ZoomBA

Programming Language

A Micro-Language For JVMs
Version : 0.3 (2025-Jan)

By : zoomba-lang.org

Contents

Preface

ZoomBA is motivated by Apache Jexl project. Scripting based on JVM is necessary, and the existing scripting languages were not simply not effective enough for glue purposes. We believed that world needed a scripting language for JVM, but at the same time, it's purpose should be isolated scripting, and with very limited scope. And, ZoomBA was born.

A brief History

Need of the hour was a language where one can write business logic for data processing freely.

There was no language available which lets intermingle with Java POJO.

Worse still - one can not write business logic freely using Java, the whole spring MVC is a challenge. Given almost all of modern enterprise application are written using Java, it is impossible to avoid Java and write Enterprise code : in many cases you would need to call appropriate Java methods to automate APIs.

Thus, one really needs a JVM scripting language that can freely call and act on POJOs, the inherent historical baggage of JavaScript disables Nashorn from being a suitable choice. But with Nashorn out of default scripting Engine now, scripting on JVM looks bleak.

A micro language like ZoomBA fills up the gap. When we decided to invent ZoomBA, we imagined a language where one can freely code data manipulation logic, as a gluing between multiple web service API calls. It was built to connect the dots, dots being plethora of micro-services.

About The Language

It is an interpreted language. It is asymptotically as fast as Python , with a general lag of 200 ms of reading and parsing files, where native python is faster. After that the speed is the same.

It is modern, unlike MVEL, it has JVM flavour in it, it is essentially very short Kotlin or Scala. Type safety is sacrificed for it being a configuration language. A JSON is a valid ZoomBA literal, thus making data transfer natural.

It is a multi-paradigm language. It supports functionals (i.e. functions taking functions as input or returning a function as output) out of the box, and every function by design can take function as input. There are tons of in built methods which uses these functionals. One massive departure from functional style however, the lack of *immutable* containers. ZoomBA has none. It altogether avoids the problem by tighter scoping of variable.

It also supports prototype based OOP. There is no reason to do object modelling in ZoomBA, plain old Java/Scala works well. ZoomBA is more like a band-aid language, it has multiple features copied from many other languages. The heavy use of __xxx__ literals, and *def* is out and out python.

The space and tab debate is very religious, and hence ZoomBA is decidedly " blocked " : Brace yourself. Pick tab/space to indent - none bothers here. You can use ";" to separate statements in a line. Lines are statements.

The Philosophy of the Logo

The *thunder symbol* in the logo signifies what precisely it was meant to do, fast forward time
to prod. It is apparent that ZoomBA is a slow language, while the current implementation is
concerned, but then, for business processes, the flexibility is the key. For example, ZoomBA
can, and does replace Spring's DI system, works as glue between any database command line
interfaces and JVM (thereby not requiring any library support), act as configurations.

Thanks

And finally, all of these pages were typed using TexShop-64. So, thanks to Richard Koch, Max
Horn Dirk Olmes. For MacTex, thanks MacTex. You guys are great! Thanks to Apple for
creating such a beautiful system to work on. Steve, we love you. RIP.

Thanks to Gabriel Hjort Blindell - for the beautiful style file he created which can be found
here. Gabriel, thanks a ton. The ZoomBA logo was created using freelogoservices. Keep up the
amazing work.

ABOUT ZOOMBA

A philosophy is needed to guide the design of any system. ZoomBA is not much different. Here we discuss the rationale behind the language, and showcase how it is distinct from its first cousin Java. While has similarity to Scala should not come as a surprise, that is an example of convergent evolution. But just as every modern animal is highly evolved, and there is really no general purpose animal, we sincerely believe there is no real general purpose language. All languages are special purpose, some special purposes may seem generic in nature. However, *ZoomBA* preaches a different philosophy than *scala*, *kotlin* and even *python*.

