 92.
Item 8 is 117.
Item 9 is 145.
```

Thus `ys` is an *infinite* sequence of integers containing the cumulative sum of another

infinite sequence of integers.

seq

Creating a sequence by calling a function repeatedly on the items from another sequence is a common operation, so it is included in Stanza's core library as the `seq` operating function. The `cum-sum` function can be rewritten using `seq` like this.

```
defn cum-sum (xs:Seqable<Int>) :
   var accum = 0
   for x in xs seq :
      accum = accum + x
      accum
```

6.4 Using The Sequence Library

Now that we've been introduced to sequences, we can unveil the full power of Stanza's core library. As mentioned in an earlier chapter, Stanza encourages users to architect their programs by defining a small set of fundamental operations on each type, and then augment that with a large library of derived operations for those types. Stanza's sequence library is structured in such a way.

The set of fundamental operations for a `Seq` is very small, comprised of just `empty?`, `peek`, and `next`. But Stanza includes a large library of useful functions for manipulating sequences. These functions are roughly categorized into three groups: sequence creators, sequence operators, and sequence reducers. Independently of this categorization, a large number of these functions are also *operating functions* and can be used with the for construct.

Sequence Creators

Sequence creators are functions that take non-`Seq` arguments and create and return `Seq` objects. In typical programming, most sequences you manipulate will have been created with a sequence creator.

to-seq

The most commonly used sequence creator is the `to-seq` function, which works on any `Seqable` object. You've already seen usages of it for converting tuples, arrays, vectors, and ranges to sequences.

repeatedly

`repeatedly` takes an argument function, `f`, and creates an infinite sequence from the results of calling `f` repeatedly. Here is an example of using it to create a sequence containing all the positive powers of 2.

```
var x = 1L
val xs = repeatedly \$ fn () :
   val cur-x = x
   x = x * 2L
   cur-x
```

Let's print out the first 10 elements.

```
do(println{next(xs)}, 0 to 10)
```

which prints out

```
1
2
4
8
16
32
64
128
256
512
```

repeat-while

`repeat-while` takes an argument function, `f`, and creates an infinite sequence by calling `f` repeatedly. `f` must return a `Maybe` object. The returned sequence contains all the wrapped objects in all the `One` objects returned by `f` and ends the first time `f` returns a `None` object.

Here is an example of using it to create a sequence containing all the positive powers of 2 that are less than 2000.

```
var x = 1L
var xs = repeat-while \$ fn () :
   val cur-x = x
   if cur-x < 2000L :
      x = x * 2L
      One(cur-x)
   else :
      None()
```

Let's print out all the items in `xs`.

```
do(println, xs)
```

which prints out

```
1
2
4
8
16
32
64
128
256
512
1024
```

Sequence Operators

Sequence operators are functions that take `Seq` (or `Seqable`) arguments and create and return `Seq` objects. The lazy `cum-sum` function that we implemented is an example of a sequence operator.

cat

One of the simplest sequence operators is the `cat` function which simply concatenates two sequences together to form a longer sequence. Here is an example.

```
val xs = ["Patrick", "Luca", "Emmy"]
val ys = ["Sunny", "Whiskey", "Rummy"]
val zs = cat(xs, ys)
do(println, zs)
```

which prints out

```
Patrick
Luca
Emmy
Sunny
Whiskey
Rummy
```

join

`join` is another simple sequence operator that takes a sequence, `xs`, and a joiner item, `x`, and creates a *lazy* sequence by inserting `x` in between each item in `xs`. Here is an example.

```
val xs = ["Patrick", "Luca", "Emmy"]
val zs = join(xs, "and")
do(println, zs)
```

which prints out

```
Patrick
and
Luca
and
Emmy
```

take-n

The `take-n` function takes an integer, `n`, and a sequence, `xs`, and returns a *lazy* sequence consisting of the first `n` elements in `xs`. It is a fatal error to call `take-n` on a sequence with less than `n` items. Here is an example of using `take-n` to print out the first 10 items in an infinite range.

```
val xs = 0 to false by 13
do(println, take-n(10, xs))
```

which prints out

```
0
13
26
39
52
65
78
91
104
117
```

filter

The `filter` function takes a predicate function, `f`, and a sequence, `xs`, and returns a *lazy* sequence consisting only of the items in `xs` for which calling `f` on them returns `true`. `filter` is also an operating function. Here is an example of using `filter` to print out only the positive items in a sequence.

```
val xs = [1, 3, -2, -7, 3, -8, 9, 10, -3]
val ys = filter({_ > 0}, xs)
do(println, ys)
```

which prints out

```
1
3
3
9
10
```

seq

The `seq` function is the most commonly used sequence operator. It takes a function, `f`, and a sequence, `xs`, and returns a *lazy* sequence comprised of the results of calling `f` on each item in the sequence. Here is an example of printing out the length of each string in a sequence.

```
val xs = ["Patrick", "Luca", "Emmy", "Sunny", "Whiskey", "Rummy"]
val ys = seq(length, xs)
do(println, ys)
```

which prints out

```
7
4
4
5
7
5
```

Sequence Reducers

Sequence reducers are functions that take `Seq` (or `Seqable`) arguments and return non-`Seq` objects.

We have already been introduced to and have been using a number of these, such as `do`, `find`, `first`, `index-when`, `all?`, `none?`, and `any?`. We'll take this opportunity to say that they each accept any `Seqable` object as their argument.

We'll show you a handful more useful reducers here, but you are encouraged to read the reference documentation for a listing of all of them.

contains?

`contains?` takes a sequence, `xs`, and an item, `y`, and returns `true` if `xs` contains `y`. Otherwise it returns `false`.

```
val xs = ["Patrick", "Luca", "Emmy"]
println(contains?(xs, "Emmy"))
println(contains?(xs, "Emily"))
```

prints out

```
true
false
```

index-of

`index-of` takes a sequence, `xs`, and an item, `y`, and returns the index of the first occurrence of `y` in `xs`. If `y` never appears in `xs`, then `false` is returned.

```
val xs = ["Patrick", "Luca", "Emmy"]
println(index-of(xs, "Emmy"))
println(index-of(xs, "Emily"))
```

prints out

```
2
false
```

unique

`unique` takes a sequence, `xs`, and returns a *list* containing all the items in `xs` but with duplicates removed.

```
val xs = ["Patrick", "Luca", "Luca", "Emmy", "Patrick", "Emmy"]
println(unique(xs))
```

prints out

```
("Patrick" "Luca" "Emmy")
```

to-array

`to-array` creates a new array containing all the items in its given sequence. It takes a single *type argument* to indicate the element type of the array. We will discuss type arguments when we introduce parametric polymorphism. Here is an example.

```
val xs = ["Patrick", "Luca", "Emmy"]
println(to-array<String>(xs))
```

prints out

```
["Patrick" "Luca" "Emmy"]
```

to-vector

`to-vector` creates a new vector containing all the items in its given sequence. Like `to-array`, it also takes a single type argument to indicate the element type of the vector. Here is an example.

```
val xs = ["Patrick", "Luca", "Emmy"]
println(to-vector<String>(xs))
```

to-list

`to-list` creates a new *list* containing all the items in its given sequence. Note that unlike `to-array` and `to-vector`, `to-list` does *not* take a type argument. We will cover lists in more detail when we talk about programming with immutable datastructures. For now, you can treat them just as another type of collection. And we will explain why `to-list` does not require a type argument in the chapter on parametric polymorphism. Here is an example.

```
val xs = ["Patrick", "Luca", "Emmy"]
println(to-list(xs))
```

which prints out

```
("Patrick" "Luca" "Emmy")
```

reduce

`reduce` takes a binary operator, `f`, an initial item, `x0`, and a sequence, `xs`. If `xs` is empty then `reduce` returns `x0`. If `xs` contains one item, then `reduce` returns the result of calling `f` on `x0` and the item in `xs`. If `xs` contains two items, then `reduce` returns the result of calling `f` on `x0` and the first item in `xs`, and then calling `f` again on that result and the second item in `xs`. If `xs` contains three items, then `reduce` returns the result of calling `f` on `x0` and the first item in `xs`, then calling `f` again on that result and the second item in `xs`, and then calling `f` again on that result and the third item in `xs`. Et cetera.

Here is an example of using the `bit-or` operator to compute the bitwise or of every integer in a tuple.

```
val xs = [1, 5, 18, 92, 1, 3]
val y = reduce(bit-or, 0, xs)
println(y)
```

which prints out

```
95
```

6.5 Collection versus Seqable

Consider the following definition of `print-odd-then-even`, a function that first prints all the odd integers in a sequence, and then prints all the even integers in the sequence.

```
defn print-odd-then-even (xs:Seqable<Int>) :
   val odd = filter({_ % 2 != 0}, xs)
   val even = filter({_ % 2 == 0}, xs)
   print("Odd integers: ")
   println-all(join(odd, ", "))
   print("Even integers: ")
   println-all(join(even, ", "))
```

Because we declared `print-odd-then-even` to accept an argument of type `Seqable`, we are able to call it on a variety of different types of collections. Let's try a few.

```
defn main () :
   val xs = [1, 2, 3, 4, 5, 6, 7, 8]
   val ys = to-array<Int>([1, 2, 3, 4, 5, 6, 7, 8])
   val zs = to-vector<Int>([1, 2, 3, 4, 5, 6, 7, 8])
   val ws = 1 through 8

   println("On tuples")
   print-odd-then-even(xs)

   println("On arrays")
   print-odd-then-even(ys)

   println("On vectors")
   print-odd-then-even(zs)

   println("On ranges")
   print-odd-then-even(ws)

main()
```

It prints out

```
On tuples
Odd integers: 1, 3, 5, 7
Even integers: 2, 4, 6, 8
On arrays
Odd integers: 1, 3, 5, 7
Even integers: 2, 4, 6, 8
On vectors
Odd integers: 1, 3, 5, 7
Even integers: 2, 4, 6, 8
On ranges
Odd integers: 1, 3, 5, 7
Even integers: 2, 4, 6, 8
```

demonstrating that it does the same thing regardless of the type of collection.

But now let's try calling it on a `Seq`. All `Seq` objects are also trivially instances of `Seqable`. Calling `to-seq` on a `Seq` object simply returns itself.

```
defn main2 () :
   val xs = to-seq(1 through 8)
   println("On seqs")
   print-odd-then-even(xs)

main2()
```

This print outs

```
On seqs
Odd integers: 1, 3, 5, 7
Even integers:
```

What is happening? How come the even integers didn't get printed out?

The problem lies in the two calls to `filter`.

```
val odd = filter({_ % 2 != 0}, xs)
val even = filter({_ % 2 == 0}, xs)
```

`filter` creates a lazy sequence, so iterating over the result of `filter` also requires iterating over the sequence it was constructed from. Thus printing out `odd` also requires iterating over `xs`, in which case, after printing out all the odd integers, we will have iterated through `xs` once completely and it will now be empty. At this point, `even` is also empty, as the sequence it was constructed from is now empty.

The fundamental problem is that `Seq` is a subtype of `Seqable`. Calling `to-seq` twice on a `Seq` object does *not* return two independent sequences. For these purposes, Stanza provides a subtype of `Seqable` called `Collection`. Identical to `Seqable`, `to-seq` is the only fundamental operation supported by `Collection`. The crucial difference is that `Seq` is not a subtype of `Collection`. This means that each call to `to-seq` on a `Collection` returns an independent sequence.

Let's rewrite our `print-odd-then-even` function with the appropriate type annotation.

```
defn print-odd-then-even (xs:Collection<Int>) :
   val odd = filter({_ % 2 != 0}, xs)
   val even = filter({_ % 2 == 0}, xs)
   print("Odd integers: ")
   println-all(join(odd, ", "))
   print("Even integers: ")
   println-all(join(even, ", "))
```

You may verify that calling `print-odd-then-even` with all the collections in `main` still behaves as before. The important point is that attempting to compile `main2` now gives this error.

```
Cannot call function print-odd-then-even of type Collection<Int> -> False
with arguments of type (Seq<Int>).
```

With the appropriate type annotation, Stanza now prevents us from calling `print-odd-then-even` incorrectly.

As a rule of thumb, you should always write your functions to accept `Collection` objects by default. If you are sure that you iterate through the sequence only once, then you may change it to accept `Seqable` objects in order to be able to pass it a `Seq`.

6.6 Revisiting Stack

Let us now revisit our `Stack` type from chapter 4. Here is a (slightly cleaned up) listing of its definitions.

```
deftype Stack
defmulti push (s:Stack, x:String) -> False
defmulti pop (s:Stack) -> String
defmulti empty? (s:Stack) -> True|False

defn Stack (capacity:Int) -> Stack :
   val items = Array<String>(capacity)
   var size = 0
   new Stack :
      defmethod push (this, x:String) :
         if size == capacity : fatal("Stack is full!")
         items[size] = x
         size = size + 1
      defmethod pop (this) :
         if size == 0 : fatal("Stack is empty!")
         size = size - 1
         items[size]
```

```
      defmethod empty? (this) :
         size == 0
      defmethod print (o:OutputStream, this) :
         print(o, "Stack containing [")
         print-all(o, join(take-n(size, items), " "))
         print(o, "]")
```

It's quite a basic definition, allowing us to push and pop items but not much else. We cannot find the index of a specific item, or determine whether it contains any capitalized strings, or get a listing of all of its unique elements. We cannot even iterate through it. The following

```
val s = Stack(10)
push(s, "Timon")
push(s, "and")
push(s, "Pumbaa")

for x in s do :
   println(x)
```

gives us this error if we try to compile it.

```
No appropriate function do for arguments
of type (? -> False, Stack). Possibilities are:
   do: <?T> . (T -> ?, Seqable<?T>) -> False
   do: <?T, ?S> . ((T, S) -> ?,
                   Seqable<?T>,
                   Seqable<?S>) -> False
   do: <?T, ?S, ?U> . ((T, S, U) -> ?, Seqable<?T>,
                       Seqable<?S>,
                       Seqable<?U>) -> False
```

It says that there are multiple definitions of `do` but all of them require a `Seqable` argument, and `Stack` is not a `Seqable`.

We shall extend the functionality of `Stack` by declaring it as a subtype of `Collection`.

```
deftype Stack <: Collection<String>
```

The mandatory minimal implementation of `Collection` is `to-seq`, so we need to now provide a method for it. Here is now our extended `Stack` construction function.

```
defn Stack (capacity:Int) -> Stack :
   val items = Array<String>(capacity)
   var size = 0
   new Stack :
      defmethod push (this, x:String) :
         if size == capacity : fatal("Stack is full!")
         items[size] = x
```

```
      size = size + 1
   defmethod pop (this) :
      if size == 0 : fatal("Stack is empty!")
      size = size - 1
      items[size]
   defmethod empty? (this) :
      size == 0
   defmethod print (o:OutputStream, this) :
      print(o, "Stack containing [")
      print-all(o, join(this, " "))
      print(o, "]")
   defmethod to-seq (this) :
      take-n(size, items)
```

Our implementation of `to-seq` simply calls `take-n` to retrieve the first `size` elements from the backing array `items`.

Now let's try exercising the power of our new extended `Stack` type.

```
defn main () :
   val s = Stack(10)
   for x in ["Timon", "Timon", "and", "Pumbaa", "Pumbaa"] do :
      push(s, x)

   println("1. Contents of s")
   println(s)

   println("\n2. Index of Pumbaa")
   println(index-of(s, "Pumbaa"))

   println("\n3. Does it contain any capitalized strings?")
   println(any?(upper-case?{_[0]}, s))

   println("\n4. Are all strings capitalized?")
   println(all?(upper-case?{_[0]}, s))

   println("\n5. What are the capitalized strings?")
   val cap-s = filter(upper-case?{_[0]}, s)
   println-all(join(cap-s, ", "))

   println("\n6. What are its unique elements?")
   println(unique(s))

main()
```

Compiling and running the above prints out

```
1. Contents of s
Stack containing [Timon Timon and Pumbaa Pumbaa]

2. Index of Pumbaa
3

3. Does it contain any capitalized strings?
true

4. Are all strings capitalized?
false

5. What are the capitalized strings?
Timon, Timon, Pumbaa, Pumbaa

6. What are its unique elements?
("Timon" "and" "Pumbaa")
```

This example shows us the full advantage of structuring your programs to contain a large library of derived operations. With a two line change to our definition of the `Stack` object, we've provided it the full capabilities of Stanza's sequence library.

Chapter 7

Programming with Immutable Datastructures

An immutable datastructure is one that cannot be *changed* after it has been created. Some examples you've already seen are strings, tuples, numbers, and true and false. In constrast, a *mutable* datastructure is one that can be changed after it has been created. Some examples are arrays, vectors, and sequences.

If something is guaranteed not to change, then there are two details that you no longer have to worry about.

1. You don't have to care about *which* object it is. There is no difference between the value `42` and the value `20 + 22`. They are the same value. You can replace every occurrence of `42` in your program with `20 + 22` and it will still behave the same way. Similarly, you can replace every occurrence of

   ```
   "Timon and Pumbaa"
   ```

   in your program with

   ```
   append("Timon", " and Pumbaa")
   ```

   without changing its behaviour.

   In contrast, consider the following call for adding a number to a vector.

   ```
   add(xs, 42)
   ```

   Now you *do* need to pay very close attention to which vector `xs` is referring to. It would be an error to add 42 to the wrong vector.

2. You don't have to think about *when* to do something to an object. Consider the following code for popping an item from the vector `xs` and then adding two new items to it.

   ```
   pop(xs)
   add(xs, 42)
   ```

```
add(xs, 43)
```

The ordering of those expressions are critically important. Every possible ordering of those three expressions results in a different behaviour. Notice that this sort of thinking is never done with strings, tuples, or numbers; simply because there's nothing than *can* be done to them except to create new objects out of them.

7.1 Lists

A `List` object represents a singly linked list of objects. A list is Stanza's most basic immutable datastructure and cannot be changed once created. Here is how to create an empty list.

```
List()
```

Here is how to create a list containing a single item.

```
List(42)
```

Here is how to create a list containing two items.

```
List(42, "Timon")
```

This works for lists containing up to four items. For creating lists containing more than four items, you may use the `to-list` function to convert sequences into lists.

```
to-list([1, 2, 3, 4, 5, "Timon", "and", "Pumbaa"])
```

You may also use `cons` (short for construct) to create a new list by tacking a new item to the beginning of an existing list.

```
val xs = List(1, 2, 3)
val ys = cons(42, xs)
```

`cons` allows you to tack on up to three items.

```
val xs = List(1, 2, 3)
val ys0 = cons(42, xs)
val ys1 = cons(42, 43, xs)
val ys2 = cons(42, 43, 44, xs)
```

To append more than three items to the beginning of another list, use the `append` function.

```
val xs = List(1, 2, 3)
val ys = append([42, 43, 44, 45, 46, 47], xs)
println(ys)
```

Compiling and running the above prints out

```
(42 43 44 45 46 47 1 2 3)
```

Fundamental Operations

A list is defined by three fundamental operations.

1. You can check whether the list is empty.
2. You can retrieve the first element in the list.
3. You can retrieve a list containing all the elements after the first one.

Assuming that `xs` is a list, here is how to check whether `xs` is empty.

```
empty?(xs)
```

Here is how to retrieve the first element in `xs`.

```
head(xs)
```

And here is how to retrieve all the elements after the first one, as another list.

```
tail(xs)
```

7.2 Example: Coin Counting

Suppose you have access to pennies, nickels, dimes, quarters, and *loonies*, and the poutine you bought costs $1.17. (Loonies are Canadian coins worth 100 cents each.) How may different combinations of coins are there that total up to $1.17?

Here is our algorithm for calculating it. `num-coin-combos` takes two arguments: `cents`, which represents the amount of money you wish to make represented in cents, and `coins`, a list of the cent values of the coins you can use.

```
defn num-coin-combos (cents:Int, coins:List<Int>) -> Int :
   if cents == 0 :
      1
   else if cents < 0 :
      0
   else if empty?(coins) :
      0
   else :
      val with-first-coin = num-coin-combos(cents - head(coins), coins)
      val without-first-coin = num-coin-combos(cents, tail(coins))
      with-first-coin + without-first-coin
```

Let's read through each case of the algorithm one by one. The first case is

```
if cents == 0 :
   1
```

There is only one way to make 0 cents, and that is to not use any coins at all. Makes sense. The second case is

```
else if cents < 0 :
   0
```

There is no way to make a negative cent value. Makes sense. The third case is

```
else if empty?(coins) :
   0
```

If we're not allowed to use *any* kind of coin, then there's also no way to make our total. Makes sense as well. The real work of the algorithm is done by the fourth case.

```
val with-first-coin = num-coin-combos(cents - head(coins), coins)
val without-first-coin = num-coin-combos(cents, tail(coins))
with-first-coin + without-first-coin
```

Consider the next type of coin in our list. Suppose it's a loonie. There are two choices we can now make.

1. We can account for 100 cents by using the loonie, and count the number of ways to make `cents - 100`. This is calculated as `with-first-coin`.
2. We can choose not to use the loonie, and count the number of ways to make `cents` without using loonies. This is calculated as `without-first-coin`.

The total number of combinations is the sum of the results of the two possible choices we can make.

Let's now use our `num-coin-combos` function to answer the original question.

```
defn main () :
   val coins = [100, 25, 10, 5, 1]
   val num-combos = num-coin-combos(117, to-list(coins))
   println("There are %_ coin combinations that total to 117 cents." %
           [num-combos])

main()
```

which prints out

```
There are 349 coin combinations that total to 117 cents.
```

Strange Lands

Suppose we find ourselves in strange lands with a strange currency. The currency is made up of *buzzles*, with a value of 57 cents, *moozles* (26 cents), *foogs* (10 cents), *goofs* (5 cents), and *tents* (3 cents). Now how many ways are there to make the $1.17 needed to buy poutine? (Though the currency may be strange, poutine is fairly universal).

Let's adapt our `main` function to calculate with the new currency.

```
defn main () :
   val coins = [57, 26, 10, 5, 3]
   val num-combos = num-coin-combos(117, to-list(coins))
   println("There are %_ coin combinations that total to 117 cents." %
           [num-combos])

main()
```

which prints out

```
There are 137 coin combinations that total to 117 cents.
```

indicating that buzzles and foogs are a little less flexible than Canadian currency.

SICP

This exercise is adapted from the best book on computer science ever written, *The Structure and Interpretation of Computer Programs* by Abelson and Sussman. I highly recommend it to anyone interested in the deep connections between languages and computation. And since Stanza is a (highly modified) Scheme dialect at heart, all the exercises can easily be done in Stanza as well.

7.3 List Library

`List` is a subtype of `Collection` and so all of Stanza's sequence library also works on lists. The core library also includes a few functions specifically for managing lists. You've been introduced to a few of them already: `head`, `tail`, `append`, `cons`. Here's a few more.

get

The `get` function allows you to retrieve the element at a specific index in a list.

```
val xs = to-list(0 to 1000 by 3)
get(xs, 11)
```

Using Stanza's built-in operator, the above could also be written as

```
val xs = to-list(0 to 1000 by 3)
xs[11]
```

headn

`headn` returns a list containing the first n items in a list.

```
val xs = to-list(0 to 1000 by 3)
headn(xs, 10)
```

tailn

`tailn` returns a new list containing the items following the first n items in a list.

```
val xs = to-list(0 to 1000 by 3)
tailn(xs, 10)
```

reverse

`reverse` takes an argument list and returns a new list containing the same items in reversed order.

```
val xs = to-list(0 to 1000 by 3)
reverse(xs)
```

last

`last` takes an argument list and returns the last item in it. The list must not be empty.

```
val xs = to-list(0 to 1000 by 3)
last(xs)
```

but-last

`but-last` takes an argument list and returns a new list containing all the items from the argument list except the last one.

```
val xs = to-list(0 to 1000 by 3)
but-last(xs)
```

map

`map` is the most commonly used function on lists. It takes an argument function, `f`, and a list, `xs`, and returns a new list containing the results of calling `f` on each item.

Here is an example that calculates the lengths of all the strings in the list `xs`.

```
val xs = to-list(["Timon" "and" "Pumbaa"])
val lengths = map(length, xs)
```

`map` is also an operating function, and it can be used together with the for construct. Here is an example of doubling every integer in the list `xs`.

```
val xs = to-list(0 to 1000 by 3)
val doubled = for x in xs map :
   x * 2
```

7.4 Example: More Coin Counting

One limitation of our previous algorithm for coin counting is that it calculated the number of ways we can make a certain total, but it never told us what these combinations actually were. You may be (as I was) actually quite curious about how to make $1.17 using buzzles and foogs.

Let's write a function called `coin-combos` that does that. Like `num-coin-combos`, `coin-combos` takes two arguments: `cents`, which represents the number of cents you wish to make, and `coins`, a list of the cent values of the coins. The difference is that `coin-combos` returns a list of *combinations*. Each combination is a list containing the number of times each coin is used.

```
defn coin-combos (cents:Int, coins:List<Int>) -> List<List<Int>> :
   if cents == 0 :
      List(map({0}, coins))
   else if cents < 0 :
      List()
   else if empty?(coins) :
      List()
   else :
      defn head+1 (xs:List<Int>) : cons(head(xs) + 1, tail(xs))
      defn cons-0 (xs:List<Int>) : cons(0, xs)
      val with-first-coin = map(head+1, coin-combos(cents - head(coins), coins))
      val without-first-coin = map(cons-0, coin-combos(cents, tail(coins)))
      append(with-first-coin, without-first-coin)
```

Let's examine each case separately.

```
if cents == 0 :
   List(map({0}, coins))
```

There is only one way to make up 0 cents, and that is by using no coins at all. So return a list with a single combination indicating that each coin is used 0 times.

```
else if cents < 0 :
   List()
```

There is no way to make a negative total so return an empty list.

```
else if empty?(coins) :
   List()
```

If we're not allowed to use *any* kind of coin, then there's also no way to make our total. Return an empty list. And finally, we're at the last case again.

```
val with-first-coin = map(head+1, coin-combos(cents - head(coins), coins))
val without-first-coin = map(cons-0, coin-combos(cents, tail(coins)))
append(with-first-coin, without-first-coin)
```

All the combinations resulting from choosing to use the first coin are computed in `with-first-coin`. And all the combinations resulting from choosing not to use the first coin are computed in `without-first-coin`. We then append both lists to get the complete list of combinations.

The fourth case relies upon two helper functions, `head+1`, which adds 1 to the head of a list, and `cons-0`, which tacks 0 on to the beginning of a list.

```
defn head+1 (xs:List<Int>) : cons(head(xs) + 1, tail(xs))
defn cons-0 (xs:List<Int>) : cons(0, xs)
```

Let's now update our `main` function to report all the different ways we can use buzzles and foogs to make $1.17. Recall that buzzles are worth 57 cents, moozles are 26 cents, foogs are 10 cents, goofs are 5 cents, and tents are 3 cents.

```
defn main () :
   val coins = [57, 26, 10, 5, 3]
   val combos = coin-combos(117, to-list(coins))
   println("There are %_ coin combinations that total to 117 cents." % [
      length(combos)])
   do(println, combos)

main()
```

Compiling and running the above prints out

```
There are 137 coin combinations that total to 117 cents.
(2 0 0 0 1)
(1 2 0 1 1)
(1 1 2 1 3)
(1 1 1 3 3)
...
(0 0 0 6 29)
(0 0 0 3 34)
(0 0 0 0 39)
```

Thus we can pay for our $1.17 poutine using two buzzles and a foog. Or if we don't mind holding up the line, we can hunt around for thirty nine tents.

Readable Combos

For the sake of readability, let's write a printing function for formatting the combinations in a readable way. `print-combo` takes as arguments a combination, `combo`, and a collection representing the names of the coins, `names`.

```
defn print-combo (combo:List<Int>, names:Collection<String>) :
   val parts = for (c in combo, n in names) seq? :
      if c == 0 : None()
      else if c == 1 : One("%_ %_" % [c, n])
      else : One("%_ %_s" % [c, n])
   println-all(join(parts, ", "))
```

You are encouraged to read the reference documentation for a description of what `seq?` does. You should be able to understand it now.

Now update the final call to `print` in the `main` function.

```
val coin-names = ["buzzle", "moozle", "foog", "goof", "tent"]
do(print-combo{_, coin-names}, combos)
```

Compiling and running the program now prints out

```
There are 137 coin combinations that total to 117 cents.
2 buzzles, 1 tent
1 buzzle, 2 moozles, 1 goof, 1 tent
1 buzzle, 1 moozle, 2 foogs, 1 goof, 3 tents
1 buzzle, 1 moozle, 1 foog, 3 goofs, 3 tents
...
15 goofs, 14 tents
12 goofs, 19 tents
9 goofs, 24 tents
6 goofs, 29 tents
3 goofs, 34 tents
39 tents
```

7.5 Extended Example: Automatic Differentiation

In your own programming, you are encouraged to define and use immutable datastructures when possible. Uses of mutation and stateful objects should serve a clear purpose. In this example, we define an immutable datastructure for manipulating algebra expressions and write a function for automatically differentiating expressions.

Symbols

`Symbol` objects are used to represent a unique constant object in Stanza. For example, the following creates and assigns symbols to `x` and to `y`.

```
val x = ‘Timon
val y = ‘Pumbaa
```

Symbols are created by prefixing an identifier with the backtick (‘) operator. Very little can be done with symbols except check whether it is equal to another symbol. The following

```
println(x == ‘Timon)
println(y == ‘Timon)
```

prints out

```
true
false
```

and represents the most common use case for symbols. In this respect they are used in much the same way as enumerated constants in other languages. We will use symbols to represent the name of variables in our algebraic expressions.

The Expression Datastructure

We will first declare a type, `Exp`, to refer to an algebraic expression.

```
deftype Exp
defstruct Const <: Exp : (value:Int)
defstruct Variable <: Exp : (name:Symbol)
defstruct Add <: Exp : (a:Exp, b:Exp)
defstruct Subtract <: Exp : (a:Exp, b:Exp)
defstruct Multiply <: Exp : (a:Exp, b:Exp)
defstruct Divide <: Exp : (a:Exp, b:Exp)
defstruct Power <: Exp : (a:Exp, b:Exp)
defstruct Log <: Exp : (a:Exp)
```

A handful of different types of expressions are supported. `Const` represents constant integer literals, `Variable` represents a named variable, and the standard arithmetic operators are represented by `Add`, `Subtract`, `Multiply`, and `Divide`. `Power` represents one expression raised to the power of another, and `Log` represents the natural logarithm of an expression. Notice that many of the expressions contain fields that are themselves types of `Exp`. So the type `Exp` contains fields of type `Exp`. We call such a type a *recursive* type.

Printing an Expression

As usual, we will provide a custom `print` method for the `Exp` type to allow us to print it out.

```
defmethod print (o:OutputStream, e:Exp) :
   defn print-operator (a:Exp, op:String, b:Exp) :
      print(o, a)
      print(o, op)
      print(o, b)
   match(e) :
      (e:Const) : print(o, value(e))
      (e:Variable) : print(o, name(e))
      (e:Log) : print(o, "ln(%_)" % [a(e)])
      (e:Add) : print-operator(a(e), " + ", b(e))
      (e:Subtract) : print-operator(a(e), " - ", b(e))
      (e:Multiply) : print-operator(a(e), " * ", b(e))
      (e:Divide) : print-operator(a(e), " / ", b(e))
      (e:Power) : print-operator(a(e), " ^ ", b(e))
```

Let's now create an expression and print it out. The expression we will create is

```
2 * x ^ 2 + (1 + 3) * x + ln(x + 4)
```

Here is our `main` function.

```
defn main () :
   val term1 = Multiply(Const(2), Power(Variable(‘x), Const(2)))
   val term2 = Multiply(Add(Const(1), Const(3)), Variable(‘x))
   val term3 = Log(Add(Variable(‘x), Const(4)))
   val exp = Add(Add(term1, term2), term3)
   println(exp)

main()
```

Compiling and running the above prints out

```
2 * x ^ 2 + 1 + 3 * x + ln(x + 4)
```

We're off to a great start!

Handling Precedence

Our printing method for expressions is close, but it doesn't handle precedences correctly. The `1 + 3` in the printed expression should be surrounded by parentheses. Otherwise the meaning is different than intended.

Let's add a mechanism to handle precedences properly. Here's the basic algorithm. Every type of expression is associated with a number that represents its precedence. `Const`, `Log`, and `Variable` expressions have the highest precedence 3. `Power` has precedence 2. `Multiply` and `Divide` have precedence 1. And `Add` and `Subtract` have the lowest precedence 0.

```
defn precedence (e:Exp) :
   match(e) :
      (e:Add|Subtract) : 0
      (e:Multiply|Divide|Power) : 1
      (e:Power) : 2
      (e:Const|Variable|Log) : 3
```

The basic rule is that if a lower precedence expression appears as a child of a higher precedence expression, then the lower precedence expression needs to be surrounded by parentheses when printed out. So we'll define a new nested function within `print` to help us print nested expressions in the context of expression `e`.

```
defn print-nested (ne:Exp) :
   if precedence(ne) < precedence(e) :
      print(o, "(%_)" % [ne])
   else :
      print(o, ne)
```

If the nested expression `ne` has lower precedence than `e`, then `ne` is printed with surrounding parentheses. Otherwise `ne` is just printed directly.

The `print-operator` function also needs to be updated to call `print-nested`.

```
defn print-operator (a:Exp, op:String, b:Exp) :
   print-nested(a)
   print(o, op)
   print-nested(b)
```

Those are all the changes needed to handle precedence. Here is the full `print` method.

```
defmethod print (o:OutputStream, e:Exp) :
   defn print-nested (ne:Exp) :
      if precedence(ne) < precedence(e) :
         print(o, "(%_)" % [ne])
      else :
         print(o, ne)
   defn print-operator (a:Exp, op:String, b:Exp) :
      print-nested(a)
      print(o, op)
      print-nested(b)
   match(e) :
      (e:Const) : print(o, value(e))
      (e:Variable) : print(o, name(e))
```

```
    (e:Log) : print(o, "ln(%_)" % [a(e)])
    (e:Add) : print-operator(a(e), " + ", b(e))
    (e:Subtract) : print-operator(a(e), " - ", b(e))
    (e:Multiply) : print-operator(a(e), " * ", b(e))
    (e:Divide) : print-operator(a(e), " / ", b(e))
    (e:Power) : print-operator(a(e), " ^ ", b(e))
```

If you compile and run the program again, it should now correctly print out

```
2 * x ^ 2 + (1 + 3) * x + ln(x + 4)
```

Operator Overloading

The code we used to construct the expression

```
val term1 = Multiply(Const(2), Power(Variable(‘x), Const(2)))
val term2 = Multiply(Add(Const(1), Const(3)), Variable(‘x))
val term3 = Log(Add(Variable(‘x), Const(4)))
val exp = Add(Add(term1, term2), term3)
```

is quite verbose. Let's overload some operators to help us with that.

Recall that the operators `+`, `-`, `*`, `/`, and `^` are just syntactic shorthands for calling the functions `plus`, `minus`, `times`, `divide`, and `bit-xor`. Thus all we need to do is define those functions on `Exp` objects.

```
defn plus (a:Exp, b:Exp) : Add(a, b)
defn minus (a:Exp, b:Exp) : Subtract(a, b)
defn times (a:Exp, b:Exp) : Multiply(a, b)
defn divide (a:Exp, b:Exp) : Divide(a, b)
defn bit-xor (a:Exp, b:Exp) : Power(a, b)
defn ln (a:Exp) : Log(a)
```

Now let's rewrite our `main` function using the new operators.

```
defn main () :
   val x = Variable(‘x)
   val [c1, c2, c3, c4] = [Const(1), Const(2), Const(3), Const(4)]
   val exp = c2 * x ^ c2 + (c1 + c3) * x + ln(x + c4)
   println(exp)
```

Much better! If we overlook the little c's in front of each constant it's essentially identical to our printed expression.

The Differentiation Algorithm

Now we can implement the differentiation algorithm! The function `differentiate` takes two arguments: the expression to differentiate, `e`, and the variable with respect to which it will differentiate, `x`.

The actual formulas used to do the differentiation are standard, and we won't explain how to derive them. If you have taken a course on calculus, you can break open your old textbook and copy the formulas here. If you haven't taken a course on calculus, then armed with this program, you'll never have to manually differentiate again.

```
defn differentiate (e:Exp, x:Symbol) -> Exp :
   defn ddx (e:Exp) : differentiate(e, x)

   match(e) :
      (e:Const) :
         Const(0)
      (e:Variable) :
         if name(e) == x : Const(1)
         else : Const(0)
      (e:Add) :
         ddx(a(e)) + ddx(b(e))
      (e:Subtract) :
         ddx(a(e)) - ddx(b(e))
      (e:Multiply) :
         a(e) * ddx(b(e)) + b(e) * ddx(a(e))
      (e:Divide) :
         val num = b(e) * ddx(a(e)) - a(e) * ddx(b(e))
         val den = b(e) ^ Const(2)
         num / den
      (e:Power) :
         e * (b(e) * ddx(a(e)) / a(e) + ln(a(e)) * ddx(b(e)))
      (e:Log) :
         ddx(a(e)) / a(e)
```

Let's try differentiating our example expression now.

```
defn main () :
   val x = Variable(`x)
   val [c1, c2, c3, c4] = [Const(1), Const(2), Const(3), Const(4)]
   val exp = c2 * x ^ c2 + (c1 + c3) * x + ln(x + c4)
   val dexp = differentiate(exp, `x)

   println("Original Expression: %_" % [exp])
   println("Differentiated Expression: %_" % [dexp])
```

Compiling and running the program prints out

```
Original Expression: 2 * x ^ 2 + (1 + 3) * x + ln(x + 4)
Differentiated Expression: 2 * x ^ 2 * (2 * 1 / x + ln(x) * 0) +
                           x ^ 2 * 0 + (1 + 3) * 1 +
                           x * (0 + 0) + (1 + 0) / (x + 4)
```

If you check the result, it *does* work! The only problem is that the result contains a lot of expressions that can be trivially simplified. We'll fix that later. But this isn't bad at all for a 22-line algorithm.

Simplification

The only thing left to do now is simplify the resulting expression. We will write a *very* simple simplifier that simply looks for patterns like adding an expression to zero, or dividing by one, et cetera. But before we introduce the simplification algorithm, we need to first write a very useful helper function.

```
defn map (f: Exp -> Exp, e:Exp) -> Exp :
   match(e) :
      (e:Add) : Add(f(a(e)), f(b(e)))
      (e:Subtract) : Subtract(f(a(e)), f(b(e)))
      (e:Multiply) : Multiply(f(a(e)), f(b(e)))
      (e:Divide) : Divide(f(a(e)), f(b(e)))
      (e:Power) : Power(f(a(e)), f(b(e)))
      (e:Log) : Log(f(a(e)))
      (e) : e
```

`map` takes an argument function, `f`, and an expression, `e`, and returns a new expression resulting from calling `f` on every subexpression in `e`. Its behaviour is analogous to the `map` function for lists. Calling `map` on a list *maps* `f` onto every element in the list. Similarly, calling `map` on an expression *maps* `f` onto every subexpression in the expression.

We're now ready to write the `simplify` function. It takes an expression as its argument, and returns a simplified version of the expression by replacing specific patterns with simpler expressions.

```
defn simplify (e:Exp) :
   defn const? (e:Exp, v:Int) :
      match(e) :
         (e:Const) : value(e) == v
         (e) : false
   defn one? (e:Exp) : const?(e, 1)
   defn zero? (e:Exp) : const?(e, 0)

   match(map(simplify, e)) :
      (e:Add) :
         if zero?(a(e)) : b(e)
```

```
      else if zero?(b(e)) : a(e)
      else : e
   (e:Subtract) :
      if zero?(a(e)) : Const(-1) * b(e)
      else if zero?(b(e)) : a(e)
      else : e
   (e:Multiply) :
      if one?(a(e)) : b(e)
      else if one?(b(e)) : a(e)
      else if zero?(a(e)) or zero?(b(e)) : Const(0)
      else : e
   (e:Divide) :
      if zero?(a(e)) : Const(0)
      else if one?(b(e)) : a(e)
      else : e
   (e:Power) :
      if one?(a(e)) : Const(1)
      else if zero?(b(e)) : Const(1)
      else : e
   (e:Log) :
      if one?(a(e)) : Const(0)
      else : e
   (e) : e
```

Most of the work of the simplifier is done in the branches of the match expression; you can read through them to understand which patterns are being simplified and what they're being simplified to. However, the most magical part of the function is the call to `map`.

```
match(map(simplify, e)) :
   (e:Add) :
   ...
```

In English, that pattern says: first simplify all the nested subexpressions in `e` *and then* look for these patterns and replace them with simpler ones.

Let's update our `main` function now to simplify the differentiated expression.

```
defn main () :
   val x = Variable(`x)
   val [c1, c2, c3, c4] = [Const(1), Const(2), Const(3), Const(4)]
   val exp = c2 * x ^ c2 + (c1 + c3) * x + ln(x + c4)
   val dexp = differentiate(exp, `x)
   val sexp = simplify(dexp)

   println("Original Expression: %_" % [exp])
   println("Differentiated Expression: %_" % [dexp])
   println("Simplified Expression: %_" % [sexp])
```

When compiled and ran it prints out

```
Original Expression: 2 * x ^ 2 + (1 + 3) * x + ln(x + 4)
Differentiated Expression: 2 * x ^ 2 * (2 * 1 / x + ln(x) * 0) +
                           x ^ 2 * 0 + (1 + 3) * 1 +
                           x * (0 + 0) + (1 + 0) / (x + 4)
Simplified Expression: 2 * x ^ 2 * 2 / x + 1 + 3 + 1 / (x + 4)
```

The simplified expression is *much* cleaner now! This concludes our automatic differentiation example. The simplicity of both the differentiation and the simplification algorithm stems from the fact that `Exp` is an immutable datastructure. In fact, the programming language Lisp, which strongly emphasized computation with immutable list structures and also heavily influenced the design of Stanza, was invented in part for writing computer algebra systems. John McCarthy started writing differentiation algorithms in Lisp even before the language was running!

Program Listing

Here's a full program listing of the example.