1.1 ZOOMBA PHILOSOPHY

To begin with, our own experience in Industry is aptly summarised by Ryan Dahl in here , the creator of Node.js :

I hate almost all software. It's unnecessary and complicated at almost every layer. At best I can congratulate someone for quickly and simply solving a problem on top of the shit that they are given. The only software that I like is one that I can easily understand and solves my problems. The amount of complexity I'm willing to tolerate is proportional to the size of the problem being solved...(continued)... Those of you who still find it enjoyable to learn the details of, say, a programming language - being able to happily recite off if NaN equals or does not equal null - you just don't yet understand how utterly fucked the whole thing is. If you think it would be cute to align all of the equals signs in your code, if you spend time configuring your window manager or editor, if put unicode check marks in your test runner, if you add unnecessary hierarchies in your code directories, if you are doing anything beyond just solving the problem - you don't understand how fucked the whole thing is. No one gives a fuck about the glib object model. The only thing that matters in software is the experience of the user. - *Ryan Dahl*

Thus, ZoomBA comes with it's tenets, which are :

1. Reduce the number of lines in the code; *Principle of Percimony*
2. If possible, in every line, reduce the number of characters;
3. Get out of the cycle of bugs and fixes by writing scientific and provable code *Provability* (see Minimum Description length).
4. The final statement is : Good code is only once written, and then forever forgotten, i.e. : limiting to 0 maintenance. *Maintainability is lack of need to Maintain*

That is,

> *To boldly go where no developer has gone before - attaining Nirvana in terms of coding*

Thus we made ZoomBA so that a language exists with it's full focus on *Business Process Automation & Validation*, not on commercial fads that sucks the profit out of business. Hence it has one singular focus in mind : *brevity* but not at the cost of maintainability. What can be done with 10 people, in 10 days, get it done in 1 day by one person.

It is being demonstrated by firms adapting to it.

1.2 DESIGN FEATURES

Here is the important list of features, which make ZoomBA a first choice of the business developers and software testers, alike.

1.2.1 *ZoomBA is Embeddable*

ZoomBA scripts are easy to be invoked as stand alone scripts, also from within java code, thus making integration of external logic into proper code base easy. Thus Java code can call ZoomBA scripts very easily, and all of ZoomBA functionality is programmatically accessible by Java caller code. This makes it distinct from Scala, where it is almost impossible to call scala code from Java. Lots of code of how to call ZoomBA can be found in the test directory. Many scripts are there as samples in samples folder. In chapter 12 we would showcase how to embed ZoomBA in Java code.

1.2.2 *Interpreted by JVM*

ZoomBA is interpreted by a program written in Java, and uses Java runtime. This means that ZoomBA and Java have a common runtime platform. You can easily move from Java to ZoomBA and vice versa.

1.2.3 *ZoomBA can Execute any Java Code*

ZoomBA enables you to use all the classes of the Java SDK's in ZoomBA, and also your own, custom Java classes, or your favourite Java open source projects. There are trivial ways to load a jar from path directly from ZoomBA script, and then loading a class as as trivial as importing the class.

1.2.4 *ZoomBA can be used Functionally*

ZoomBA is also a functional language in the sense that every function is a value and because every value is an object so ultimately every function is an object. Functions are first class citizens.

ZoomBA provides a lightweight syntax for defining anonymous functions, it supports higher-order functions, it allows functions to be nested, and supports currying and Closures. It also supports Operator Overloading.

1.2.5 *ZoomBA is Dynamically Typed*

ZoomBA, unlike statically typed languages, does not expect you to provide type information. You don't have to specify a type in most cases, and you certainly don't have to repeat it.

1.2.6 *ZoomBA Vs Java*

Most of the time one can treat ZoomBA as a very tiny shorthand for incredibly fast programming using JVM. However, ZoomBA has a set of features, which completely differs from Java. Some of these are:

1. All types are objects. An assignment of the form $x = 1$ makes actually an Integer object in JVM, not a int.
2. Type inference : by that one does not need to cast a variable to a type before accessing its functions. This makes reflective calls intuitive, as $a.b.f()$ can be written as $a['b'].f()$ and thus, the value of $'b'$ itself can come from another variable. Objects are treated like property buckets, as in Dictionaries.
3. Nested Functions : functions can be nested :

```
1  def parent_function( a ) {
2      def child_function (b) {
3          // child can use parents args: happily.
4          a + b
5      }
6      if ( a != null ) {
7          return child_function
8      }
9  }
```

4. Functions are objects, as the above example aptly shows, they can be returned, and assigned :

```
1  fp = parent_function(10)
2  r = fp(32 ) // result will be 42.
```

5. Closures : the above example demonstrates the closure using partial functions, and this is something that is syntactically new to Java land.

1.3 SETTING UP ENVIRONMENT

1.3.1 *Installing Java Run Time*

You need to install Java runtime 1.9 (64 bit) at minimum (It supports all the way up to Java 21+) To test whether or not you have successfully installed it try this in your command prompt :

```
 $ java --version
openjdk 21.0.1 2023-10-17 LTS
OpenJDK Runtime Environment Temurin-21.0.1+12 (build 21.0.1+12-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.1+12 (build 21.0.1+12-LTS, mixed mode, sharing)
```

1.3.2 *Download ZoomBA one jar*

From 0.3-SNAPSHOT onward ZoomBA got split into production snapshot and debuggable instrumented snapshot. Snapshot : prod-snapshot Debuggable, Instrumented Snapshot : instrumented-snapshot

Another way is to simply run the script from here:

```
 $  bash -c "$(curl -fsSL https://gitlab.com/non.est.sacra/zoomba/-/raw/master/install.sh)"
```

Once we run it follow the instruction it spews.

1.3.3 *Add to Path*

If you are using nix platform, then you should create an alias :

```
alias zmb="java -jar ZoomBA.lang.core-*-onejar.jar"
```

in your .login file.

If you are using Windows, then you should create a batch file that looks like this:

```
@echo off
rem init.cmd: to be run on every login
doskey zmb=java -jar ZoomBA.lang.core-*-onejar.jar  $*
```

and then follow the steps as shown here.

1.3.4 *Test Setup*

Open a command prompt, and type :

```
$ zmb # the alias
//h brings this help
//h key_word : shows help about the keyword
//q quits REPL. In debug mode runs till next BreakPoint
//v shows variables
//r loads and runs a script from REPL
Enjoy ZoomBA...(0.3-SNAPSHOT-2025-Jan-23 08:35) running on JRE (21.0.1)
(zoomba)
```

Notice the timestamp. That uniquely specifies when the binary was built.

It should produce the prompt of REPL of (ZoomBA).

1.3.5 *Maven Setup for Java Integration*

In the dependency section (latest release is 0.2) :

```
<dependency>
  <groupId>org.zoomba-lang</groupId>
  <artifactId>zoomba.lang.core</artifactId>
  <version>0.2</version> <!-- or 0.3-SNAPSHOT -->
</dependency>
```

That should immediately make your project a ZoomBA supported one.

1.3.6 *Setting up Editors*

IDEs are good - and that is why we have minimal editor support, Sublime Text is my favourite one. You also have access to the syntax highlight file for zoomba and a specially made theme for zoomba editing - (ES) both of them can be found : here. There is also a vim syntax file. If you use them with your sublime text editor - then typical zoomba script file looks like this :

To include for vim :

Create these two files :

```
$HOME/.vim/ftdetect/zmb.vim
$HOME/.vim/syntax/zmb.vim
```