```
defpackage calculus :
   import core

;Expression definition
deftype Exp
defstruct Const <: Exp : (value:Int)
defstruct Variable <: Exp : (name:Symbol)
defstruct Add <: Exp : (a:Exp, b:Exp)
defstruct Subtract <: Exp : (a:Exp, b:Exp)
defstruct Multiply <: Exp : (a:Exp, b:Exp)
defstruct Divide <: Exp : (a:Exp, b:Exp)
defstruct Power <: Exp : (a:Exp, b:Exp)
defstruct Log <: Exp : (a:Exp)

;Precedences
defn precedence (e:Exp) :
   match(e) :
      (e:Add|Subtract) : 0
      (e:Multiply|Divide|Power) : 1
      (e:Power) : 2
      (e:Const|Variable|Log) : 3

;Print behaviour for expressions
defmethod print (o:OutputStream, e:Exp) :
   defn print-nested (ne:Exp) :
```

```
      if precedence(ne) < precedence(e) :
         print(o, "(%_)" % [ne])
      else :
         print(o, ne)
   defn print-operator (a:Exp, op:String, b:Exp) :
      print-nested(a)
      print(o, op)
      print-nested(b)
   match(e) :
      (e:Const) : print(o, value(e))
      (e:Variable) : print(o, name(e))
      (e:Log) : print(o, "ln(%_)" % [a(e)])
      (e:Add) : print-operator(a(e), " + ", b(e))
      (e:Subtract) : print-operator(a(e), " - ", b(e))
      (e:Multiply) : print-operator(a(e), " * ", b(e))
      (e:Divide) : print-operator(a(e), " / ", b(e))
      (e:Power) : print-operator(a(e), " ^ ", b(e))

;Overloaded operators
defn plus (a:Exp, b:Exp) : Add(a, b)
defn minus (a:Exp, b:Exp) : Subtract(a, b)
defn times (a:Exp, b:Exp) : Multiply(a, b)
defn divide (a:Exp, b:Exp) : Divide(a, b)
defn bit-xor (a:Exp, b:Exp) : Power(a, b)
defn ln (a:Exp) : Log(a)

;Differentiation algorithm
defn differentiate (e:Exp, x:Symbol) -> Exp :
   defn ddx (e:Exp) : differentiate(e, x)

   match(e) :
      (e:Const) :
         Const(0)
      (e:Variable) :
         if name(e) == x : Const(1)
         else : Const(0)
      (e:Add) :
         ddx(a(e)) + ddx(b(e))
      (e:Subtract) :
         ddx(a(e)) - ddx(b(e))
      (e:Multiply) :
         a(e) * ddx(b(e)) + b(e) * ddx(a(e))
      (e:Divide) :
         val num = b(e) * ddx(a(e)) - a(e) * ddx(b(e))
         val den = b(e) ^ Const(2)
```

```
         num / den
      (e:Power) :
         e * (b(e) * ddx(a(e)) / a(e) + ln(a(e)) * ddx(b(e)))
      (e:Log) :
         ddx(a(e)) / a(e)

;Map helper
defn map (f: Exp -> Exp, e:Exp) -> Exp :
   match(e) :
      (e:Add) : Add(f(a(e)), f(b(e)))
      (e:Subtract) : Subtract(f(a(e)), f(b(e)))
      (e:Multiply) : Multiply(f(a(e)), f(b(e)))
      (e:Divide) : Divide(f(a(e)), f(b(e)))
      (e:Power) : Power(f(a(e)), f(b(e)))
      (e:Log) : Log(f(a(e)))
      (e) : e

;Simplification algorithm
defn simplify (e:Exp) :
   defn const? (e:Exp, v:Int) :
      match(e) :
         (e:Const) : value(e) == v
         (e) : false
   defn one? (e:Exp) : const?(e, 1)
   defn zero? (e:Exp) : const?(e, 0)

   match(map(simplify, e)) :
      (e:Add) :
         if zero?(a(e)) : b(e)
         else if zero?(b(e)) : a(e)
         else : e
      (e:Subtract) :
         if zero?(a(e)) : Const(-1) * b(e)
         else if zero?(b(e)) : a(e)
         else : e
      (e:Multiply) :
         if one?(a(e)) : b(e)
         else if one?(b(e)) : a(e)
         else if zero?(a(e)) or zero?(b(e)) : Const(0)
         else : e
      (e:Divide) :
         if zero?(a(e)) : Const(0)
         else if one?(b(e)) : a(e)
         else : e
      (e:Power) :
```

```
         if one?(a(e)) : Const(1)
         else if zero?(b(e)) : Const(1)
         else : e
      (e:Log) :
         if one?(a(e)) : Const(0)
         else : e
      (e) : e

;Main program
defn main () :
   val x = Variable(`x)
   val [c1, c2, c3, c4] = [Const(1), Const(2), Const(3), Const(4)]
   val exp = c2 * x ^ c2 + (c1 + c3) * x + ln(x + c4)
   val dexp = differentiate(exp, `x)
   val sexp = simplify(dexp)

   println("Original Expression: %_" % [exp])
   println("Differentiated Expression: %_" % [dexp])
   println("Simplified Expression: %_" % [sexp])

;Start!
main()
```

Exercises

1. Our differentiation algorithm is general enough to always give the right answer (for the types of expressions it supports), but it's often *too* general. This is most obvious in the differentiation rule for `Power` expressions. The current rule handles the case where both the base and exponent are functions of x, but typically only one of the two is a function of x and the other is a constant expression. Look for these special cases and handle them more intelligently.
2. Extend the simplifier to be able to simplify `1 + 3` to `4`.
3. Extend the simplifier to be able to simplify `1 + x + 3` to `4 + x`.
4. Extend the simplifier to be able to simplify `x - x` to `0`.
5. Extend the simplifier to be able to simplify `x + 1 - x` to `1`.
6. Extend the simplifier to be able to simplify `x * x` to `x ^ 2`.
7. Extend the simplifier to be able to simplify `x / x` to `1`.
8. Extend the simplifier to be able to simplify `x ^ 2 / x` to `x`.
9. Extend the simplifier to be able to simplify `(x + 1) ^ 2 / (x + 1)` to `x + 1`.

Chapter 8

Parametric Polymorphism

This chapter will introduce you to the concept of *parametric polymorphism* and show you how to parameterize your functions using *type arguments*, and your types using *type parameters*.

8.1 The Need for Polymorphism

Thus far, none of the functions you have written so far have been parameterized by type. Here is an example implementation of a function that reverses a list of integers.

```
defn reverse-list (xs:List<Int>) -> List<Int> :
   if empty?(xs) :
      xs
   else :
      append(
         reverse-list(tail(xs))
         List(head(xs)))
```

But notice that it only works on integers. Thus the following does not compile.

```
reverse-list(List("Timon", "and", "Pumbaa"))
```

It gives this error.

```
Cannot call function reverse-list of type List<Int> -> List<Int>
with arguments of type (FullList<String>).
```

To handle this, we can write an overloaded version of `reverse-list` that accepts a list of strings.

```
defn reverse-list (xs:List<String>) -> List<String> :
   if empty?(xs) :
      xs
```

```
   else :
      append(
         reverse-list(tail(xs))
         List(head(xs)))
```

Now `reverse-list` will work on both integers and strings. So the following

```
println(reverse-list(List(1, 2, 3)))
println(reverse-list(List("Timon", "and", "Pumbaa")))
```

compiles and prints out

```
(3 2 1)
("Pumbaa" "and" "Timon")
```

However, the code for the string version of `reverse-list` is identical to the integer version, save for its type signature. This is an obvious duplication of effort. Also, this is clearly a subpar solution. What if we next want to reverse a list of characters? It is not practical to define an overloaded version of `reverse-list` for every type of list we wish to reverse.

The Limitations of the ? Type

What we need is the ability to call `reverse-list` on lists of *any* type. Well, we've already learned about one mechanism that will allow us to do this: the `?` type. So let's replace our two overloaded `reverse-list` functions with a single one that accepts a `List<?>` as its argument.

```
defn reverse-list (xs:List) -> List :
   if empty?(xs) :
      xs
   else :
      append(
         reverse-list(tail(xs))
         List(head(xs)))
```

Recall that the default type parameter is `?` for a type without explicit type parameters. Thus `List` is equivalent to `List<?>`. The above definition of `reverse-list` *will* allow us to call both lists of integers and strings. Try out the following code again

```
println(reverse-list(List(1, 2, 3)))
println(reverse-list(List("Timon", "and", "Pumbaa")))
```

and verify that it still prints out

```
(3 2 1)
("Pumbaa" "and" "Timon")
```

It seems to work fine now on these cases. What is the problem?

The problem is in the type of the result of the `reverse-list` function. `reverse-list` is annotated to return a `List<?>`. Thus the following obviously incorrect code will still compile.

```
val xs = reverse-list(List("Timon", "and", "Pumbaa"))
println(head(xs) + 1)
```

When the compiled program is ran, it crashes with this error.

```
FATAL ERROR: Cannot cast value to type.
   at core/core.stanza:2619.12
   at test.stanza:15.8
```

This is disappointing. The reverse of a list of strings is obviously still a list of strings. So `head(xs)` should be a `String`, and Stanza should have stopped us from trying to add an integer to it. More precisely, what we need is the ability for `reverse-list` to accept lists of any type, but have it also return lists of the *same* type.

In place of `reverse-list`, we'll instead call the `reverse` function included in Stanza's core library, and see that it does not suffer from these problems.

```
val xs = reverse(List("Timon", "and", "Pumbaa"))
println(head(xs) + 1)
```

Attempting to compile the above gives this error.

```
No appropriate function plus for arguments of type (String, Int).
Possibilities are:
   plus: (Byte, Byte) -> Byte at core/core.stanza:2488.21
   plus: (Int, Int) -> Int at core/core.stanza:2619.12
   plus: (Long, Long) -> Long at core/core.stanza:2688.21
   plus: (Float, Float) -> Float at core/core.stanza:2742.21
   plus: (Double, Double) -> Double at core/core.stanza:2792.21
```

which is much more reassuring. We'll now see how we can write such functions ourselves.

8.2 Explicit Type Arguments

Here is how to write a *polymorphic* `reverse-list` function that takes an explicit type argument.

```
defn reverse-list<ElementType> (xs:List<ElementType>) -> List<ElementType> :
   if empty?(xs) :
      xs
   else :
      append(
         reverse-list<ElementType>(tail(xs))
         List(head(xs)))
```

`reverse-list` takes a single type argument called `ElementType` that represents the type of the elements inside the `xs` list. Now we need to provide a type argument to `reverse-list` when we call it.

```
reverse-list<Int>(List(1, 2, 3))
```

What that does is *instantiate* a version of `reverse-list` by replacing `ElementType` with `Int` in its type signature. Thus the instantiated function has type

```
List<Int> -> List<Int>
```

and we then call it with `List(1, 2, 3)`. Let's use our polymorphic function to reverse lists of integers and strings.

```
val xs = reverse-list<Int>(List(1, 2, 3))
val ys = reverse-list<String>(List("Timon", "and", "Pumbaa"))
println(xs)
println(ys)
```

Compiling and running the above prints out the same message as before.

```
(3 2 1)
("Pumbaa" "and" "Timon")
```

Let's also verify that the return type of `reverse-list` is of the proper type.

```
val xs = reverse-list<String>(List("Timon", "and", "Pumbaa"))
println(head(xs) + 1)
```

Attempting to compile the above gives this error.

```
No appropriate function plus for arguments of type (String, Int).
Possibilities are:
   plus: (Byte, Byte) -> Byte at core/core.stanza:2488.21
   plus: (Int, Int) -> Int at core/core.stanza:2619.12
   plus: (Long, Long) -> Long at core/core.stanza:2688.21
   plus: (Float, Float) -> Float at core/core.stanza:2742.21
   plus: (Double, Double) -> Double at core/core.stanza:2792.21
```

So the return type is correct, and Stanza properly catches our mistakes.

Note that we are responsible for instantiating a correct version of `reverse-list` to call. If we pass in the wrong type arguments,

```
reverse-list<String>(List(1, 2, 3))
```

then the program will fail to compile. The above gives this error when we attempt to compile it.

```
Cannot call function reverse-list of type List<String> -> List<String>
with arguments of type (FullList<Int>).
```

As a comment on programming style, the purpose of each type argument in a polymorphic function is typically quite obvious. Thus programmers do not feel the need to give them descriptive names. Here is how `reverse-list` would commonly be written.

```
defn reverse-list<T> (xs:List<T>) -> List<T> :
   if empty?(xs) :
      xs
   else :
      append(
         reverse-list<T>(tail(xs))
         List(head(xs)))
```

The vast majority of type arguments are simply named `T` (short for `Type`), or `S` (because it's a letter close to `T`).

8.3 Captured Type Arguments

Our polymorphic `reverse-list` function can now reverse lists of any type and also correctly returns a list of the same type. It's just a little cumbersome to use because we have to pass in the element type of the list we're reversing each time. This is because `T` is declared as an *explicit* type argument. We'll see now how to have Stanza automatically infer the type argument by declaring it as a *captured* type argument. Here is a polymorphic `reverse-list` written using a *captured* type argument.

```
defn reverse-list<?T> (xs:List<?T>) -> List<T> :
   if empty?(xs) :
      xs
   else :
      append(
         reverse-list(tail(xs))
         List(head(xs)))
```

A captured type argument is declared with a `?` prefix, which indicates that it is not passed in explicitly. Instead, it is *captured* from the types of the arguments it is called with. The type signature above says that `reverse-list` requires a list to be passed in for `xs`. Capture `T` from the element type of `xs`.

Now we can call `reverse-list` without passing in an explicit type argument.

```
reverse-list(List(1, 2, 3))
```

The argument `List(1, 2, 3)` has type `List<Int>`, and thus the type argument `T` captures the element type `Int`.

In the following call,

```
reverse-list(List("Timon", "and", "Pumbaa"))
```

the argument `List("Timon", "and", "Pumbaa")` has a type `List<String>`, and thus the type argument `T` captures the element type `String`.

Let's try our example of reversing both integer lists and string lists again.

```
val xs = reverse-list(List(1, 2, 3))
val ys = reverse-list(List("Timon", "and", "Pumbaa"))
println(xs)
println(ys)
```

Notice that we no longer need to pass in type arguments. Compiling and running the above prints out

```
(3 2 1)
("Pumbaa" "and" "Timon")
```

We can also verify that the return type is correct.

```
val xs = reverse-list(List("Timon", "and", "Pumbaa"))
println(head(xs) + 1)
```

Attempting to compile the above gives this error.

```
No appropriate function plus for arguments of type (String, Int).
Possibilities are:
   plus: (Byte, Byte) -> Byte at core/core.stanza:2488.21
   plus: (Int, Int) -> Int at core/core.stanza:2619.12
   plus: (Long, Long) -> Long at core/core.stanza:2688.21
   plus: (Float, Float) -> Float at core/core.stanza:2742.21
   plus: (Double, Double) -> Double at core/core.stanza:2792.21
```

Thus the `reverse-list` function is now polymorphic and it does not require any explicit type arguments. We've finished generalizing `reverse-list` at this point, and it actually now has the same type signature as the `reverse` function in the core library.

Capture Locations

Here's another example polymorphic function.

```
defn store-in-odd-slots<?T> (xs:Array<?T>, v:T) -> False :
   for i in 1 to length(xs) by 2 do :
      xs[i] = v
```

`store-in-odd-slots` is a polymorphic function that accepts an array, `xs`, and an item, `v`, and stores `v` at every odd index in `xs`. Let's try it out.

```
val xs = to-array<String>(["Patrick", "Sunny", "Luca", "Whiskey", "Emmy", "Rummy"])
store-in-odd-slots(xs, "and")
println(xs)
```

prints out

```
["Patrick" "and" "Luca" "and" "Emmy" "and"]
```

Let's now take a closer look at the type signature of `store-in-odd-slots`.

```
defn store-in-odd-slots<?T> (xs:Array<?T>, v:T) -> False
```

The `?T` following the function name

```
store-in-odd-slots<?T>
```

means that the function is polymorphic and accepts a single captured type argument. The argument list

```
(xs:Array<?T>, v:T)
```

contains two references to `T`, but only one of them is prefixed with a `?`. This means that `T` is captured *only* from the element type of `xs`.

The capture location for `T` was chosen carefully. Consider the following type definitions.

```
deftype Shape
deftype Circle <: Shape
```

where all circles are also shapes, but not all shapes are circles.

The following usage of `store-in-odd-slots`

```
val shapes = Array<Shape>(10)
store-in-odd-slots(shapes, new Circle)
```

compiles correctly. `T` is captured from the element type of `Array<Shape>`, and is thus `Shape`. The instantiated `store-in-odd-slots` therefore has type

```
(Array<Shape>, Shape) -> False
```

and can be suitably called with `shapes` and `new Circle`.

But this next usage

```
val circles = Array<Circle>(10)
store-in-odd-slots(circles, new Shape)
```

fails with this error

```
Cannot call function store-in-odd-slots of type (Array<Circle>, Circle) -> False
with arguments of type (Array<Circle>, Shape).
```

This is consistent with our intuition. You cannot store an arbitrary shape into an array that can only hold circles. As an exercise, think about what would happen if `store-in-odd-slots` was instead declared the following way.

```
defn store-in-odd-slots<?T> (xs:Array<T>, v:?T) -> False
```

As a general rule of thumb, the majority of polymorphic functions operate on a collection of some sort. The type argument is *almost always* captured from the element type of the collection.

Multiple Capture Locations

After reading the previous section, you might be naturally wondering what happens when there are *multiple* capture locations. If there are multiple capture locations, then the final captured type is the *union* of all the types captured from each location.

Here is an example of a function that makes use of two capture locations.

```
defn append-lists<?T> (xs:List<?T>, ys:List<?T>) -> List<T> :
   if empty?(xs) : ys
   else : cons(head(xs), append-lists(tail(xs), ys))
```

The type argument `T` is captured from both the element type of `xs` *and* the element of type `ys`. Thus if we call `append-lists` on a list of integers and a list of strings,

```
val xs = List(1, 2, 3)
val ys = List("Timon", "and", "Pumbaa")
val zs = append-lists(xs, ys)
```

then the resulting type of `zs` is `List<Int`String>|.

Example: map-list

Let's try writing our own polymorphic `map` function on lists. We'll call ours `map-list`. `map-list` accepts a function, `f`, and a list, `xs`, and returns a new list containing the results of calling `f` on each item in `xs`. To start off, here's the function definition without any type annotations.

```
defn map-list (f, xs) :
   if empty?(xs) :
      List()
   else :
      val y = f(head(xs))
      val ys = map-list(f, tail(xs))
      cons(y, ys)
```

Let's verify that it works as intended.

```
val xs = to-list(["Timon", "and", "Pumbaa" "are", "good", "friends"])
val lengths = map-list(length, xs)
println(lengths)
```

Compiling and running the above prints out

```
(5 3 6 3 4 7)
```

Let's start off with figuring out the type of `xs`, because it seems easier. It's a list for sure, and `map-list` should be able to work on lists of any type. So `xs` is therefore of type

```
xs:List<?T>
```

and `T` is a captured type argument for `map-list`.

Next, let's figure out the type of `f`. It's a function for sure, and it's called with only a single argument. So it's at least

```
f:? -> ?
```

Next we know that `f` is called with items from `xs`, which is a list of `T`'s, so `f` has to accept `T`'s. Now we know it's at least

```
f:T -> ?
```

Finally, what is `f` allowed to return? Well, `f` is allowed to return anything actually. So let's introduce another captured type argument. The final type of `f` is

```
f:T -> ?S
```

Now that we know the types of its arguments, the last step is to figure out what `map-list` returns. We know that it returns a list, and we also know that the list contains the results of calling `f`. Since we now know that `f` returns `S`'s, therefore `map-list` returns a list of `S`'s. Here is the complete type signature for `map-list`.

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

Let's try our test code again with our typed `map-list` function and ensure it works as expected.

```
val xs = to-list(["Timon", "and", "Pumbaa" "are", "good", "friends"])
val lengths = map-list(length, xs)
println(lengths)
```

Running the above prints out

```
(5 3 6 3 4 7)
```

as before.

To double check the inferred return type of `map-list`, let's cast `lengths` to an obviously incorrect type, and read what Stanza says about its type.

```
lengths as False
```

Compiling the above gives us the error

```
Cannot cast expression of type List<Int> to type False.
```

So Stanza says that `lengths` is a list of integers, which is correct.

Example: map-both

Here's some more practice on using captured type arguments. Here is the un-annotated definition for the `map-both` function.

```
defn map-both (f, g, xs) :
   for x in xs map :
      [f(x), g(x)]
```

`map-both` accepts two functions, `f` and `g`, and a list, `xs`, and returns a list containing two-element tuples. The first elements in all the tuples are the results of calling `f` on each item in `xs`, and the second elements in all the tuples are the results of calling `g` on each item in `xs`.

Similar to before, the list, `xs`, is the easiest argument to figure out the type signature for.