For most nix systems it would be same as :
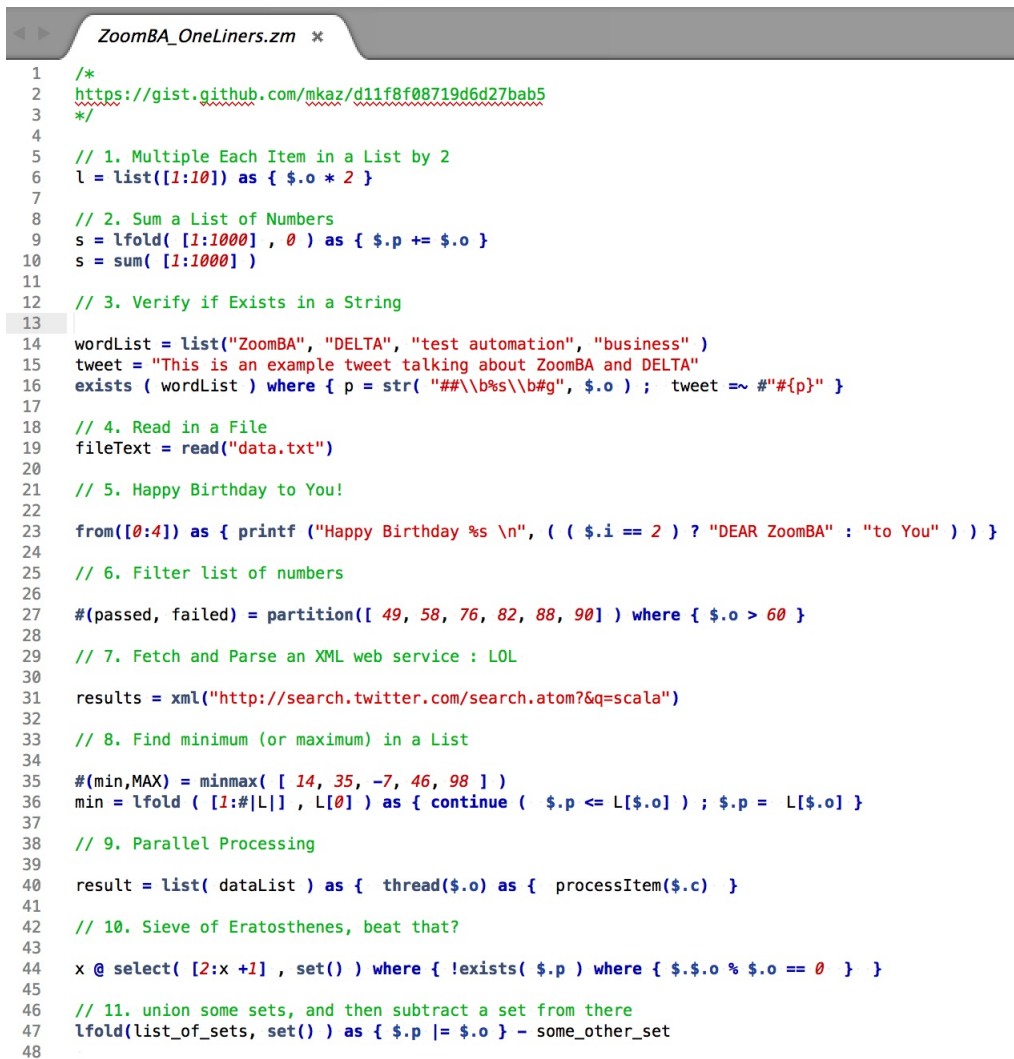
```
mkdir -p ~/.vim/ftdetect/
touch ~/.vim/ftdetect/zmb.vim
touch ~/.vim/syntax/zmb.vim
```

Now on the $HOME/.vim/ftdetect/zmb.vim file, put this line :

```
autocmd BufRead,BufNewFile *.zmb,*.z, *.zm  set filetype=zmb
```

Note that you should not have blanks between commas. And then, copy the content of the vim syntax file here in the $HOME/.vim/syntax/zmb.vim file as is.

If everything is fine, you can now open zmb scripts in vim!

```
1   /*
2   https://gist.github.com/mkaz/d11f8f08719d6d27bab5
3   */
4
5   // 1. Multiple Each Item in a List by 2
6   l = list([1:10]) as { $.o * 2 }
7
8   // 2. Sum a List of Numbers
9   s = lfold( [1:1000] , 0 ) as { $.p += $.o }
10  s = sum( [1:1000] )
11
12  // 3. Verify if Exists in a String
13
14  wordList = list("ZoomBA", "DELTA", "test automation", "business" )
15  tweet = "This is an example tweet talking about ZoomBA and DELTA"
16  exists ( wordList ) where { p = str( "##\\b%s\\b#g", $.o ) ;  tweet =~ #"#{p}" }
17
18  // 4. Read in a File
19  fileText = read("data.txt")
20
21  // 5. Happy Birthday to You!
22
23  from([0:4]) as { printf ("Happy Birthday %s \n", ( ( $.i == 2 ) ? "DEAR ZoomBA" : "to You" ) ) }
24
25  // 6. Filter list of numbers
26
27  #(passed, failed) = partition([ 49, 58, 76, 82, 88, 90] ) where { $.o > 60 }
28
29  // 7. Fetch and Parse an XML web service : LOL
30
31  results = xml("http://search.twitter.com/search.atom?&q=scala")
32
33  // 8. Find minimum (or maximum) in a List
34
35  #(min,MAX) = minmax( [ 14, 35, -7, 46, 98 ] )
36  min = lfold ( [1:#|L|] , L[0] ) as { continue (  $.p <= L[$.o] ) ; $.p =  L[$.o] }
37
38  // 9. Parallel Processing
39
40  result = list( dataList ) as {  thread($.o) as {  processItem($.c)  }
41
42  // 10. Sieve of Eratosthenes, beat that?
43
44  x @ select( [2:x +1] , set() ) where { !exists( $.p ) where { $.$.o % $.o == 0  }  }
45
46  // 11. union some sets, and then subtract a set from there
47  lfold(list_of_sets, set() ) as { $.p |= $.o } - some_other_set
48
```

FIGURE 1.1 – Using Sublime Text.

1.3.7 *The Proverbial "Hello, World"*

In any editor of your choice, save this line in a file 'hello.zmb' :

```
1   println('Hello, World!')
```

and go to the command prompt, and run :

```
$ zmb hello.zmb
Hello, World
$
```

and that would be the proverbial starting code.

```
ZoomBA_OneLiners.zm (/Codes/zoomba/zoomba99) – VIM1

ZoomBA_OneLiners.zm
 1 /*
 2 https://gist.github.com/mkaz/d11f8f08719d6d27bab5
 3 */
 4
 5 // 1. Multiple Each Item in a List by 2
 6 l = list([1:10]) as { $.o * 2 }
 7
 8 // 2. Sum a List of Numbers
 9 s = lfold( [1:1000] , 0 ) as { $.p += $.o }
10 s = sum( [1:1000] )
11
12 // 3. Verify if Exists in a String
13
14 wordList = list("ZoomBA", "DELTA", "test automation", "business" )
15 tweet = "This is an example tweet talking about ZoomBA and DELTA"
16 exists ( wordList ) where { p = str( "##\\b%s\\b#g", $.o ) ;  tweet =~ #"#{p}" }
17
18 // 4. Read in a File
19 fileText = read("data.txt")
20
21 // 5. Happy Birthday to You!
22
23 from([0:4]) as { printf ("Happy Birthday %s \n", ( ( $.i == 2 ) ? "DEAR ZoomBA" : "to You" ) ) }
24
25 // 6. Filter list of numbers
26
27 #(passed, failed) = partition([ 49, 58, 76, 82, 88, 90] ) where { $.o > 60 }
28
29 // 7. Fetch and Parse an XML web service : LOL
30
31 results = xml("http://search.twitter.com/search.atom?&q=scala")
32
33 // 8. Find minimum (or maximum) in a List
34
35 #(min,MAX) = minmax( [ 14, 35, -7, 46, 98 ] )
36 min = lfold ( [1:#|L|] , L[0] ) as { continue (  $.p <= L[$.o] ) ; $.p =  L[$.o] }
37
38 // 9. Parallel Processing
39
40 result = list( dataList ) as {  thread($.o) as {  processItem($.c)  }
41
42 // 10. Sieve of Eratosthenes, beat that?
43
44 x @ select( [2:x +1] , set() ) where { !exists( $.p ) where { $.$.o % $.o == 0  }  }
45
46 // 11. union some sets, and then subtract a set from there
47 lfold(list_of_sets, set() ) as { $.p |= $.o } - some_other_set
48
```

FIGURE 1.2 – Using Vim (MacVim).

ZoomBA Syntax in 3 Minutes

With some understanding on any of C,C++, Java, Kotlin, Scala, Python or JavaScript then it will be very easy for you to learn ZoomBA. The biggest syntactic difference between ZoomBA and other languages is that the ';' statement end character is optional. When we consider a ZoomBA program it can be defined as a collection of functions that communicate via taking input from the previous and producing output which is to be taken as input to another, next in line.

2.1 BUILDING BLOCKS

2.1.1 *Identifiers & Reserved Keywords*

ZoomBA is case-sensitive, which means identifier Hello and hello would have different meaning. All ZoomBA components require names. Names used for objects, classes, variables and methods are called identifiers. A keyword cannot be used as an identifier and identifiers are case-sensitive.

1. Fully Reserved : you can not use these identifiers as variables.

    if, else, for, while, where, size, empty, def, isa , null, type

2. Partially Reserved : you should not use these as variables.

    atomic, xml, thread, system, lines, list, dict, type , println, read, send, random, hash , minmax, fold, index, rindex, int, float, INT, FOAT, bool, num, str.

3. Specially Reserved : you should not use these as variables.

    @,$, anything with '__'.