```
xs:List<?T>
```

`f` needs to be a function that can be called with items from `xs`, and can return anything.

```
f:T -> ?S
```

`g` also needs to be a function that can called with items from `xs`, and can also return anything.

```
g:T -> ?R
```

`map-both` returns a list of tuples. The first elements in the tuples are results of calling `f`, and the second elements are results of calling `g`.

```
List<[S, R]>
```

Thus the complete definition for `map-both` is

```
defn map-both<?T,?S,?R> (f:T -> ?S, g:T -> ?R, xs:List<?T>) -> List<[S, R]> :
   for x in xs map :
      [f(x), g(x)]
```

Let's try it out on a list of strings.

```
val xs = to-list(["Timon", "and", "Pumbaa", "are", "good", "friends"])
val zs = map-both(
   xs,
   fn (x) : x[2]
   fn (y) : length(y) * 2)
println(zs)
```

which prints out

```
([’m’ 10] [’d’ 6] [’m’ 12] [’e’ 6] [’o’ 8] [’i’ 14])
```

Let's cast `zs` to something silly to see what Stanza says about its type. Attempting to compile the following

```
zs as False
```

gives us this error.

```
Cannot cast expression of type List<[Char, Int]> to type False.
```

So `zs` has type `List<[Char, Int]>`, which is what we expect.

8.4 Parametric Types

You have been shown how to define your own types using `deftype` and also the shorthand `defstruct`. But none of the types you've defined thus far accept *type parameters*. This stood out the most in our definition of the `Stack` type which was only able to store *String* objects. We'll now learn how to declare our own *parametric types*.

Declaring a Parametric Type

Here is an example of a simple type that takes two type parameters.

```
deftype Either<L,R>
```

`Either` contains two wrapped objects, a left object of type `L`, and a right object of type `R`.

This is all there is to defining a parametric type! The rest of this section covers mechanisms that have already been introduced, but we'll go through them in the context of the `Either` type for practice.

Declaring Multis

Let's define the fundamental operations for an `Either` object, which are simply getter functions for retrieving the two wrapped objects.

```
defmulti left<?L> (e:Either<?L,?>) -> L
defmulti right<?R> (e:Either<?,?R>) -> R
```

Notice that the `left` and `right` functions each take only a single type argument. The other type parameter for the `Either` object is left as `?` to indicate that it is free to be anything.

Creating Either Objects

Now let's write a constructor function for creating `Either` objects. We'll start with a function that can only create `Either<Int,String>` objects.

```
defn Either (l:Int, r:String) :
   new Either<Int,String> :
      defmethod left (this) : l
      defmethod right (this) : r
```

Let's try it out.

```
val e = Either(42, "Timon")
println("The left object is %_." % [left(e)])
println("The right object is %_." % [right(e)])
```

prints out

```
The left object is 42.
The right object is Timon.
```

Polymorphic Constructor Function

Now that we can successfully create specific `Either` objects, let's generalize our constructor function by making it polymorphic using type arguments. The following declares `Either` as taking two explicit type arguments, one for each wrapped object.

```
defn Either<L,R> (l:L, r:R) :
   new Either<L,R> :
      defmethod left (this) : l
      defmethod right (this) : r
```

Now `Either` objects are created in the following way.

```
val e = Either<Int,String>(42, "Timon")
```

The way in which `Either` objects are created now resembles how we've been creating many of the other types included in the core library, such as arrays and vectors. This is not a coincidence. The construction function for arrays and vectors are also just regular functions that take explicit type arguments and return instances of parametric types.

Parametric Structs

The `defstruct` expression also accepts type parameters for creating parametric structs. As mentioned previously, the `defstruct` expression is simply a syntactic shorthand for declaring a new type, getter functions for its fields, and a default construction function. Thus all the code we've written previously to define the `Either` type can be neatly expressed as

```
defstruct Either<L,R> :
   left: L
   right: R
```

Constructor Function with Captured Arguments

Specifically for creating `Either` objects, it is also not necessary to have the user explicitly specify the types of the left and right objects. Let's make the constructor function more convenient to call by using captured type arguments.

```
defn Either<?L,?R> (l:?L, r:?R) :
   new Either<L,R> :
      defmethod left (this) : l
      defmethod right (this) : r
```

Now we can create an `Either` object like this

```
val e = Either(42, "Timon")
```

and have Stanza automatically infer that `e` is an `Either<Int,String>` based on the types of `42` and `"Timon"`.

When *not* to Use Captured Arguments

We showed above how to write a constructor function using captured arguments that did not require the left and right object types to be passed in explicitly to `Either`. This makes the constructor function for `Either` objects very similar to the constructor function for `List` objects, which also does not require any explicit type arguments. This is *not* always an appropriate thing to do.

Let us suppose that `Either` is a *mutable* datastructure; that we can change the left and right objects after the object has been created. The type definition for `Either` would stay the same, but it would gain two more fundamental operations.

```
defmulti left<?L> (e:Either<?L,?>) -> L
defmulti right<?R> (e:Either<?,?R>) -> R
defmulti set-left<?L> (e:Either<?L,?>, v:L) -> False
defmulti set-right<?R> (e:Either<?,?R>, v:R) -> False
```

Notice, especially, the capture locations of the type arguments in the setter functions.

The constructor function would be changed to now accept not the left and right objects, but the *initial* left and right objects, since they may change later on.

```
defn Either<?L,?R> (l0:?L, r0:?R) :
   var l = l0
   var r = r0
   new Either<L,R> :
      defmethod left (this) : l
      defmethod right (this) : r
      defmethod set-left (this, v:L) : l = v
      defmethod set-right (this, v:R) : r = v
```

For the next part, let us again assume that we have definitions for some basic shapes.

```
deftype Shape
deftype Circle <: Shape
deftype Rectangle <: Shape

defmethod print (o:OutputStream, c:Circle) : print(o, "Circle")
defmethod print (o:OutputStream, r:Rectangle) : print(o, "Rectangle")
```

Let's try creating a mutable `Either` object now.

```
defn my-favorite-shape () -> Shape :
   new Circle

val e = Either(42, my-favorite-shape())
println("After creation:")
println("The left object is %_" % [left(e)])
println("The right object is %_" % [right(e)])

set-left(e, 256)
set-right(e, new Rectangle)
println("\nAfter mutation:")
println("The left object is %_" % [left(e)])
println("The right object is %_" % [right(e)])
```

Compiling and running the above prints out

```
After creation:
The left object is 42
The right object is Circle

After mutation:
The left object is 256
The right object is Rectangle
```

Everything seems to be working, but pay attention to what happens next.

The type signature for `my-favorite-shape` is not as precise as it could be. It's annotated to return `Shape`, but it's more precise to say that it returns `Circle`. So let's improve `my-favorite-shape`'s type signature.

```
defn my-favorite-shape () -> Circle :
   new Circle
```

Now try compiling and running the program again. It will now give this error.

```
Cannot call function set-right of type (Either<?, Circle>, Circle) -> False
with arguments of type (Either<Int, Circle>, Rectangle).
```

What is going on? Why would changing (actually improving) the type signature for `my-favorite-shape` affect the later call to `set-right`?

The problem, as is evident in the error message, is that the inferred type for `e` is `Either<Int, Circle>`. This is not right. Even though the *initial* right object was a `Circle`, that doesn't mean we want `e` to *only ever* hold `Circle` objects as its right object.

This is one of those cases where using a captured type argument is inappropriate. For a mutable `Either` object, the types of the left and right objects should be passed in explicitly.

Here is the constructor function rewritten to use explicit type arguments.

```
defn Either<L,R> (l0:L, r0:R) :
   var l = l0
   var r = r0
   new Either<L,R> :
      defmethod left (this) : l
      defmethod right (this) : r
      defmethod set-left (this, v:L) : l = v
      defmethod set-right (this, v:R) : r = v
```

And here is our original test code rewritten to pass in explicit type arguments.

```
defn my-favorite-shape () -> Shape :
   new Circle

val e = Either<Int, Shape>(42, my-favorite-shape())
println("After creation:")
println("The left object is %_" % [left(e)])
println("The right object is %_" % [right(e)])

set-left(e, 256)
set-right(e, new Rectangle)
println("\nAfter mutation:")
println("The left object is %_" % [left(e)])
println("The right object is %_" % [right(e)])
```

Verify that it still compiles and runs correctly.

At this point, we can try making the same change to `my-favorite-shape`'s type signature.

```
defn my-favorite-shape () -> Circle :
   new Circle
```

This time, however, the program still compiles and continues to run as before.

Here are the basic rules of thumb for choosing between using explicit or captured type arguments. If you're creating an immutable object then feel free to use captured type arguments. If you're creating a mutable object, then use explicit type arguments.

These issues surrounding captured type arguments and mutable objects is also why `to-array` and `to-vector` require explicit type arguments and why `to-list` does not.

8.5 Match Expressions and Type Erasure

One subtlety concerning Stanza's parametric type system is a concept called *type erasure*. It roughly means that, given a program, if we replace every type argument with the `?` type, it should still run and compute the same result (providing that the original program doesn't fail). Said another way, the setting of a type argument can never change what is computed by a program.

Here is an example of incorrectly attempting to use a type argument to affect which branch is taken in a match expression.

```
defn check-if<T> (x) :
   match(x) :
      (x:T) : true
      (x) : false
```

Let's try it out on a `String` object.

```
println(check-if<Int>("Timon"))
```

Compiling and running the above prints out

```
FATAL ERROR: Cannot cast value to type.
   at test.stanza:15.8
   at test.stanza:18.8
```

Here is what is happening underneath. The *dispatch* type in a match branch has all of its type arguments and parametric types erased. Thus the code above is equivalent to the following

```
defn check-if<T> (x) :
   match(x) :
      (y:?) :
         val x = y as T
         true
      (y:?) :
         false
```

and the error message arises because `y` cannot be cast to a `T` object. We intentionally designed Stanza so that there is no possible way to write a function such as `check-if`.

8.6 Revisiting Stack

At this point, we have all the requisite knowledge for writing a parametric version of our `Stack` class from chapter 6. Here are our old definitions for the `Stack` type and its fundamental operations.

```
deftype Stack <: Collection<String>
defmulti push (s:Stack, x:String) -> False
defmulti pop (s:Stack) -> String
defmulti empty? (s:Stack) -> True|False
```

A `Stack` is declared to be a collection of strings, and its fundamental operations allow us to push and pop strings from it.

Here is the original constructor function.

```
defn Stack (capacity:Int) -> Stack :
   val items = Array<String>(capacity)
   var size = 0
   new Stack :
      defmethod push (this, x:String) :
         if size == capacity : fatal("Stack is full!")
         items[size] = x
         size = size + 1
      defmethod pop (this) :
         if size == 0 : fatal("Stack is empty!")
         size = size - 1
         items[size]
      defmethod empty? (this) :
         size == 0
      defmethod print (o:OutputStream, this) :
         print(o, "Stack containing [")
         print-all(o, join(this, " "))
         print(o, "]")
      defmethod to-seq (this) :
         take-n(size, items)
```

Parametric Type Declaration

The first step is to declare `Stack` as a parametric type.

```
deftype Stack<T> <: Collection<T>
```

Thus, `Stack` now takes a type parameter, `T`, that indicates what types of objects the stack may hold, and is also no longer a collection of strings. It is now a collection of `T`'s.

Polymorphic Fundamental Operations

The second step is to declare its fundamental operations as polymorphic functions.

```
defmulti push<?T> (s:Stack<?T>, x:T) -> False
defmulti pop<?T> (s:Stack<?T>) -> T
defmulti empty? (s:Stack) -> True|False
```

both `push` and `pop` now accept a captured type argument, `T`, that indicates the element type of the stack object. Here are some points to take note of. Notice that the `x` argument for `push` is not a capture location for `T`. This is consistent with our earlier discussion in the section on capture locations. Also notice that the `empty?` multi is unchanged, as the types of the objects in a stack are not needed to check whether the stack is empty.

Polymorphic Constructor Function

The last step is to make its constructor function polymorphic.

```
defn Stack<T> (capacity:Int) -> Stack<T> :
   val items = Array<T>(capacity)
   var size = 0
   new Stack :
      defmethod push (this, x:T) :
         if size == capacity : fatal("Stack is full!")
         items[size] = x
         size = size + 1
      defmethod pop (this) :
         if size == 0 : fatal("Stack is empty!")
         size = size - 1
         items[size]
      defmethod empty? (this) :
         size == 0
      defmethod print (o:OutputStream, this) :
         print(o, "Stack containing [")
         print-all(o, join(this, " "))
         print(o, "]")
      defmethod to-seq (this) :
         take-n(size, items)
```

The constructor function now takes an explicit type argument, `T`, indicating the element type of the stack object, and returns a `Stack<T>`. Notice that the backing array, `items`, is no longer an `Array<String>`. It is now declared as an `Array<T>` in order to hold the items in the stack. `push` now also accepts a `T` value instead of a `String` value. The rest of the function is unchanged.

Trying It Out

Let's try out our parametric `Stack` type using a variation of our original test code.

```
defn main () :
   val s = Stack<Int>(10)
   for x in [1, 5, 2, 42, -11, 2, 5, 10, -42] do :
      push(s, x)

   println("1. Contents of s")
   println(s)

   println("\n2. Index of 42")
   println(index-of(s, 42))

   println("\n3. Does it contain any negative numbers?")
   println(any?({_ < 0}, s))

   println("\n4. Are all numbers negative?")
   println(all?({_ < 0}, s))

   println("\n5. What are the negative numbers?")
   val cap-s = filter({_ < 0}, s)
   println-all(join(cap-s, ", "))

   println("\n6. What are its unique elements?")
   println(unique(s))

main()
```

Compiling and running the above prints out

```
1. Contents of s
Stack containing [1 5 2 42 -11 2 5 10 -42]

2. Index of 42
3

3. Does it contain any negative numbers?
true

4. Are all numbers negative?
false

5. What are the negative numbers?
-11, -42
```

```
6. What are its unique elements?
(1 5 2 42 -11 10 -42)
```

Our parametric stack type is now quite general. It can hold items of different types, and it supports all the operations in the core sequence library. Actually, the `Array` and `Vector` types in the core library are defined in much the same way as `Stack`.

Chapter 9

Advanced Control Flow

Thus far, the only control flow mechanism you've been shown was the `label` construct for creating labeled scopes. But this one construct was powerful enough to express early returns from functions, early breaks from loops, and also (though we haven't shown it) early jumps to the next loop iteration. Each of the above functionality has traditionally been a separate keyword and language feature in other languages, but they're all expressible with just the `label` construct in Stanza. It is a powerful and general mechanism.

In actuality, Stanza only has a single control flow mechanism, called *targetable coroutines*. The `label` construct is just a common usage pattern for them. In this chapter we'll learn about the other common usage pattern of coroutines: attempts and failures, exception handlers, and generators. At the very end, we'll show you the general coroutine construct, along with some examples demonstrating their use.

9.1 First Class Labeled Scopes

Here is a function that finds the smallest power of two that is greater or equal to the given argument, `n`.

```
defn min-pow-2 (n:Int) :
   label<Int> return :
      var x = 1
      while true :
         if x >= n : return(x)
         x = x * 2
      fatal("Unreachable")
```

Let's try it out.

```
defn main () :
   defn test (n:Int) :
      println("The minimum power of 2 greater or equal to %_ is %_." %
```

```
            [n, min-pow-2(n)])
   test(10)
   test(100)
   test(1000)
   test(10000)
   test(100000)
   test(1000000)

main()
```

Compiling and running the above prints out

```
The minimum power of 2 greater or equal to 10 is 16.
The minimum power of 2 greater or equal to 100 is 128.
The minimum power of 2 greater or equal to 1000 is 1024.
The minimum power of 2 greater or equal to 10000 is 16384.
The minimum power of 2 greater or equal to 100000 is 131072.
The minimum power of 2 greater or equal to 1000000 is 1048576.
```

Pulling Out the Body

That's fairly standard so far. But now let's pull out the body of the while loop into a separate function, called `pow-2?`. It accepts an argument, `x`, that is the current number being tested, an argument, `n`, as the limit we're trying to reach, and also a function called `return`, which we'll explain later.

```
defn pow-2? (x:Int, n:Int, return:Int -> Void) -> Int :
   if x >= n : return(x)
   else : x * 2
```

`pow-2?` first checks whether `x` is greater or equal to `n`, and calls `return` with `x` if it is. Otherwise it returns the next value of `x` to `test`, which is `x * 2`.

We now update the `min-pow-2` function to call `pow-2?`.

```
defn min-pow-2 (n:Int) :
   label<Int> return :
      var x = 1
      while true :
         x = pow-2?(x, n, return)
      fatal("Unreachable")
```

Compile and run the program to verify that it still works.

What is happening here!? We've somehow passed the exit function out of `min-pow-2` and into `pow-2?`. Then when `pow-2?` called `return`, it returned not from `pow-2?`, but from `min-pow-2`!.

Let's review the definition of the `label` construct. Here is its general form.

```
label<T> exit :
   body
```

The `label` construct requires the type it returns, `T`, the name of the exit function, `exit`, and the body to execute, `body`. `label` creates an exit function of type `T -> Void` with the name `exit`, and then executes `body`. If `body` never calls the exit function then the result of `body` is returned by `label`. If `body` calls the exit function then `label` *immediately* returns the argument passed to the exit function. The return type `Void` for the exit function indicates that it *doesn't return* to its caller.

There is nothing in the description of `label` preventing us from passing out the exit function, so the call to `return` in `pow-2?` is simply causing the `label` construct to return `x`, which is then returned by `min-pow-2`.

Storing the Exit Function

We can even store the exit function in a variable if we like. Here's a global variable

```
var RETURN: Int -> Void
```

into which we will store the exit function. Thus the exit function will no longer be passed in to `pow-2?` as an argument.

```
defn pow-2? (x:Int, n:Int) -> Int :
   if x >= n : RETURN(x)
   else : x * 2
```

It will instead be stored in the global variable before `pow-2?` is called.

```
defn min-pow-2 (n:Int) :
   label<Int> return :
      RETURN = return
      var x = 1
      while true :
         x = pow-2?(x, n)
      fatal("Unreachable")
```

Compile and verify that the program still works as before.

Here you're starting to see just how flexible the `label` construct really is. Storing the exit function seems like a strange thing to want to do but keep it in the back of your mind as we talk about the other constructs.

9.2 Dynamic Wind

One of the issues that accompanies having a powerful control flow mechanism is that in a sequence of expressions, evaluating the first expression does not guarantee that the last one will be evaluated.

```
f()
g()
h()
```

For example, in the above sequence, even after `f` returns, there is no guarantee that `h` will be called. If `g` calls an exit function then `h` will be skipped entirely.

Let us suppose that we do not want to pass in the limit, `n`, as an argument to `pow-2?`. We would like to keep it stored in a global variable called `LIMIT` and set it to the appropriate value *only during the call* to `pow-2?`. At all other times, `LIMIT` should retain its initial value of `0`. Here is an initial attempt.

```
var LIMIT: Int = 0
var RETURN: Int -> Void

defn pow-2? (x:Int) -> Int :
   if x >= LIMIT : RETURN(x)
   else : x * 2

defn min-pow-2 (n:Int) :
   label<Int> return :
      RETURN = return
      var x = 1
      while true :
         val old-limit = LIMIT
         LIMIT = n
         x = pow-2?(x)
         LIMIT = old-limit
      fatal("Unreachable")

defn main () :
   defn test (n:Int) :
      println("The minimum power of 2 greater or equal to %_ is %_." %
              [n, min-pow-2(n)])
   test(10)
   test(100)
   test(1000)
   test(10000)
   test(100000)
   test(1000000)
```

```
main()
println("After main, LIMIT is %_." % [LIMIT])
```

Printing and compiling the above prints out

```
The minimum power of 2 greater or equal to 10 is 16.
The minimum power of 2 greater or equal to 100 is 128.
The minimum power of 2 greater or equal to 1000 is 1024.
The minimum power of 2 greater or equal to 10000 is 16384.
The minimum power of 2 greater or equal to 100000 is 131072.
The minimum power of 2 greater or equal to 1000000 is 1048576.
After main, LIMIT is 1000000.
```

So `min-pow-2` is working correctly, but `LIMIT` is not being restored back to its original value. What is happening? Well the call to `pow-2?` in

```
LIMIT = n
x = pow-2?(x)
LIMIT = old-limit
```

may call the exit function, `return`. If that happens, then the `label` construct immediately returns and the last `LIMIT = old-limit` expression is never evaluated.

Stanza provides the special function `dynamic-wind` to handle these situations. It allows you to surround a body of code between some *wind in* and *wind out* code. The wind in code is *guaranteed* to execute whenever the control flow enters the body, and the wind out code is *guaranteed* to execute whenever the control flow exits the body. Here is how it's used.