2.1.2 *Assignments*

Most basic syntax of ZoomBA is, like any other language : assignment.

```
1   a = 1 //  assigns local variable a to Integer 1
2   b = 1.0 //assigns local variable a to b float 1.0
3   c = 'Hello, ZoomBA' // assigns local variable c to String 'Hello, ZoomBA'
4   d = "Hello, ZoomBA" // same, strings are either single or double quoted
5   /*
6   assigns the *then* value of a to e,
7   subsequent change in a wont reflect in e
8   */
9   e = a
```

2.1.3   Comments

See from the previous subsection "//" used as line comments. Along with the multiline comment "/*" with "*/" :

```
1   // this is line comment
2   /* this is multiline
3    comment, for large and long lines!
4   */
```

2.1.4   Basic Types

Basic types are :

```
1    a = 1 // Integer
2    s = 'Hello, ZoomBA' // String
3    c = _'a' // Character
4    d = 1.0 // Double
5    l = 1l // long integer
6    L = 1L // Large Integer ( BigInteger )
7    D = 1.0D // Large Floating  ( BigDecimal )
8    tt = true // boolean
9    tf = false // boolean
10   null_literal = null // special null type
```

2.1.5   Multiple Assignment

ZoomBA supports multiple assignment. It has various usage:

```
1   a = 1 // Integer
2   b = 1.0 //  float
3   c = 'Hello, ZoomBA' // String
4   // instead, do this straight :
5   #(a,b,c ) = [ 1 , 1.0 , 'Hello, ZoomBA' ]
```

2.2 OPERATORS

2.2.1 *Arithmetic*

```
1   a = 1 + 1 // addition : a <- 2
2   z = 1 - 1 // subtraction : z <- 0
3   m = 2 * 3 // multiply : m <- 6
4   d = 3.0 / 2.0 // divide d <- 1.5
5   #(q,r) = 100 /% 7 // div mod q=14, r=2
6   x = 2 ** 10 // Exponentiation x <- 1024
7   y = -x // negation, y <- -1024
8   r = 3 % 2 // modulo, r <- 1
9   a += 1 // increment and assign
10  z -= 1 // decrement and assign
11  x *= 2 // multiply and assign
12  y /= 3 // divide and assign
13  a /? b // does a divide b?
14  2 /? 4 // true
15  5 /? 13 // false
```

2.2.2 *Logical*

```
1   o = true || true // true , or operator
2   a = true && false // false , and operator
3   defined_a = is(a) // true if a is defined, false otherwise
4   o = (10 != 20 ) // not, true
5   o = ! ( 10 = 20 ) // true
6   o = (10 == 20) // false
7   o = ( true ^ false ) // true : xor operator for boolean
8   o = ( true ^ true ) // false : xor operator for boolean
```

2.2.3 *BitWise*

```
1   o = 1 | 2  // o = 3 , bit wise  or operator
2   o = 1 & 2  // o = 0 , bit wise  and operator
3   o = 1 ^ 2  // o = 3 , bit wise xor operator
4   // now the mutable do and assign
5   x = 1
6   x |= 2  // x = 3 ,  bit wise  or operator
7   x &=  2  // x = 2 , bit wise  and operator
8   x ^= 2  // x = 0 , bit wise xor operator
```

2.2.4 *Comparison*

```
1   t = 10 < 20 // true, less than
2   f = 10 > 20 // false, greater then
3   t = 10 <= 10 // true, less than or equal to
4   t = 10 >= 10 // true, greater then or equal to
5   t = ( 10 == 10 ) // true, equal to
6   f = ( 10 != 10 ) // false, not equal to
7   t = ( 10 = 10.0 )  // true, they are same
8   f = ( 10 === 10.0 )  // false, they have same value but not type
9   t = ( 10 .= 10 ) // true, this is underlying equals for the runtime
```

2.2.5   Ternary

```
1  // basic ternary
2  min = a < b ? a : b // general form (expression)?option1:option2
3  // try fixing null with it
4  non_null = a == null? b : a
5  // or use the null coalescing operator
6  non_null = a ?? b
7  // same can be used as definition coalescing
8  defined = is(a) ? a : b
9  //same as above
10 defined = a ?? b
```

2.3   CONDITIONS

People coming from any other language would find them trivial.

2.3.1   If

```
1  x = 10
2  if ( x < 100 ){
3     x = x**2
4  }
5  println(x) // printlns back x to standard output : 100
```

2.3.2   Else

```
1  x = 1000
2  if ( x < 100 ){
3     x = x**2
4  }else{
5     x = x/10
6  }
7  println(x) // printlns back x to standard output : 100
```

2.3.3   Else If

```
1  x = 100
2  if ( x < 10 ){
3     x = x**2
4  } else if( x > 80  ){
5     x = x/10
6  } else {
7     x = x/100
8  }
9  println(x) // printlns back x to standard output : 10
```