```
defn min-pow-2 (n:Int) :
   label<Int> return :
      RETURN = return
      var x = 1
      while true :
         val old-limit = LIMIT
         dynamic-wind(
            fn () :
               LIMIT = n
            fn () :
               x = pow-2?(x)
            fn (final) :
               LIMIT = old-limit)
      fatal("Unreachable")
```

The wind in, body, and wind out code is given to `dynamic-wind` as three anonymous functions. The `final` argument for the wind out code is a boolean value that indicates whether it is guaranteed to be the last time the wind out code is called. In this example, `final` will always be `true`.

Now compiling and running the program again prints out

```
The minimum power of 2 greater or equal to 10 is 16.
The minimum power of 2 greater or equal to 100 is 128.
The minimum power of 2 greater or equal to 1000 is 1024.
The minimum power of 2 greater or equal to 10000 is 16384.
The minimum power of 2 greater or equal to 100000 is 131072.
The minimum power of 2 greater or equal to 1000000 is 1048576.
After main, LIMIT is 0.
```

indicating the `LIMIT` is properly being reset to its original value.

9.3 Dynamically Scoped Variables

In the above example, we say that `LIMIT` is being used as a *dynamically scoped* variable. This is a common pattern and Stanza provides a syntactic shorthand for our call to `dynamic-wind`.

Here is the `min-pow-2` function written using the `let-var` shorthand.

```
defn min-pow-2 (n:Int) :
   label<Int> return :
      RETURN = return
      var x = 1
      while true :
         let-var LIMIT = n :
            x = pow-2?(x)
      fatal("Unreachable")
```

The general form of `let-var` is

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

It temporarily sets the `x` variable to the value `v` before executing `body`. `x` is restored to its previous value after `body` is finished executing.

9.4 Attempts and Failures

Attempts and failures are syntactic sugar for another use case of targetable coroutines that operate very similarly to labeled scopes. Here is an example.

```
defn read-letter (xs:Seq<Char>) -> Char :
   if not letter?(peek(xs)) : fail()
   next(xs)
```

```
defn read-digit (xs:Seq<Char>) -> Char :
   if not digit?(peek(xs)) : fail()
   next(xs)

defn read-all (xs:Collection<Char>) :
   val xs-seq = to-seq(xs)
   while not empty?(xs-seq) :
      attempt :
         println("Read letter: %_" % [read-letter(xs-seq)])
      else attempt :
         println("Read digit: %_" % [read-digit(xs-seq)])
      else :
         println("Read something else: %~" % [next(xs-seq)])

read-all("42 is the answer.")
```

Compiling the above prints out

```
Read digit: 4
Read digit: 2
Read something else: ' '
Read letter: i
Read letter: s
Read something else: ' '
Read letter: t
Read letter: h
Read letter: e
Read something else: ' '
Read letter: a
Read letter: n
Read letter: s
Read letter: w
Read letter: e
Read letter: r
Read something else: '.'
```

The function `read-all` calls `read-letter` and `read-digit` in an `attempt` block. If the block evaluates without ever calling `fail` then the result of the block is returned by `attempt`. If the block calls `fail`, then `attempt` *immediately* returns the result of evaluating the code in the `else` branch.

9.5 Example: S-Expression Parser

Here is an example of using `attempt` and `fail` to program a simple s-expression parser. An s-expression, in this case, will be defined as either

1. a positive integer,
2. a symbol consisting of letters,
3. or a list containing more s-expressions.

Overall Structure

Here is the basic structure of the parser.

```
defn parse-sexp (sexp:String) -> List :
   val chars = to-seq(sexp)

   defn eat-while (pred?: Char -> True|False) -> String :
      ...

   defn eat-whitespace () -> False :
      ... calls eat-while ...

   defn parse-symbol () -> Symbol :
      ... calls eat-while ...

   defn parse-number () -> Int :
      ... calls eat-while ...

   defn parse-sequence () -> List :
      ... calls eat-whitespace, parse-symbol, parse-number, and parse-list

   defn parse-list () -> List :
      ... calls parse-list ...

   parse-sequence()
   ...
```

parse-sexp is given a string, and returns a list of s-expressions. Upon entering the function, we ask to view the string as a sequence of characters, `chars`. `eat-while` and `eat-whitespace` are helper functions. `parse-symbol`, `parse-number`, `parse-sequence`, and `parse-list` are mutually recursive functions that parse symbols, numbers, sequences of s-expressions, and lists of s-expressions.

Helper Functions

The `eat-while`, and `eat-whitespace` functions are helper functions for reading from `chars`. `eat-while` takes a predicate function, `pred?`, and eats characters from `chars` as long as `pred?` returns `true`. It returns a string containing the eaten characters. `eat-whitespace` eats all leading spaces in `chars`. Here are their definitions.

```
defn eat-while (pred?: Char -> True|False) -> String :
   string-join(take-while(pred?, chars))

defn eat-whitespace () -> False :
   eat-while({_ == ' '})
   false
```

Parsing Symbols

The `parse-symbol` function eats and returns the next symbol from `chars`. If the next character in `chars` is not a letter, then `parse-symbol` fails.

```
defn parse-symbol () -> Symbol :
   if not letter?(peek(chars)) : fail()
   to-symbol(eat-while(letter?))
```

Parsing Numbers

The `parse-number` function eats and returns the next positive integer from `chars`. If the next character in `chars` is not a digit, or if the number cannot be represented in 32 bits, then `parse-number` fails.

```
defn parse-number () -> Int :
   if not digit?(peek(chars)) : fail()
   val x = to-int(eat-while(digit?))
   if x is-not Int : fail()
   x as Int
```

Parsing Sequences

The `parse-sequence` function reads as many s-expressions as possible by calling `parse-symbol`, `parse-number`, and `parse-list` repeatedly.

```
defn parse-sequence () -> List :
   eat-whitespace()
   if empty?(chars) :
      List()
```

```
   else :
      attempt : cons(parse-symbol(), parse-sequence())
      else attempt : cons(parse-number(), parse-sequence())
      else attempt : cons(parse-list(), parse-sequence())
      else : List()
```

Notice the use of `attempt` to first try parsing a symbol, and then if that fails to then try parsing a number, followed by trying to parse a list.

Parsing Lists

The `parse-list` function eats and returns the next list from `chars`. A list is simply a sequence of s-expressions surrounded by `()` characters. If the next character is not an opening parenthesis then `parse-list` fails.

```
defn parse-list () -> List :
   if peek(chars) != '(' : fail()
   next(chars)
   val items = parse-sequence()
   if empty?(chars) : fatal("Unclosed opening parenthesis.")
   else if peek(chars) == ')' : (next(chars), items)
   else : fatal("Expected closing parenthesis but got %~." % [next(chars)])
```

Driver

Finally, to start off the function, we attempt to read as many s-expressions as possible from `chars` using `parse-sequence`.

```
val items = parse-sequence()
if empty?(chars) : items
else : fatal("Unexpected character: %~." % [next(chars)])
```

Listing

Here is the complete definition of `parse-sexp`.

```
defn parse-sexp (sexp:String) :
   val chars = to-seq(sexp)

   defn eat-while (pred?: Char -> True|False) -> String :
      string-join(take-while(pred?, chars))

   defn eat-whitespace () -> False :
      eat-while({_ == ' '})
```

```
      false

   defn parse-symbol () -> Symbol :
      if not letter?(peek(chars)) : fail()
      to-symbol(eat-while(letter?))

   defn parse-number () -> Int :
      if not digit?(peek(chars)) : fail()
      val x = to-int(eat-while(digit?))
      if x is-not Int : fail()
      x as Int

   defn parse-sequence () -> List :
      eat-whitespace()
      if empty?(chars) :
         List()
      else :
         attempt : cons(parse-symbol(), parse-sequence())
         else attempt : cons(parse-number(), parse-sequence())
         else attempt : cons(parse-list(), parse-sequence())
         else : List()

   defn parse-list () -> List :
      if peek(chars) != '(' : fail()
      next(chars)
      val items = parse-sequence()
      if empty?(chars) : fatal("Unclosed opening parenthesis.")
      else if peek(chars) == ')' : (next(chars), items)
      else : fatal("Expected closing parenthesis but got %~." % [next(chars)])

   val items = parse-sequence()
   if empty?(chars) : items
   else : fatal("Unexpected character: %~." % [next(chars)])
```

Trying it Out

Let's try it out on the following string.

```
do(println, parse-sexp("This (is) (commonly (called an (S) (Expression)))"))
```

When compiled and ran it prints out

```
This
(is)
(commonly (called an (S) (Expression)))
```

Unveiling The Internals

The `attempt` construct is just syntactic sugar for a function call.

```
attempt : conseq
else : alt
```

is equivalent to

```
with-attempt(
   fn () : conseq
   fn () : alt)
```

As an exercise, try and implement your own `with-attempt` function by using the `label` construct.

9.6 Exception Handling

Our s-expression parser from the previous example fails when called with invalid input (though with very nice error messages). Here is what happens if we forget a closing parenthesis at the end.

```
do(println, parse-sexp("This (is) (commonly (called an (S) (Expression))"))
```

Compiling and running the above prints out

```
FATAL ERROR: Unclosed opening parenthesis.
   at test.stanza:39.25
   at test.stanza:32.29
   at core/core.stanza:3725.13
   at core/core.stanza:847.16
   at core/core.stanza:3724.14
   ...
```

But what if we cannot guarantee that the input is correct? Suppose we want users to type an arbitrary string into the terminal and print the parsed s-expression if it's well formed, or else ask them to try again if it's not.

A potential solution would be to write another function called `sexp?` that returns `true` or `false` depending on whether its argument is a well formed string. But checking whether an s-expression is well formed is almost as much work as parsing it, so that's an inefficient solution.

Stanza provides us a mechanism for handling this called *exceptions*.

Exception Objects

The first step is to declare our own `Exception` types, one for each type of error the parser can encounter.

```
defstruct UnclosedParenthesis <: Exception
defmethod print (o:OutputStream, e:UnclosedParenthesis) :
   print(o, "Unclosed opening parenthesis.")

defstruct UnmatchedParenthesis <: Exception : (char:Char)
defmethod print (o:OutputStream, e:UnmatchedParenthesis) :
   print(o, "Expected closing parenthesis but got %~." % [char(e)])

defstruct UnexpectedCharacter <: Exception : (char:Char)
defmethod print (o:OutputStream, e:UnexpectedCharacter) :
   print(o, "Unexpected character: %~." % [char(e)])
```

There are three different errors that are detected by our parser.

1. The string is missing a closing parenthesis at the end.
2. We are currently reading a list and encountered a strange character.
3. We've read as many s-expressions as possible and there is a strange character left over.

Throwing Exceptions

The next step is to change the calls to `fatal` to calls to `throw` with our newly defined `Exception` objects.

```
defn parse-sexp (sexp:String) :

   ...

   defn parse-list () -> List :
      ...
      if empty?(chars) : throw(UnclosedParenthesis())
      else if peek(chars) == ')' : (next(chars), items)
      else : throw(UnmatchedParenthesis(next(chars)))

   val items = parse-sequence()
   if empty?(chars) : items
   else : throw(UnexpectedCharacter(next(chars)))
```

Catching Exceptions

The final step is to *catch* the thrown exceptions. We can decide which types of exceptions to catch, and which not to. In this example, we'll assume that the string doesn't contain any strange characters and catch only the unclosed parenthesis error.

```
try :
   do(println, parse-sexp("This (is) (commonly (called an (S) (Expression))"))
catch (e:UnclosedParenthesis) :
   println("You forgot to close an opening parenthesis. Please try again.")
```

Compiling and running the program now prints out

```
You forgot to close an opening parenthesis. Please try again.
```

Here is the general form of the `try` construct.

```
try :
   body
catch (e:ExceptionA) :
   a-handler
catch (e:ExceptionB) :
   b-handler
...
```

The `try` construct evaluates the given `body` after installing the given exception handlers. If `body` is evaluated without ever calling `throw` then its result is returned by `try`. If `body` calls `throw` with some `Exception` object, then `try` immediately searches for the first exception handler that can accept the `Exception` object and returns the result of evaluating that handler.

9.7 Generators

The control flow constructs you've been introduced to so far, labeled scopes, attempts and failures, and exceptions, have all served the purpose of *leaving* a block of code. Generators are the first control flow construct you will learn capable of *resuming* a block of code.

Here is a generator that *yields* the first three positive integers.

```
val xs:Seq<Int> = generate<Int> :
   println("Yielding One")
   yield(1)
   println("Yielding Two")
   yield(2)
   println("Yielding Three")
   yield(3)
```

Notice that the `generate` construct returns a `Seq`. Let's try printing out the items in the sequence.

```
println("The first item in xs is")
println(next(xs))
println("The second item in xs is")
println(next(xs))
println("The third item in xs is")
println(next(xs))
```

Compiling and running the above prints out

```
The first item in xs is
Yielding One
1
The second item in xs is
Yielding Two
2
The third item in xs is
Yielding Three
3
```

The Ability to Resume

It's worth paying attention to the order in which the messages are printed out. The snippet

```
println("The first item in xs is")
println(next(xs))
```

by itself, prints out

```
The first item in xs is
Yielding One
1
```

Thus the call to `next(xs)` causes control to enter the block of code in the `generate` construct. The message `"Yielding One"` is printed out, and then the call to `yield(1)` leaves the `generate` construct and `1` is the return value of `next(xs)`.

The next snippet

```
println("The second item in xs is")
println(next(xs))
```

prints out

```
The second item in xs is
Yielding Two
2
```

Thus the call to `next(xs)` causes control to *re-enter* the block `generate` construct, resuming from just after the first call to `yield`. The message `"Yielding Two"` is printed out, and then the call to `yield(2)` leaves the `generate` construct once again and `2` is the return value of the second call to `next(xs)`.

The last snippet

```
println("The third item in xs is")
println(next(xs))
```

prints out

```
The third item in xs is
Yielding Three
3
```

Similarly, the call to `next(xs)` resumes the block in the `generate` construct from just after the second call to `yield`. The message `"Yielding Three"` is printed out, and then the call to `yield(3)` leaves the `generate` construct once again and `3` is the return value of the third call to `next(xs)`.

Thus the `generate` construct provides a very convenient way of creating a lazily constructed sequence.

General Form

Here is the general form of the `generate` construct.

```
generate<T> :
   body
```

`generate` returns a `Seq<T>` by lazily executing the given `body` in a scope containing the generation functions, `yield` and `break`.

`yield` is of type `T -> False` and its argument becomes an element in the generated `Seq`. Execution of the generate block pauses at `yield`, and is resumed on the next call to `next` on the sequence.

`break` is both of type `() -> Void` and `T -> Void`. If no argument is given to `break`, then execution of the generate block ends here and marks the end of the generated sequence. If an argument is given to `break`, then that element is first yielded before the generate block is ended.

If the generated type, `T`, is not explicitly provided, then it is assumed to be `?` by default.

Example: Flattening a Tuple

In this example, we'll determine whether two tuples contain the same elements as each other if we lay out their elements in depth-first order. For example, the tuple

```
[[1] [2 [3]] [[4 5] 6]]
```

contains the elements `1`, `2`, `3`, `4`, `5`, `6` once laid out in depth-first order.

Here's the most straightforward way of doing this. We'll write a function called `flatten` that returns a `Vector` containing a tuple's elements in depth-first order.

```
defn flatten (x:Tuple) -> Vector :
   val v = Vector<?>()
   defn loop (x) :
      match(x) :
         (x:Tuple) : do(loop, x)
         (x) : add(v, x)
   loop(x)
   v
```

Let's try it out.

```
println(flatten([[1] [2 [3]] [[4 5] 6]]))
```

Compiling and running the above prints out

```
[1 2 3 4 5 6]
```

To check whether two tuples contain the same elements, we can just flatten each of them and then compare the elements.

```
defn same-elements? (a:Tuple, b:Tuple) :
   if all?(equal?, flatten(a), flatten(b)) :
      println("%_ and %_ have the same elements." % [a, b])
   else :
      println("%_ and %_ have different elements." % [a, b])
```

Let's test it out on the following tuples.

```
same-elements?(
   [[1] [2 [3]] [[4 5] 6]]
   [1 [[2 3 4] [5]] [6]])

same-elements?(
   [[1] [2 [3]] [[4 5] 6]]
   [[[0] 2] [3 [4 5]] 6])
```

Compiling and running the above prints out

```
[[1] [2 [3]] [[4 5] 6]] and [1 [[2 3 4] [5]] [6]] have the same elements.
```

```
[[1] [2 [3]] [[4 5] 6]] and [[[0] 2] [3 [4 5]] 6] have different elements.
```

Notice though, that in both cases, we computed a full flattening of both tuples before checking to see whether they are equal. This is obviously inefficient in the second case since we can tell they are clearly different just by examining their first element. How do we avoid computing the full flattening?

The solution is to *lazily* compute the flattening. Let's change `flatten` to use the `generate` construct to lazily compute the flattened tuples. To track how much of the tuples are being flattened let's also add a print statement.

```
defn flatten (x:Tuple) -> Seq :
   val index = to-seq(0 to false)
   generate :
      defn loop (x) :
         match(x) :
            (x:Tuple) :
               do(loop, x)
            (x) :
               println("Yielding Item %_" % [next(index)])
               yield(x)
      loop(x)
```

Compiling and running the program again prints out

```
Yielding Item 0
Yielding Item 0
Yielding Item 1
Yielding Item 1
Yielding Item 2
Yielding Item 2
Yielding Item 3
Yielding Item 3
Yielding Item 4
Yielding Item 4
Yielding Item 5
Yielding Item 5
[[1] [2 [3]] [[4 5] 6]] and [1 [[2 3 4] [5]] [6]] have the same elements.
Yielding Item 0
Yielding Item 0
[[1] [2 [3]] [[4 5] 6]] and [[[0] 2] [3 [4 5]] 6] have different elements.
```

Thus the results are the same as before, and you can see that, for the second comparison, both generators (one for each tuple) are only computing up to the first element.

9.8 Coroutines

The `label`, `attempt`, `try`, and `generate` constructs are all specific usage patterns of Stanza's *targetable coroutine* system. Here we'll show you how to use the coroutine system in its full generality. It is rare in daily programming to encounter a problem that requires a use of coroutines that isn't already handled by one of the special case constructs. But for implementing libraries and frameworks that make heavy use of concurrency and non-standard control flow, coroutines may be indispensable.

Sending Things Out

Here is the function that will represent our coroutine body.

```
defn my-process (co:Coroutine<Int,String>, a:Int) -> String :
   println("Passing out Timon")
   suspend(co, "Timon")
   println("Passing out and")
   suspend(co, "and")
   println("Passing out Pumbaa")
   suspend(co, "Pumbaa")
   println("Coroutine is done")
   "Done"
```

The type `Coroutine<Int,String>` represents a coroutine for which integers are sent into the coroutine, and for which strings are sent back from the coroutine. The function for sending values out of the coroutine is `suspend`.

Let's now create our coroutine object and resume it a few times.

```
println("Create coroutine")
val co = Coroutine<Int,String>(my-process)

println("\nResume with 42")
val x = resume(co, 42)
println("Got back x = %_" % [x])

println("\nResume with 43")
val y = resume(co, 43)
println("Got back y = %_" % [y])

println("\nResume with 44")
val z = resume(co, 44)
println("Got back z = %_" % [z])

println("\nResume with 45")
```

```
val w = resume(co, 45)
println("Got back w = %_" % [w])
```

Notice that `resume` is called with integers. When the above is compiled and ran it prints out

```
Create coroutine

Resume with 42
Passing out Timon
Got back x = Timon

Resume with 43
Passing out and
Got back y = and

Resume with 44
Passing out Pumbaa
Got back z = Pumbaa

Resume with 45
Coroutine is done
Got back w = Done
```

Thus `suspend` acts much like `yield` did for the `generate` construct, and `resume` acts much like `next` did. This is no accident of course. The `generate` construct is implemented in terms of `suspend` and `resume` underneath.

Breaking Things Off

In addition to `suspend`, a function called `break` can also be used to send values out of a coroutine. The difference is that a call to `break` cannot later be resumed.

Let's change our `my-process` function to send out `"Pumbaa"` with `break` instead of `suspend`.

```
defn my-process (co:Coroutine<Int,String>, a:Int) -> String :
   println("Passing out Timon")
   suspend(co, "Timon")
   println("Passing out and")
   suspend(co, "and")
   println("Passing out Pumbaa")
   break(co, "Pumbaa")
   println("Coroutine is done")
   "Done"
```

Compiling and running the program again prints out

```
Create coroutine

Resume with 42
Passing out Timon
Got back x = Timon

Resume with 43
Passing out and
Got back y = and

Resume with 44
Passing out Pumbaa
Got back z = Pumbaa

Resume with 45
FATAL ERROR: Cannot resume coroutine. Coroutine is already closed.
   at core/core.stanza:984.13
   at core/core.stanza:862.16
   at core/core.stanza:897.40
   at core/core.stanza:862.16
   at test.stanza:31.8
```

The coroutine is *closed* after the call to `break`, and thus our call to `resume` fails.

Sending Things In

The obvious unanswered question now is: what is happening with the `42`, `43`, `44`, and `45` values that `resume` is being called with? To answer that, let's update our `my-process` function to print out the return values of `suspend` (and change the call to `break` back into `suspend`).

```
defn my-process (co:Coroutine<Int,String>, a:Int) -> String :
   println("Came in a = %_" % [a])
   println("Passing out Timon")
   val b = suspend(co, "Timon")

   println("Came in b = %_" % [b])
   println("Passing out and")
   val c = suspend(co, "and")

   println("Came in c = %_" % [c])
   println("Passing out Pumbaa")
   val d = suspend(co, "Pumbaa")

   println("Came in d = %_" % [d])
```

```
   println("Coroutine is done")
   "Done"

println("Create coroutine")
val co = Coroutine<Int,String>(my-process)

println("\nResume with 42")
val x = resume(co, 42)
println("Got back x = %_" % [x])

println("\nResume with 43")
val y = resume(co, 43)
println("Got back y = %_" % [y])

println("\nResume with 44")
val z = resume(co, 44)
println("Got back z = %_" % [z])

println("\nResume with 45")
val w = resume(co, 45)
println("Got back w = %_" % [w])
```

Compiling and running the program again prints out

```
Create coroutine

Resume with 42
Came in a = 42
Passing out Timon
Got back x = Timon

Resume with 43
Came in b = 43
Passing out and
Got back y = and

Resume with 44
Came in c = 44
Passing out Pumbaa
Got back z = Pumbaa

Resume with 45
Came in d = 45
Coroutine is done
Got back w = Done
```

Thus, `suspend` sends its argument out from the coroutine, and returns the value sent into the coroutine. `resume` sends its argument into the coroutine, and returns the value sent out from the coroutine.

Closing Things Off

From outside the coroutine body, we may also choose to *close* a coroutine when we're finished with it. Let's try closing our coroutine after getting back `"Pumbaa"`.

```
println("Create coroutine")
val co = Coroutine<Int,String>(my-process)

println("\nResume with 42")
val x = resume(co, 42)
println("Got back x = %_" % [x])

println("\nResume with 43")
val y = resume(co, 43)
println("Got back y = %_" % [y])

println("\nResume with 44")
val z = resume(co, 44)
println("Got back z = %_" % [z])

close(co)

println("\nResume with 45")
val w = resume(co, 45)
println("Got back w = %_" % [w])
```

Compiling and running the above prints out

```
Create coroutine

Resume with 42
Came in a = 42
Passing out Timon
Got back x = Timon

Resume with 43
Came in b = 43
Passing out and
Got back y = and

Resume with 44
```

```
Came in c = 44
Passing out Pumbaa
Got back z = Pumbaa

Resume with 45
FATAL ERROR: Cannot resume coroutine. Coroutine is already closed.
   at core/core.stanza:984.13
   at core/core.stanza:862.16
   at core/core.stanza:897.40
   at core/core.stanza:862.16
   at test.stanza:40.8
```

Checking a Coroutine's Status

There are two functions for checking on the status of a coroutine, `active?` and `open?`.

Calling `active?` on a coroutine will return `true` if the coroutine's body is currently running, and `false` otherwise. Only active coroutines can be suspended or broken from.

Calling `open?` on a coroutine will return `true` if the coroutine's body is not currently running and open to be resumed. Only open coroutines can be resumed.

Nested Coroutines

A coroutine may also launch more coroutines. Notice that unlike `yield`, the calls to `suspend`, `break`, and `resume` explicitly requires, as its first argument, the target coroutine. Being able to explicitly designate the target of the `suspend`, `break`, and `resume` operations are key to allowing nested coroutines to work properly.

Consider the following code, where the coroutine, `co1`, launches a second coroutine, `co2`, within its body. Then within `co2`'s body, there are `suspend` and `break` calls on both `co1` and `co2`. Pay attention to how this interacts.

```
val co1 = Coroutine<False,Int> \$ fn (co1, x0) :
   val co2 = Coroutine<False,False> \$ fn (co2, y0) :
      for i in 0 to false do :
         suspend(co1, i)
         if i == 5 :
            println("Breaking from coroutine 2!")
            break(co2, false)
   resume(co2, false)
   -1

while open?(co1) :
   println(resume(co1, false))
```

Compiling and running the above prints out

```
0
1
2
3
4
5
Breaking from coroutine 2!
-1
```

The two nested coroutines are quite confusing. To get a better sense of what's happening, let's rewrite the second coroutine using the special case `label` construct.

```
val co1 = Coroutine<False,Int> \$ fn (co1, x0) :
   label<False> break :
      for i in 0 to false do :
         suspend(co1, i)
         if i == 5 :
            println("Breaking from coroutine 2!")
            break(false)
   -1

while open?(co1) :
   println(resume(co1, false))
```

Compiling and running the above prints out the same message as before.

It is highly unlikely that you will feel the desire to directly launch new coroutines from within other coroutines, as we did here. But this example shows that they nest appropriately and generally *do the right thing*. Thus, for whatever abstractions you build on top of Stanza's targetable coroutine system, you can rest assured that they will recurse and compose correctly.

9.9 Example: Key Listener

The following example demonstrates using coroutines to easily implement a key listener that translates individual key presses into events on strings.

Let us define a `KeyListener` type, and its fundamental operation.

```
deftype KeyListener
defmulti key-pressed (c:Char) -> False
```

A `KeyListener` object listens to individual key presses from a keyboard and translates them into higher level events. Here is the definition of the constructor function for a `KeyListener`.

```
defn KeyListener (entered: String -> False) -> KeyListener
```

`KeyListener` takes a callback function called `entered` that accepts `String` objects. Our `KeyListener` translates key presses into calls to `entered` on space-separated words. It also supports deleting characters with the backspace key, entering of double-quoted strings, and escaped double-quotes within a double-quoted string. Here is specifically what it has to do.

1. A `KeyListener` has an internal buffer for storing characters. When keys corresponding to letters are pressed, they are stored into the internal buffer.
2. When the backspace key is pressed, the last character is deleted from the internal buffer.
3. Once a full word is completed (indicated by the spacebar being pressed), the `entered` function should be called with the contents of the internal buffer.
4. If the double quote key is pressed, then this indicates that a string is being started, and all subsequent characters until the next double quote should be stored in the internal buffer. Upon completion of the string, the `entered` function should be called with the entire contents of the internal buffer.
5. During entering of a string, if a backslash character followed by a double quote character is entered, then the double quote character is stored in the internal buffer, and entering of the string continues. If the backslash character is not followed by a double quote then both characters are ignored.

Coroutine Framework

Here is the basic framework that we will use to ease programming the `KeyListener`.

```
defn KeyListener (entered: String -> False) -> KeyListener :
   val co = Coroutine<Char,False> \$ fn (co, c0) :
      ;Retrieve the next character
      defn next-char () :
         suspend(co, false)

      ...

   new KeyListener :
      defmethod key-pressed (this, c:Char) :
         resume(co, c)
```

We immediately create a coroutine that accepts characters and sends back dummy values of type `False`. Within the coroutine body, the helper function `next-char` requests the next character by suspending the coroutine and returning the next character sent back into the coroutine. A new `KeyListener` object is returned that calls `resume` on the coroutine whenever a key is pressed.

Buffer Managing Routines

The following definitions within the coroutine body help us manage the `KeyListener`'s internal buffer.

```
;Buffer commands
val buffer = Vector<Char>()
defn pop-char () :
   if not empty?(buffer) :
      pop(buffer)
defn add-char (c:Char) :
   add(buffer, c)
defn empty-buffer () :
   entered(string-join(buffer))
   clear(buffer)
```

The buffer is represented as a vector of characters. `pop-char` removes the last character in the buffer if possible. `add-char` adds the given character to the end of the buffer. `empty-buffer` calls the `entered` callback with the contents of the buffer, and then clears the buffer.

Dispatch Mode

The key press parser operates in a number of different modes. The default mode is the dispatch mode, which determines which mode to next enter based on the previously pressed key.

```
;Dispatch mode
defn* parse (c:Char) :
   if letter?(c) :
      parse-word(c)
   else if c == '\"' :
      parse-string(next-char())
   else :
      parse(next-char())
```

If a letter key was pressed, then we start parsing a word event. If the double-quote key is pressed, then we start parsing a string event. Otherwise, the key press is ignored.

Word Mode

The word parsing mode accepts key presses until one word is completed.

```
;Word parsing mode
defn* parse-word (c:Char) :
```

```
   if letter?(c) :
      add-char(c)
      parse-word(next-char())
   else if c == '\b' :
      pop-char()
      parse-word(next-char())
   else if (c == ' ') or (c == '\"') :
      empty-buffer()
      parse(c)
   else :
      parse-word(next-char())
```

If the last key pressed was a letter, then that letter is added to the buffer. If the last key was a backspace, then we delete a character from the buffer. If the last key was the spacebar or the double-quote key, then the word is completed. We empty the buffer and then go back to the dispatch mode. All other keys are ignored.

String Mode

The string parsing mode accepts key presses until another (un-escaped) double-quote key finishes the string.

```
;String parsing mode
defn* parse-string (c:Char) :
   if c == '\"' :
      empty-buffer()
      parse(next-char())
   else if c == '\b' :
      pop-char()
      parse-string(next-char())
   else if c == '\\' :
      if next-char() == '\"' : add-char('\"')
      parse-string(next-char())
   else :
      add-char(c)
      parse-string(next-char())
```

If the last key pressed was the double-quote key then the string is completed. We empty the buffer and then go back to the dispatch mode. If the last key was a backspace, then we delete a character from the buffer. If the last key was the backslash key, then we add a double-quote to the buffer if the following key is a double-quote. Otherwise both keys are ignored. Finally, all other characters are added to the buffer.

Testing the KeyListener

Here is the entire `KeyListener` constructor function.

```
defn KeyListener (entered: String -> False) -> KeyListener :
   val co = Coroutine<Char,False> \$ fn (co, c0) :
      ;Retrieve the next character
      defn next-char () :
         suspend(co, false)

      ;Buffer commands
      val buffer = Vector<Char>()
      defn pop-char () :
         if not empty?(buffer) :
            pop(buffer)
      defn add-char (c:Char) :
         add(buffer, c)
      defn empty-buffer () :
         entered(string-join(buffer))
         clear(buffer)

      ;Dispatch mode
      defn* parse (c:Char) :
         if letter?(c) :
            parse-word(c)
         else if c == '\"' :
            parse-string(next-char())
         else :
            parse(next-char())

      ;Word parsing mode
      defn* parse-word (c:Char) :
         if letter?(c) :
            add-char(c)
            parse-word(next-char())
         else if c == '\b' :
            pop-char()
            parse-word(next-char())
         else if (c == ' ') or (c == '\"') :
            empty-buffer()
            parse(c)
         else :
            parse-word(next-char())

      ;String parsing mode
```

```
      defn* parse-string (c:Char) :
         if c == '\"' :
            empty-buffer()
            parse(next-char())
         else if c == '\b' :
            pop-char()
            parse-string(next-char())
         else if c == '\\' :
            if next-char() == '\"' : add-char('\"')
            parse-string(next-char())
         else :
            add-char(c)
            parse-string(next-char())

      ;Launch!
      parse(c0)

   new KeyListener :
      defmethod key-pressed (this, c:Char) :
         resume(co, c)
```

Let's try it out on some simulated key presses.

```
defn keys-pressed (kl:KeyListener, cs:Seqable<Char>) :
   do(key-pressed{kl, _}, cs)

defn main () :
   val kl = KeyListener \$ fn (s) :
      println("String entered: %_" % [s])

   ;Test backspace
   keys-pressed(kl, "Timom\bn ")

   ;Test simple word
   keys-pressed(kl, "and ")

   ;Test backspace against empty buffer
   keys-pressed(kl, "P\b\b\b\bPumbaa")

   ;Test unrecognized characters
   keys-pressed(kl, " a\#\$!re")

   ;Test strings with escaped quotes
   keys-pressed(kl, \<S>"\"good\" friends!!"<S>)

main()
```

Note that

```
\<S>literal !@\#\$%\|" characters<S>
```

is Stanza's syntax for a literal un-escaped string. All characters between the starting `<S>` tag and the ending `<S>` tag are part of the string. Any tag may be used in place of `S`.

Compiling and running the above prints out

```
String entered: Timon
String entered: and
String entered: Pumbaa
String entered: are
String entered: "good" friends!!
```

Our `KeyListener` calls the callback function at the correct times and with the correct input!

The coroutine mechanism allowed us to keep the code fairly straightforward and modular, even though the logic behind the `KeyListener` itself is actually quite sophisticated. As an exercise, you may try to implement an equivalent `KeyListener` function *without* using the coroutine mechanism to fully appreciate how tedious and error-prone it is.

Chapter 10

Stanza's Type System

Types are the basis for how Stanza decides whether expressions are legal or not, and how to select the appropriate version of an overloaded function. This chapter will explain the different kinds of types in Stanza, what values each type represents, and the subtype relation.

10.1 Kinds of Types

There are only a handful of basic kinds of types in Stanza. Here is a listing of them all.

1. Ground Types (e.g. `Int`, `String`, `True`)
2. Parametric Types (e.g. `Array<Int>`, `List<String>`)
3. Tuple Types (e.g. `[Int]`, `[Int, String]`, `[Int, True, String]`)
4. Function Types (e.g. `Int -> String`, `(Int, Int) -> Int`)
5. Union Types (e.g. `Int`String|, `True`False|, `Circle`Rectangle|Point|)
6. Intersection Types (e.g. `Collection<Int>&Lengthable`)
7. Void Type (`Void`)
8. Unknown Type (`?`)

Each kind of type will be described in detail in this chapter. The only type that has not been introduced yet is the void type. Most importantly, we'll cover the *subtyping* rules for each kind of type, which are the rules that Stanza uses to determine whether a program is legal.

10.2 The Subtype Relation

Stanza's type system is built upon the *subtyping* relation. The most important operation on types is determining whether one type is a *subtype* of another. We use the following notation

```
A <: B
```

to indicate that the type `A` is a subtype of the type `B`. Intuitively, what this means is that Stanza will allow you to pass a value of type `A` to any place that is expecting a value of type `B`.

In previous chapters, we have demonstrated examples using the types

```
deftype Shape
deftype Circle <: Shape
```

According to these definitions, the `Circle` type is a subtype of `Shape`.

```
Circle <: Shape
```

This means that we may pass a `Circle` to any place that is expecting a `Shape`. For example, we can call functions that accept `Shape` arguments with `Circle` objects. We can initialize values and variables of type `Shape` with `Circle` objects. And we can return `Circle` objects from functions annotated to return `Shape` objects.

Whether one type is a subtype of another is calculated from a set of *subtyping rules*. There is a small set for handling each kind of type, and we'll introduce them to you gradually.

10.3 Ground Types

Ground types are the most basic types in Stanza and are simply types that don't take any type parameters. The majority of types used in daily programming are simple ground types. `Int`, `String`, `True`, `False`, and `Char` are a few examples of ground types that you've used.

Reflexivity Rule

There are two subtyping rules for ground types. The first is that a ground type is a subtype of itself.

```
T <: T
```

This rule is almost trivial. For example, here are some relations derivable from this rule.

```
Int <: Int
String <: String
```

```
True <: True
```

meaning that you can call a function that accepts `String` with an `String` object.

Parent Rule

Users may define their own ground types using `deftype`. For example, here is the type declaration for `Circle` again.

```
deftype Circle <: Shape
```

The general form for `deftype` is

```
deftype T <: P
```

Here is the second subtyping rule for ground types. The type `T` is a subtype of `X` if it can be proven that its parent type `P` is a subtype of `X`.

```
Assuming deftype T <: P
T <: X if P <: X
```

Thus we can derive

```
Circle <: Shape
```

from

```
Assuming deftype Circle <: Shape
Circle <: Shape because Shape <: Shape
```

This rule is what allows us to pass `Circle` objects to functions that accept `Shape` objects.

10.4 Parametric Types

Parametric types are types that take one or more *type parameters*. `Array<Int>`, `List<String>`, and our own type, `Stack<String>`, are examples of parametric types.

Covariance Rule

First consider the case where a base type, `A`, takes only a single type parameter. This rule says that a parametric type `A<T>` is a subtype of another parametric type `A<S>` if it can be proven that its type parameter, `T`, is a subtype of the other's type parameter, `S`.

```
A<T> <: A<S> if T <: S
```

In general, for arbitrary numbers of type parameters, the parametric type `A<T1,T2, ..., Tn>` is a subtype of another parametric type `A<S1, S2, ..., Sn>` if its

type parameters, T1, T2, ..., Tn are respectively subtypes of the other's type parameters, S1, S2, ..., Sn.

```
A<T1, T2, ..., Tn> <: A<S1, S2, ..., Sn> if
   T1 <: S1 and
   T2 <: S2 and
   ...
   Tn <: Sn
```

For example, we can derive

```
List<Circle> <: List<Shape>
```

from

```
List<Circle> <: List<Shape> because
   Circle <: Shape
```

This rule is what allows us to pass a list of circles to a function expecting a list of shapes.

Parent Rule

Consider again the simple case where a base type, A, takes a single type parameter. Assume that A is defined the following way.

```
deftype A<S> <: P
```

The parent rule for parametric types says that the parametric type A<T> is a subtype of X if it can be proven that the result of replacing every occurrence of S in P with T is a subtype of X.

```
Assuming deftype A<S> <: P
A<T> <: X if P[S := T] <: X
```

where the notation P[S := T] stands for the result of replacing every occurrence of S in P with T.

Our parametric Stack type, for example, is declared

```
deftype Stack<T> <: Collection<T>
```

We can derive

```
Stack<Circle> <: Collection<Shape>
```

from

```
Stack<Circle> <: Collection<Shape> because
   Collection<Circle> <: Collection<Shape> because
      Circle <: Shape
```

Here is the general form of the rule for arbitrary numbers of type parameters.

```
Assuming deftype A<S1, S2, ..., Sn> <: P
A<T1, T2, ..., Tn> <: X if
   P[S1 := T1, S2 := T2, ..., Sn := Tn] <: X
```

It says that the parametric type `A<T1, T2, ..., Tn>` is a subtype of `X` if it can be proven that the result of replacing every occurrence of `S1`, `S2`, ..., `Sn` in `P` respectively with `T1`, `T2`, ..., `Tn` is a subtype of `X`.

10.5 Tuple Types

Tuple types are used for representing the types of tuple objects. They're special in that they take a *variable* number of type parameters. Here is an example of a tuple type. The type

```
[Int, String]
```

represents a two-element tuple, where the first element is an `Int` and the second element is a `String`.

Covariance Rule

This rule says that the tuple type `[T1, T2, ..., Tn]` is a subtype of the tuple type `[S1, S2, ..., Sn]` if the types of the elements `T1`, `T2`, ..., `Tn` are respectively subtypes of the types of the other's elements `S1`, `S2`, ..., `Sn`.

```
[T1, T2, ..., Tn] <: [S1, S2, ..., Sn] if
   T1 <: S1 and
   T2 <: S2 and
   ...
   Tn <: Sn
```

For example, we can derive

```
[Circle, Rectangle] <: [Shape, Shape]
```

from

```
[Circle, Rectangle] <: [Shape, Shape] because
   Circle <: Shape and
   Rectangle <: Shape
```

Collapsed Tuple Rule

The type `Tuple` is used to represent a tuple of *unknown* arity. This rule allows us to pass tuples with known arity to places expecting tuples with unknown arity.

```
[T1, T2, ..., Tn] <: X if Tuple<T1|T2|...|Tn> <: X
```

It says that a tuple of known arity containing elements of type `T1`, `T2`, ..., `Tn` is a subtype of `X` if it can be proven that the tuple of unknown arity `Tuple<T1`T2|...|Tn>| is a subtype of `X`.

The type `Tuple<T>` is defined to be a subtype of `Collection<T>` in the core library, so this rule is what allows us to pass in tuples to functions that expect collections. For example, we can derive

```
[Int, Int, Int] <: Collection<Int>
```

from

```
[Int, Int, Int] <: Collection<Int> because
   Tuple<Int|Int|Int> <: Collection<Int> because
      Int|Int|Int <: Int
```

10.6 Function Types

Function types are used to represent the type of function objects. We've used them in the previous chapters to write functions that accept other functions as arguments.

Let us first consider just functions that take a single argument.

```
T1 -> S1 <: T2 -> S2 if
   S1 <: S2 and
   T2 <: T1
```

The rule says that a function type `T1 -> S1` is a subtype of another function type `T2 -> S2` if it can be proven that the return type of the first, `S1`, is a subtype of the return type of the second, `S2`, and if the argument type of the second, `T2`, is a subtype of the argument type of the first, `T1`.

Covariance

This rule is a little confusing at first, so let's go it carefully. First, the following relation

```
Int -> Circle <: Int -> Shape
```

can be derived from

```
Int -> Circle <: Int -> Shape because
   Circle <: Shape and
   Int <: Int
```

Suppose we are calling a function, `f`, that requires a function argument. What this rule means is that if `f` requires its argument to return `Shape` objects, then we are allowed to pass it a function that returns `Circle` objects. This makes sense as all circles are shapes.

So assuming that `f` calls its argument function, then whatever `f` will do with the resultant `Shape` objects, `f` can also do with `Circle` objects.

Contravariance

Next, the following relation

```
Shape -> Int <: Circle -> Int
```

can be derived from

```
Shape -> Int <: Circle -> Int because
   Int <: Int and
   Circle <: Shape
```

Suppose we are again calling a function, `f`, that requires a function argument. What this rule means is that if `f` requires its argument to accept `Circle` objects, then we are allowed to pass it a function that accepts `Shape` objects. This makes sense as all functions that can accept `Shape` objects, can also accept `Circle` objects.

The general rule for function types results from the combination of functions being *covariant* in its return type and `contravariant` in its argument types.

General Form

Here is the general form of the function subtyping rule for arbitrary numbers of arguments.

```
(T1, T2, ..., Tn) -> R1 <: (S1, S2, ..., Sn) -> R2 if
   R1 <: R2 and
   S1 <: T1 and
   S2 <: T2 and
   ...
   Sn <: Tn
```

Thus a function type `(T1, T2, ..., Tn) -> R1` is a subtype of another function type `(S1, S2, ..., Sn) -> R2` if it can be proven that the return type of the first, `R1`, is a subtype of the return type of the second, `R2`, and the argument types of the second, `S1`, `S2`, ..., `Sn`, are respectively subtypes of the argument types of the first, `T1`, `T2`, ..., `Tn`.

10.7 Union Types

Union types are used to represent a value who could either be of one type or another. The type `Int`String|, for example, represents a value that could either be an `Int` or a `String`.

Expecting a Union Type

The following rule says that a type, `X`, is a subtype of a union type, `AB|`, if it can be proven that `X` is either a subtype of `A` or a subtype of `B`.

```
X <: A|B if X <: A or X <: B
```

For example, we can derive

```
Int <: Int|String
```

from

```
Int <: Int|String because Int <: Int
```

This rule allows us to write functions that accept a variety of types, and be allowed to pass it a specific one.

Passing a Union Type

The following rule says that a union type, `AB|`, is a subtype of `X`, if it can be proven that both `A` is a subtype of `X` and `B` is a subtype of `X`.

```
A|B <: X if A <: X and B <: X
```

For example, we can derive

```
Circle|Rectangle|Point <: Shape
```

from

```
Circle | (Rectangle|Point) <: Shape because
   Circle <: Shape and
   Rectangle|Point <: Shape because
      Rectangle <: Shape and
      Point <: Shape
```

This rule is what causes Stanza to error if you attempt to pass a `IntString|` object to a function that requires an `Int` object.

10.8 Intersection Types

Intersection types are the dual of union types, and are used to indicate that a value is both of one type and also of another. The type `Collection<Int> & Lengthable`, for example, represents an object that is simultaneously both a collection of integers and also a `Lengthable` object.

Expecting an Intersection Type

The following rule says that a type, `X`, is a subtype of an intersection type, `A&B`, if it can be proven that `X` is both a subtype of `A` and also a subtype of `B`.

```
X <: A&B if X <: A and X <: B
```

For example, we can derive

```
Stack<Int> <: Collection<Int> & Lengthable
```

from

```
Stack<Int> <: Collection<Int> & Lengthable because
   Stack<Int> <: Collection<Int> and
   Stack<Int> <: Lengthable
```

Passing an Intersection Type

The following rule says that an intersection type `A&B` is a subtype of `X` if it can be proven that either `A` is a subtype of `X` or `B` is a subtype of `X`.

```
A&B <: X if A <: X or B <: X
```

For example, we can derive

```
Stack<Int> <: Lengthable
```

from

```
Stack<Int> <: Lengthable because
   Collection<Int> & Lengthable <: Lengthable because
      Lengthable <: Lengthable
```

10.9 The Void Type

The void type is a special type in Stanza that represents *no value*.

It is used, for example, as the return type of the `fatal` function, which simply prints an error message and then immediately quits the program. `fatal` never returns, so it's inappropriate to say that it returns *any* type. `throw` is another function that returns `Void`, as it also never returns to its caller.

It is occasionally also used as a type parameter for collection types. For example, the following call

```
val xs = List()
```

creates an object of type `List<Void>` and assigns it to `xs`. Recall that calling `head` on a value of type `List<T>` returns `T`. Similarly, calling `head` on a value of type `List<Void>` returns `Void`, indicating that such a call would not return.

The only subtyping rule for `Void` is that it is a subtype of any type, `T`.

```
Void <: T
```

For programmers familiar with the `void` type in the C and Java programming language, note that this is not the same concept. A C function that returns `void` still returns. It simply returns a meaningless value, so you're forbidden from using it for anything. In contrast, a Stanza function that returns `Void` *does not return.*

10.10 The Unknown Type

The unknown type is a very important type and forms the basis of Stanza's optional typing system. There are two subtyping rules that defines its behaviour.

Expecting an Unknown Type

The following rule says that *any* type, `T`, is a subtype of `?`.

```
T <: ?
```

When we declare a function that accepts arguments of type `?`, it is this rule that allows us to pass any object to the function.

Passing an Unknown Type

The following rule says that the unknown type, `?`, is a subtype of *any* type, `T`.

```
? <: T
```

Given a value or argument declared with the `?` type, it is this rule that allows us to pass this value anywhere, regardless of what type is actually expected.

These two rules together allows Stanza to model the behaviour of dynamically-typed scripting languages in a principled manner. The behaviour of the Python programming language, for example, can be mimicked by declaring every argument and value as having the unknown type.

Chapter 11

Calling Foreign Functions

One of the most important features that a practical programming language must support is the ability to call functions written in other languages. There are too many useful libraries written in the established languages to consider rewriting them in another programming language. Stanza provides support for calling any function using the calling convention for the C programming language. This means that you can use any library written in C, or that provides a C interface, in Stanza. Since the dominant consumer operating systems today use a C calling convention, this means that the vast majority of libraries can be called from Stanza. This chapter will show you how.

11.1 Writing a C Function

Here is a fibonacci function written in C. Create a `fibonacci.c` file with the following contents.

```
\#include<stdio.h>
\#include<stdlib.h>

int generate_fib (int n) {
   int a = 0;
   int b = 1;
   while(n > 0){
      printf("%d\n", b);
      int c = a + b;
      a = b;
      b = c;
      n = n - 1;
   }
   return 0;
}
```

```
int main (int nargs, char** argvs) {
   generate_fib(10);
   return 0;
}
```

Compile and run the above program by typing

```
cc fibonacci.c -o fibonacci
./fibonacci
```

in the terminal. It should print out

```
1
1
2
3
5
8
13
21
34
55
```

In the next step, we will call the `generate_fib` function from Stanza.

11.2 Calling our C Function

The first step is just to remove the `main` function in `fibonacci.c` since the program is now being initialized and driven by Stanza.

Next create a file named `fib.stanza` with the following contents.

```
defpackage fib :
   import core
   import collections

extern generate_fib: int -> int

lostanza defn call-fib () -> ref<False> :
   call-c generate_fib(10)
   return false

println("Calling fibonacci")
call-fib()
println("Done calling fibonacci")
```

To compile both the `fib.stanza` and `fibonacci.c` files together, and run the program, type the following into the terminal.

```
stanza fib.stanza -ccfiles fibonacci.c -o fib
./fib
```

It should print out

```
Calling fibonacci
1
1
2
3
5
8
13
21
34
55
Done calling fibonacci
```

Thus our Stanza program successfully calls and returns from the `generate_fib` function written in C. Let's go through the program step by step.

Declaring an External Function

The line

```
extern generate_fib: int -> int
```

declares that there is a function defined externally called `generate_fib` that takes a single integer argument and returns a single integer argument.

Notice that `int` is not capitalized. This is important. `int` refers to the *LoStanza* integer type, and is different from the *Stanza* type `Int`. We'll go over what this means later.

Let us suppose that `generate_fib` took two arguments instead of one. Make the following change to the `generate_fib` function, where it now accepts an argument, `b0`, to indicate the initial value of `b`.

```
int generate_fib (int b0, int n) {
   int a = 0;
   int b = b0;
   ...
}
```

Then the `extern` statement, and the call to `generate_fib` would have to be updated accordingly.

```
extern generate_fib: (int, int) -> int

lostanza defn call-fib () -> ref<False> :
   call-c generate_fib(2, 10)
   return false
```

Compiling and running the new program now prints out

```
Calling fibonacci
2
2
4
6
10
16
26
42
68
110
Done calling fibonacci
```

Declaring a LoStanza Function

LoStanza is a small sub-language within Stanza that allows users to precisely specify data layouts and perform low-level hardware operations. LoStanza can be used for writing high performance code, communicating with external peripherals, and implementing system level functions. Stanza's garbage collector, for example, is written in LoStanza. In this chapter, we are using it to interface with externally defined functions.

The line

```
lostanza defn call-fib () -> ref<False>
```

declares a LoStanza function called `call-fib`. Its return type, `ref<False>`, indicates that it returns a reference to the Stanza type, `False`.

The line

```
call-c generate_fib(10)
```

calls the `generate_fib` function with the argument `10`. The `call-c` tells Stanza to call `generate_fib` with the *C calling convention*. By default, Stanza uses the *Stanza calling convention* to call other functions, and if you forget the `call-c` it will seriously confuse `generate_fib` and crash the program.

Finally, the line

```
return false
```

simply returns `false` to whomever called `call-fib`.

C Functions that Return void

When a C function is declared to return a value of type `void`, it means that the function is called for its side effects only, and returns an arbitrary value. Let's change `generate_fib` to return `void`.

```
void generate_fib (int b0, int n) {
   int a = 0;
   int b = b0;
   while(n > 0){
      int c = a + b;
      printf("%d\n", c);
      a = b;
      b = c;
      n = n - 1;
   }
}
```

Stanza does not provide any explicit support for modeling arbitrary values, so the `extern` statement would remain as

```
extern generate_fib: (int, int) -> int
```

and, as the programmer, you would have to remember (or document) that `generate_fib` returns some random integer that should not be used.

11.3 Calling LoStanza from Stanza

The arguments to `generate_fib` are currently hardcoded into the `call-fib` function. Let's change that to allow both `b0` and `n` to be passed as arguments to `call-fib`.

```
extern generate_fib: (int, int) -> int

lostanza defn call-fib (b0:int, n:int) -> ref<False> :
   call-c generate_fib(b0, n)
   return false
```

And our test code will now call `call-fib` with different arguments.

```
println("Calling fibonacci(1, 10)")
call-fib(1, 10)
println("Calling fibonacci(2, 10)")
call-fib(2, 10)
println("Done calling fibonacci")
```

However, attempting to compile the above gives us the following error.

```
LoStanza function call-fib of type (int, int) -> ref<False>
can only be referred to from LoStanza.
```

As mentioned, `int` is a *LoStanza* type, and you're not allowed to call it directly from Stanza with *Stanza* objects.

Convert Stanza Objects to LoStanza Values

The type `Int` is declared like this.

```
lostanza deftype Int :
   value: int
```

We will explain what that means in more detail later, but for now, notice that it contains a field called `value` that is of type `int`. Thus, we will modify our `call-fib` function to accept references to `Int` objects, and then pass their `value` fields to `generate_fib`.

```
lostanza defn call-fib (b0:ref<Int>, n:ref<Int>) -> ref<False> :
   call-c generate_fib(b0.value, n.value)
   return false
```

With this change, the program now compiles correctly, and prints out

```
Calling fibonacci(1, 10)
1
1
2
3
5
8
13
21
34
55
Calling fibonacci(2, 10)
2
2
4
6
10
16
26
42
68
110
```

```
Done calling fibonacci
```

A LoStanza function can be called from Stanza if and only if all of its argument types and return type are `ref<T>`, indicating that it accepts and returns a reference to a Stanza object. LoStanza functions that can be suitably called from Stanza are indistinguishable from regular Stanza functions. So in addition to being called directly, they can also be passed as arguments, and stored in datastructures.

Convert LoStanza Values to Stanza Objects

Let us now change `generate_fib` to return the `n`'th fibonacci number, instead of printing all of them.

```
int generate_fib (int b0, int n) {
   int a = 0;
   int b = b0;
   while(n > 0){
      int c = a + b;
      a = b;
      b = c;
      n = n - 1;
   }
   return b;
}
```

We'll also update our `call-fib` function to return the result of `generate_fib`.

```
lostanza defn call-fib (b0:ref<Int>, n:ref<Int>) -> int :
   val result = call-c generate_fib(b0.value, n.value)
   return result
```

Here's the updated test code that prints out the result of calling `call-fib`.

```
println("fibonacci(1, 10) =")
println(call-fib(1, 10))
println("fibonacci(2, 10) =")
println(call-fib(2, 10))
println("Done calling fibonacci")
```

However, attempting to compile the above gives us this familiar error.

```
LoStanza function call-fib of type (ref<Int>, ref<Int>) -> int
can only be referred to from LoStanza.
```

As mentioned already, a LoStanza function can be called from Stanza if and only if all of its argument types *and return type* are `ref<T>`. We learned how to convert Stanza `Int` objects into LoStanza `int` values in the previous section. Now we'll learn how to convert LoStanza `int` values back into Stanza `Int` objects.

To create a Stanza `Int` object, we use the LoStanza `new` operator.

```
lostanza defn call-fib (b0:ref<Int>, n:ref<Int>) -> ref<Int> :
   val result = call-c generate_fib(b0.value, n.value)
   return new Int{result}
```

Our test code now compiles and runs, and prints out

```
fibonacci(1, 10) =
89
fibonacci(2, 10) =
178
Done calling fibonacci
```

Note that the LoStanza `new` operator is completely different than the Stanza `new` operator. It is best to consider LoStanza as a completely separate language from Stanza. It has its own syntax, operators, and behaviour. The thing that makes LoStanza unique is that there is a well-defined and flexible interface between it and Stanza.

11.4 LoStanza Types

There are a handful of additional LoStanza types in addition to the `int` type that we used in the declaration of the `generate_fib` function.

Primitive Types

Here is a listing of the rest of the LoStanza primitive types, along with an example of their values.

```
val x:byte = 42Y
val x:int = 42
val x:long = 42L
val x:float = 42.0f
val x:double = 42.0
```

A `byte` is an 8-bit unsigned integer. An `int` is a 32-bit signed integer. A `long` is a 64-bit signed integer. A `float` is a 32-bit single precision floating point number. And a `double` is a 64-bit double precision floating point number.

The above primitive types have an associated Stanza type, each declared to contain a single `value` field containing the LoStanza representation of its value. The associated Stanza types for `byte`, `int`, `long`, `float`, and `double`, are `Byte`, `Int`, `Long`, `Float`, and `Double`, respectively. In addition to `Byte`, the Stanza type `Char` is also declared to contain a single `value` field of type `byte`.

As an example, let us write a version of `generate_fib` that works on double precision floating point numbers.

```
double generate_fib_d (double b0, int n) {
   double a = 0.0;
   double b = b0;
   while(n > 0){
      double c = a + b;
      a = b;
      b = c;
      n = n - 1;
   }
   return b;
}
```

Here is the LoStanza code needed to be able to call `generate_fib_d` from Stanza.

```
extern generate_fib_d: (double, int) -> double

lostanza defn call-fib (b0:ref<Double>, n:ref<Int>) -> ref<Double> :
   val result = call-c generate_fib_d(b0.value, n.value)
   return new Double{result}
```

Now armed with double precision floating point, let's calculate the 100'th fibonacci number.

```
println("fibonacci(1.0, 100) = ")
println(call-fib(1.0, 100))
println("fibonacci(2.0, 100) = ")
println(call-fib(2.0, 100))
println("Done calling fibonacci")
```

Compiling and running the above prints out

```
fibonacci(1.0, 100) =
573147844013817200640.000000000000000
fibonacci(2.0, 100) =
1146295688027634401280.000000000000000
Done calling fibonacci
```

Notice that the `call-fib` function is overloaded to accept both `Int` and `Double` arguments. LoStanza functions have all the same features as Stanza functions, and this includes their ability to be overloaded.

Pointer Types

Pointers are represented in LoStanza with the `ptr<t>` type. The little `t` represents any LoStanza type. For example, here is the type representing a pointer to an `int`,

```
ptr<int>
```

and here is the type representing a *pointer to* a pointer to an `int`,

```
ptr<ptr<int>>
```

The type

```
ptr<?>
```

represents a generic pointer to anything.

As an example of their use, let's call the C `malloc` and `free` functions to allocate and delete space for three integers.

```
extern malloc: long -> ptr<?>
extern free: ptr<?> -> int

lostanza defn try-pointers () -> ref<False> :
   val ints:ptr<int> = call-c malloc(3 * sizeof(int))
   call-c free(ints)
   return false
```

The `[]` operator in LoStanza is the dereference operator and retrieves the value stored at the given pointer address. Here is an example of storing and retrieving values into and from the `ints` pointer.

```
lostanza defn try-pointers () -> ref<False> :
   val ints:ptr<int> = call-c malloc(3 * sizeof(int))
   [ints] = 42
   [ints + 4] = 43
   [ints + 8] = 44
   val x = [ints]
   val y = [ints + 4]
   val z = [ints + 8]
   call-c free(ints)
   return false
```

Programmers familiar with C should note that arithmetic on pointers *do not* automatically operate in terms of the size of the pointer's data type. To retrieve the `i`'th element from a pointer, assuming that its elements are stored contiguously, we use the following syntax.

```
lostanza defn try-pointers () -> ref<False> :
   val ints:ptr<int> = call-c malloc(3 * sizeof(int))
   ints[0] = 42
   ints[1] = 43
   ints[2] = 44
   val x = ints[0]
   val y = ints[1]
   val z = ints[2]
```

```
   call-c free(ints)
   return false
```

This is equivalent to the previous example.

Declaring a LoStanza Type

Consider the following definition of the C type `Point3D` and function `get_origin`.

```
typedef struct {
   float x;
   float y;
   float z;
} Point3D;

Point3D* get_origin () {
  Point3D* p = (Point3D*)malloc(sizeof(Point3D));
  p->x = 0.0f;
  p->y = 0.0f;
  p->z = 0.0f;
  return p;
}
```

`Point3D` is a struct that contains three `float` fields, and `get_origin` returns a pointer to a `Point3D`.

Here is how we would declare our own LoStanza type to mirror the C type definition.

```
lostanza deftype Point3D :
   x: float
   y: float
   z: float
```

Here's a function that demonstrates calling `get_origin` and returning the `x` field in the returned point.

```
extern get_origin: () -> ptr<Point3D>

lostanza defn origin-x () -> ref<Float> :
   val p = call-c get_origin()
   return new Float{p.x}
```

Here's some code to test the `origin-x` function.

```
println("The x coordinate of the origin is %_." % [origin-x()])
```

which prints out

```
The x coordinate of the origin is 0.000000.
```

Reference Types

A reference to a Stanza object is represented with the `ref<T>` type. The big `T` represents any Stanza type. We've already used the `ref<Int>`, and `ref<Float>` types in our examples.

Our previous function `origin-x` returned the `x` coordinate of the origin. But we would really like to just return the entire point to Stanza. Similar to how we converted `int` values to `Int` objects, this is done using the `new` operator.

```
lostanza defn origin () -> ref<Point3D> :
   val p = call-c get_origin()
   return new Point3D{p.x, p.y, p.z}
```

And here are the LoStanza getter functions for a `Point3D` that allows Stanza to retrieve the coordinates within it.

```
lostanza defn x (p:ref<Point3D>) -> ref<Float> :
   return new Float{p.x}
lostanza defn y (p:ref<Point3D>) -> ref<Float> :
   return new Float{p.y}
lostanza defn z (p:ref<Point3D>) -> ref<Float> :
   return new Float{p.z}
```

Here's some code to test our new `origin` function.

```
val p = origin()
println("The x coordinate of the origin is %_." % [x(p)])
println("The y coordinate of the origin is %_." % [y(p)])
println("The z coordinate of the origin is %_." % [z(p)])
```

Compiling and running the above code prints out

```
The x coordinate of the origin is 0.000000.
The y coordinate of the origin is 0.000000.
The z coordinate of the origin is 0.000000.
```

As one last example, let's write, in LoStanza, a constructor function for `Point3D` objects that can be called from Stanza.

```
lostanza defn Point3D (x:ref<Float>, y:ref<Float>, z:ref<Float>) -> ref<Point3D> :
   return new Point3D{x.value, y.value, z.value}
```

Here's some test code for trying out our constructor function.

```
val p2 = Point3D(1.0f, 3.4f, 4.2f)
println("The x coordinate of p2 is %_." % [x(p2)])
println("The y coordinate of p2 is %_." % [y(p2)])
println("The z coordinate of p2 is %_." % [z(p2)])
```

which, when compiled and ran, prints out

```
The x coordinate of p2 is 1.000000.
The y coordinate of p2 is 3.400000.
The z coordinate of p2 is 4.200000.
```

With these definitions, `Point3D` becomes a type that we can freely manipulate from Stanza. We can create `Point3D` objects, and we can retrieve its fields.

Literal Strings

A literal string in LoStanza has type `ptr<byte>` and refers to a pointer to a memory location where the ascii byte representation of its characters are stored.

For example, the following snippet will retrieve the ascii byte value of the character `'o'` and store it in the value `c`.

```
val str:ptr<byte> = "Hello"
val c:byte = str[4]
```

The characters are also stored with a terminating zero byte after all the characters. This allows the literal strings to be suitably used with external libraries expecting C language strings.

External Unknown Arity Functions

Neither LoStanza nor Stanza supports the definition of functions that take an unknown number of arguments. But there are external libraries containing such functions. The C `printf` function is the most famous one.

The `printf` function would be declared like this.

```
extern printf: (ptr<byte>, ? ...) -> int
```

Here is an example of calling it from a function called `test`.

```
lostanza defn test () -> ref<False> :
   call-c printf("The friendship between %s and %s is valued at over %d.\n",
                 "Timon", "Pumbaa", 9000)
   return false

test()
```

Compiling and running the above prints out

```
The friendship between Timon and Pumbaa is valued at over 9000.
```

11.5 External Global Variables

Let us suppose that `generate_fib` was written differently. Suppose that it does not accept any arguments, and also returns `void`. Instead it will retrieve its argument from a global variable named `FIB_PARAM`, and store the result in `FIB_PARAM` when finished.

```
int FIB_PARAM;

void generate_fib (void) {
   int a = 0;
   int b = 1;
   while(FIB_PARAM > 0){
      int c = a + b;
      a = b;
      b = c;
      FIB_PARAM = FIB_PARAM - 1;
   }
   FIB_PARAM = b;
}
```

To call the new `generate_fib`, our LoStanza `call-fib` function would need to be able to read and write to the `FIB_PARAM` variable. Here's how to do that.

```
extern FIB_PARAM : int
extern generate_fib : () -> int

lostanza defn call-fib (n:ref<Int>) -> ref<Int> :
   FIB_PARAM = n.value
   call-c generate_fib()
   return new Int{FIB_PARAM}

println("fib(10) = %_" % [call-fib(10)])
```

Compiling and running the above prints out

```
fib(10) = 89
```

11.6 Function Pointers

Certain C libraries tend to make heavy use of function pointers for implementing callbacks or parameterized behaviour. Let us suppose there is a C function called `choose_greeting` that when given an integer argument returns one of several possible greeting functions to return. These greeting functions then accept a C string and print out an appropriate message.

```
void standard_greeting (char* name) {
```

```
  printf("Hello %s!\n", name);
}

void chill_greeting (char* name) {
  printf("'Sup %s.\n", name);
}

void excited_greeting (char* name) {
  printf("%c", name[0]);
  for(int i=0; i<5; i++)
    printf("%c", name[1]);
  printf("%s! Heyyyy!\n", name+2);
}

typedef void (*Greeting)(char* name);
Greeting choose_greeting (int option) {
  switch(option){
  case 1: return &chill_greeting;
  case 2: return &excited_greeting;
  default: return &standard_greeting;
  }
}
```

The `extern` declaration for `choose_greeting` would look like this.

```
extern choose_greeting: int -> ptr<(ptr<byte> -> int)>
```

Here's how to decipher that piece by piece. The returned greeting functions all have type

```
ptr<byte> -> int
```

The `choose_greeting` function returns a *pointer* to a greeting function. So the return type of `choose_greeting` is

```
ptr<(ptr<byte> -> int)>
```

And `choose_greeting`, itself, requires an integer argument. Thus the full type for `choose_greeting` is

```
int -> ptr<(ptr<byte> -> int)>
```

Here is the LoStanza `greet` function which takes an integer argument called `option` and greets Patrick appropriately.

```
lostanza defn greet (option:ref<Int>) -> ref<False> :
   val greet = call-c choose_greeting(option.value)
   call-c [greet]("Patrick")
   return false
```

Notice that the value `greet` has type `ptr<(ptr<byte> -> int)>`, and thus it needs to be first dereferenced before it can be called.

```
call-c [greet]("Patrick")
```

Let's try it out!

```
println("Option 0")
greet(0)

println("\nOption 1")
greet(1)

println("\nOption 2")
greet(2)
```

Compiling and running the above prints out

```
Option 0
Hello Patrick!

Option 1
'Sup Patrick.

Option 2
Paaaaatrick! Heyyyy!
```

11.7 The Address Operator

The `greet` function in the previous example accepts an integer argument to select the type of greeting, but it only ever greets Patrick. Let's generalize `greet` to accept whom to greet as well.

We want `greet` to be callable from Stanza, so the name will be passed in as a `String` object.

```
lostanza defn greet (option:ref<Int>, name:ref<String>) -> ref<False> :
   ...
```

But the `greet` function requires a `ptr<byte>` as its argument, and `name` is a `ref<String>`. How do we get access to a pointer to the string's characters?

The `String` type is declared in the core library as

```
lostanza deftype String :
   length: long
   hash: int
   chars: byte ...
```

The ellipsis following the `byte` indicates that the `String` object ends with a variable number of trailing `byte` values. We need a pointer to those values to call `greet` with. To do that we will use the `addr` operator, which will return the pointer address of a location.

Let's now write our `greet` function with the `addr` operator.

```
lostanza defn greet (option:ref<Int>, name:ref<String>) -> ref<False> :
   val greet = call-c choose_greeting(option.value)
   call-c [greet](addr(name.chars))
   return false
```

And update our test code to pass in a different name for each type of greeting.

```
println("Option 0")
greet(0, "Emmy")

println("\nOption 1")
greet(1, "Patrick")

println("\nOption 2")
greet(2, "Luca")
```

Attempting to compile the above, however, gives us this error.

```
Cannot retrieve address of unstable location using addr operator.
```

What does that mean?

Stable and Unstable Locations

Underneath the hood, Stanza uses a precise *relocating* garbage collector. What this means is that objects are constantly being shuffled around in memory during the garbage collection process. An unstable location is a location whose address is not fixed, such as a field in a Stanza object. In contrast, a stable location is one whose address is fixed, such as a piece of memory allocated using `malloc`.

The error above is saying that we cannot use the `addr` operator to retrieve the address of `name.chars`, which is an unstable location. `name` is a Stanza string and will be relocated whenever the garbage collector runs, and so the address of `name.chars` is constantly changing.

However, we are planning to pass the address of `name.chars` to C and then *immediately* start executing C code. Additionally, the C function is guaranteed not to hang onto the pointer after it returns. Thus, in this particular case, we know that Stanza's garbage collector will never have a chance to run, and it *is* safe to retrieve the pointer of `name.chars`.

To *force* Stanza to give you the address of an unstable location, Stanza provides you the `addr!` operator. So let's update our `greet` function by using the `addr!` operator this time,

```
lostanza defn greet (option:ref<Int>, name:ref<String>) -> ref<False> :
   val greet = call-c choose_greeting(option.value)
   call-c [greet](addr!(name.chars))
   return false
```

and try compiling and running the program again. The program now prints out

```
Option 0
Hello Emmy!

Option 1
'Sup Patrick.

Option 2
Luuuuuca! Heyyyy!
```

You should stick to using the `addr` operator whenever you can, and use the `addr!` operator only when you're *very* sure that the object won't be relocated while you're using the pointer.

11.8 Calling LoStanza from C

So far we've only considered calling C functions from Stanza, but what if you wanted to call a Stanza function from C? Stanza supports both directions of calling and this section will explain how.

Let us reconsider the `generate_fib` function again. This time, we will have `generate_fib` call a Stanza function for each number that is generated. Here is the code for `generate_fib`.

```
\#include<stdio.h>
\#include<stdlib.h>

void number_generated (int x);

void generate_fib (int n) {
   int a = 0;
   int b = 1;
   while(n > 0){
      number_generated(b);
      int c = a + b;
      a = b;
      b = c;
      n = n - 1;
   }
}
```

Notice that we assume the existence of a function called `number_generated` that we can call from C.

C will call `number_generated` using the C calling convention, so we need to be able to define a LoStanza function that is expecting to be called with the C calling convention. The `extern` keyword will allow us to do that. Our `number_generated` function will push the generated number to a global vector called `FIB_NUMBERS`.

```
val FIB_NUMBERS = Vector<Int>()

extern defn number_generated (n:int) -> int :
   add(FIB_NUMBERS, new Int{n})
   return 0
```

The implementation of the `call-fib` function remains as it was before.

```
extern generate_fib: int -> int

lostanza defn call-fib (n:ref<Int>) -> ref<False> :
   call-c generate_fib(n.value)
   return false
```

Let's try it out then! Here's our test code.

```
call-fib(20)
println("Generated Numbers: %_" % [FIB_NUMBERS])
```

Compiling and running the above prints out

```
Generated Numbers: [1 1 2 3 5 8 13 21 34 55 89 144 233
                    377 610 987 1597 2584 4181 6765]
```

11.9 Passing Callbacks to C

In the last section, we showed you how to write a LoStanza function that can be called with C. However, C libraries are not typically architected to directly call a named user function. Instead, the user will pass the library a pointer to a callback function that is then later called by the library.

Let's change our `generate_fib` function so that it no longer directly calls the `number_generated` function. It will accept instead, as an argument, a pointer to a callback function which it will call.

```
void generate_fib (int n, void (*number_generated)(int x)) {
   int a = 0;
   int b = 1;
   while(n > 0){
      number_generated(b);
```

```
      int c = a + b;
      a = b;
      b = c;
      n = n - 1;
   }
}
```

We shall keep the LoStanza definition of `number_generated` the same, but we will need to change the declaration of the `generate_fib` function, and also pass a pointer to `number_generated` to the call to `generate_fib`.

```
extern generate_fib: (int, ptr<(int -> int)>) -> int

lostanza defn call-fib (n:ref<Int>) -> ref<False> :
   call-c generate_fib(n.value, addr(number_generated))
   return false
```

Notice the use of the standard `addr` operator for retrieving the address of the `number_generated` function.

Compiling and running the above prints out

```
Generated Numbers: [1 1 2 3 5 8 13 21 34 55 89 144 233
                    377 610 987 1597 2584 4181 6765]
```

Let's take this time to review everything that this example demonstrates.

1. Stanza is calling `call-fib`, which is a function written in LoStanza.
2. `call-fib` is calling `generate_fib`, which is a function written in C.
3. `generate_fib` is passed a pointer to the `number_generated` function which is written in LoStanza.
4. `generate_fib` runs and calls `number_generated` multiple times.
5. Each time `number_generated` is called, it creates a Stanza `Int` object from the argument passed to it by `generate_fib`, and calls the Stanza function `add` to push it onto a vector.

This will likely be the most complicated usage of Stanza's foreign function interface you will come across, but it's nice to know that the flexibility is there when you need it.

Chapter 12

Appendix

Stanza has a number of convenience constructs that make your life easier, but they are not necessary for day to day programming. You may skim through this appendix and learn about these constructs as their need arises.

12.1 Stanza Compiler Options

.stanza Configuration File

Stanza's platform and compiler settings are stored in the `.stanza` file that was created when you installed Stanza with `stanza install`. When you run Stanza it will first look for an appropriate `.stanza` file. Here are the places that Stanza searches in, in order, for the `.stanza` file.

1. Stanza first looks in the current working directory.
2. If the `STANZA_CONFIG` environment variable is set, then Stanza looks in that directory.
3. If the `HOME` environment variable is set, then Stanza looks in that directory.

Basic Compilation

To compile `myfile.stanza` and generate the binary `myprogram` use the following command.

```
stanza myfile.stanza -o myprogram
```

Optimization

To compile with optimizations, use the `-optimize` flag.

```
stanza myfile.stanza -o myprogram -optimize
```

Be warned that Stanza's optimizer is only designed to handle *correct* programs. A *correct* program is defined to be a program that successfully runs to completion without ever failing with a call to `fatal`. If an unoptimized program runs to completion and generates a result, then the optimized program is guaranteed to run to completion and generate the same result. However, if the unoptimized program fails, then the behaviour of the optimized program is undefined.

Generating Assembly Files

By default, Stanza generates a temporary `.asm` file containing the generated assembly instructions and then links it with GCC. To use a specific name for the `.asm` file use the `-s` flag.

```
stanza myfile.stanza -s myprogram.s -o myprogram
```

The above command will generate the assembly file `myprogram.s` and link it to produce the binary file `myprogram`.

For expert users that only want the assembly file, the `-o` flag may be omitted. The following command only generates the assembly file `myprograms.s`.

```
stanza myfile.stanza -s myprogram.s
```

Pkg Files

Stanza's separate compilation system allows for packages to be compiled into `.pkg` files. The following command compiles each package in `myfile.stanza` to a separate `.pkg` file.

```
stanza myfile.stanza -pkg
```

By default, the resultant `.pkg` files are generated in the current working directory. To specify the folder into which they should be generated, provide the path after the `-pkg` flag. The following command puts the resultant `.pkg` files in the `mypkgs` folder.

```
stanza myfile.stanza -pkg mypkgs
```

Note that the current compiler requires for source files containing mutually dependent packages to be compiled together. For example, if `myfile1.stanza` contains

```
defpackage mypackage1 :
   import mypackage2
...
```

and `myfile2.stanza` contains

```
defpackage mypackage2 :
   import mypackage1
```

...

then `myfile1.stanza` and `myfile2.stanza` must be compiled together with the following command.

```
stanza myfile1.stanza myfile2.stanza -pkg
```

Automatic Pkg Loading

When you compile a program, Stanza automatically looks for the `.pkg` files containing the definitions of the packages that you import. Here is the order in which Stanza looks for appropriate `.pkg` files.

1. If you've provided a path using the `-pkg-path` flag, then Stanza will first look there for `.pkg` files. For example, the following command compiles `myfile.stanza` using the `.pkg` files in the `mypkgs` folder.

   ```
   stanza myfile.stanza -pkg-path mypkgs
   ```

2. If the `-pkg-path` flag is not provided, then Stanza will first look in the current working directory for `.pkg` files.

3. If the `-optimize` flag is provided, then Stanza will look in the directories specified by the `fast-pkg-dirs` option in your `.stanza` configuration file. To add additional directories to the pkg path, add the following to your `.stanza` file.

   ```
   fast-pkg-dirs = ("/path/to/myfastpkgs1" "/path/to/myfastpkgs2")
   ```

4. If the `-optimize` flag is provided, then Stanza will look in the `fast-pkgs` folder in your Stanza installation directory.

5. Stanza will then look in the directories specified by the `pkg-dirs` option in your `.stanza` configuration file. To add additional directories to the pkg path, add the following to your `.stanza` file.

   ```
   pkg-dirs = ("/path/to/mypkgs1" "/path/to/mypkgs2")
   ```

6. Stanza will then look in the `pkgs` folder in your Stanza installation directory.

C Compiler Options

Stanza provides the `-ccfiles` flag to include additional files to the call to the C compiler. The following command compiles the `myfile.stanza` program and links it against the functions contained in `supportfunctions.c` to produce the `myprogram` executable.

```
stanza myfile.stanza -ccfiles supportfunctions.c -o myprogram
```

You may also use the `-ccflags` flag to include additional flags to the C compiler. The following command compiles the `myfile.stanza` program and calls the C compiler with the additional `-lmylib` flag to produce the `myprogram` executable.

```
stanza myfile.stanza -ccflags -lmylib -o myprogram
```

Note that to provide multiple flags to the C compiler, the flags must be quoted.

```
stanza myfile.stanza -ccflags -lmylib1 -lmylib2 -o myprogram
```

Target Platform Settings

By default, Stanza generates code appropriate for the platform that you specified in the call to `stanza install`. If you wish to generate code appropriate for a different platform, then you can override the platform using the `-platform` flag.

The following generates the assembly file `myprogram.s` appropriate for the Windows platform.

```
stanza myfile.stanza -s myprogram.s -platform windows
```

12.2 The When Expression

The `when` expression provides a convenient syntax for very short if expressions. The following

```
val name =
   if meerkat? : "Timon"
   else : "Pumbaa"
```

assigns the string "Timon" to `name` if `meerkat?` is `true`, otherwise it assigns "Pumbaa". It can be equivalently written as

```
val name = "Timon" when meerkat? else "Pumbaa"
```

In general, the form

```
a when c else b
```

is equivalent to the if expression

```
if c : a
else : b
```

Optional Else Branch

You may also leave off the `else` branch, in which case

```
a when c
```

is equivalent to the if expression

```
if c : a
```

This form is often convenient if you want to call a function only when some condition is `true`.

```
press(button) when action == "press"
```

The when expression is another example of a convenience construct implemented as a macro.

12.3 The Where Expression

The `where` expression provides a convenient syntax for pulling out short definitions from complicated expressions. The following code

```
println("They call me Mr. %_" % [name]) where :
   val name = "Pig!" when angry? else "Pumbaa."
```

first defines `name`, and then prints the message. It is equivalent to

```
let :
   val name = "Pig!" when angry? else "Pumbaa."
   println("They call me Mr. %_" % [name])
```

The where expression is also implemented as a macro. As you can see, Stanza's core library makes heavy use of macros.

12.4 The Switch Expression

The `switch` expression provides a convenient syntax for choosing amongst a number of nested `if` branches. Here is an example of evaluating the first branch for which `empty?` evaluates to `true`.

```
switch empty? :
   a : println("List a is empty.")
   b : println("List b is empty.")
   head(c) : println("The head of list c is empty.")
   else : println("Nothing is empty.")
```

The above is equivalent to these nested `if` expressions.

```
if empty?(a) :
   println("List a is empty.")
```

```
else if empty?(b) :
   println("List b is empty.")
else if empty?(head(c)) :
   println("The head of list c is empty.")
else :
   println("Nothing is empty.")
```

If the `else` branch is omitted then a default `else` branch is provided that prints an error and causes the program to fail.

The `switch` construct is commonly used with an anonymous function as its predicate. Here is an example of using `switch` to evaluate different branches depending on the value of `x`.

```
switch {x == _} :
   0 : println("Sunday")
   1 : println("Monday")
   2 : println("Tuesday")
   3 : println("Wednesday")
   4 : println("Thursday")
   5 : println("Friday")
   6 : println("Saturday")
   else : println("Elseday")
```

12.5 More on Visibility

Package Qualified Identifiers

Suppose our main program makes use of the following definitions from an `animals` package.

```
public defstruct Dog
public defstruct Cat
public name (x:Dog|Cat) -> String
public sound (x:Dog|Cat) -> String
```

Package-qualified identifiers allow us to reference those definitions without having to import the `animals` package. Here is a `main` function written using package-qualified identifiers and without importing `animals`.

```
defpackage animal-main :
   import core

defn main () :
   val d = animals/Dog("Shadow")
   val c = animals/Cat("Sassy")
   println("My dog %_ goes %_!" % [animals/name(d), animals/sound(d)])
   println("My cat %_ goes %_!" % [animals/name(c), animals/sound(c)])
```

In general, a package qualified identifier is any identifier that contains the `'/'` character. The characters after the last occurrence of the `'/'` form the name of the definition being referenced. The characters before the last occurrence form the name of the package containing the definition being referenced. For example, the following identifier

```
stanza/compiler/type/FunctionType
```

refers to the `FunctionType` definition in the `stanza/compiler/type` package.

Package-qualified identifiers are mostly used by macro writers. Macros should expand into references to package-qualified identifiers to prevent users from having to explicitly import the runtime libraries that the macros depend upon.

Top Level Identifiers

Identifiers whose only occurrence of the `'/'` character is at the beginning of the identifier are called *top-level* identifiers. For example, `/sound` and `/name` are top-level identifiers.

Top level identifiers are used to refer to a definition that is visible from the top most scope in the current package. It is used to refer to a top-level definition when its actual name has been shadowed by a local definition.

For example, the following

```
defn main () :
   val s = "Hello"
   val length = 42
   println(length(s))
```

fails to compile with the error

```
Value length of type Int cannot be called as a function.
```

This is because `length` refers to the value `42`, *not* the function that returns the length of a string. We can get around this either by renaming the `length` value to something else, or by using a top-level identifier to refer to the length function.

```
defn main () :
   val s = "Hello"
   val length = 42
   println(/length(s))
```

Protected Visibility

In addition to public and private visibilities, Stanza supports one last visibility setting: the protected visibility. A definition with protected visibility *can* be referred to from other packages, but they can *only* be referred to using package-qualified identifiers.

Suppose we have an `animals` package containing the following definitions.

```
public defstruct Dog
public defstruct Cat
public name (x:Dog|Cat) -> String
protected sound (x:Dog|Cat) -> String
```

And we will import the `animals` package into our `animals-main` package.

```
defpackage animals-main :
   import animals

defn main () :
   val d = Dog("Shadow")
   val c = Cat("Sassy")
   name(d)
   animals/sound(c)
```

All of the public definitions in `animals` can be directly referred to in `animals-main` after they have been imported, *but* the protected function `sound` must be package-qualified.

Protected definitions are most often used by macro writers. Often, a macro simply expands into a decorated call to a helper function. We want to encourage users to use the macro form, and *not* call the helper function directly. By annotating the macro with the protected visibility we make it unlikely for users to accidentally call the helper function.