2.4   LOOPS

2.4.1   While

```
1  i = 0
2  while ( i < 42 ){
3    println(i)
4    i += 1
5  }
```

2.4.2 *For*

For can iterate over any iterable, in short, it can iterate over string, any collection of objects, a list, a dictionary or a range. That is the first type of for :

```
1  for ( i : [0:42] ){ // [a:b] is a range type
2    println(i)
3  }
```

The result is the same as the while loop. A standard way, from C/C++/Java is to println the same as in :

```
1  for ( i = 0 ; i < 42 ; i+= 1  ){ // assignment, condition, after
2    println(i)
3  }
```

There is support for implicit variable for a loop. This variable is called $:

```
1  for ( [0:42] ){ // [a:b] is a range type
2    println($)
3  }
```

Just like in Python, we also have a tuple passed as parameter, first one being the index, second one being the object in question for the iteration:

```
1  for ( idx, obj : [10:13] ){ // [a:b] is a range type
2    printf("Index is %d and Object is %d %n", idx, obj )
3  }
```

Similar way, we have implicit for index, and object :

```
1  for ( [10:13] ){ // [a:b] is a range type
2    printf("Index is %d and Object is %d %n", _$, $ )
3  }
```

Scoping rules are simple. *foreach* is effectively a closed function call. Thus if the iterator variables references are there outside scope, they gets replaced inside, it is hidden. Once outside the for loop, they can be accessed again.

2.5 FUNCTIONS

2.5.1 *Defining*

Functions are defined using the *def* keyword. And they can be assigned to variables, if one may wish to.

```
1  def count_to_num(n){
2      for ( i : [0:n] ){
3          println(i)
4      }
5  }
6  // just assign it
7  fp = count_to_num
```

One can obviously return a value from function :

```
1  def say_something(word){
2      return ( "Hello " + word )
3  }
```

But it is overkill. The last executed statement of a function is the return value, thus :

```
1  def say_something(word){
2      ( "Hello " + word )  // returns it
3  }
```

2.5.2   *Function Calling*

Calling a function is trivial :

```
1  // calls the function with parameter
2  count_to_num(42)
3  // calls the function at the variable with parameter
4  fp(42)
5  // calls and assigns the return value
6  greeting = say_something("Homo Erectus!" )
```

2.5.3   *Global Variables*

As you would be knowing that the functions create their local scope, so if you want to use variables - they must be defined in global scope. Every external variable is readonly to the local scope, but to println to it, use global variables, which starts with $.

```
1   $a = 0
2   x = 0
3   def use_var(){
4       $a = 42
5       println(a) // prints 42
6       println(x) // prints 0, can access global
7       x = 42
8       println(x) // prints 42
9   }
10  // call the method
11  use_var()
12  println($a) // global a, prints 42
13  println(x) // local x, prints 0 still
```

The result would be :

```
42
0
42
```

```
42
0
```

2.6 ANONYMOUS FUNCTION AS A FUNCTION PARAMETER

A basic idea about what it is can be found here. As most of the Utility functions use a specific type of anonymous function, which is nick-named as "Anonymous Parameter" to a utility function.

2.6.1 *Why it is needed?*

Consider a simple problem of creating a list from an existing one, by modifying individual elements in some way. This comes under map , but the idea can be shared much simply:

```
1  l = list()
2  for ( x : [0:n] ){
3    l.add ( x * x )
4  }
5  return l // yep, here...
```

Observe that the block inside the *for loop* takes minimal two parameters, in case we println it like this :

```
1  // return is not really required, last executed line is return
2  def map(x){ x * x }
3  l = list()
4  for ( x : [0:n] ){
5    l.add ( map(x) )
6  }
7  return l // more general
```

Observe now that we can now create another function, lets call it list_from_list :

```
1  def map(x){ x * x }
2  def list_from_list(fp, old_list)
3   l = list()
4   for ( x : old_list ){
5     // use the function *reference* which was passed
6     l.add( fp(x) )
7   }
8   return l // great...
9  }
10 list_from_list(map,[0:n]) // same as previous 2 implementations
```

The same can be achieved in a much sorter way, with this :

```
1  list([0:n]) as { $.item * $.item  }
2  // if you like short-symbols
3  list([0:n]) -> { $.o  * *  2  }
```

The curious block construct after the list function arguments is called anonymous (function) parameter, which takes over the map function. The loop stays implicit, and the result is equivalent from the other 3 examples.

This reads as follows : *create a list from arguments using the block as ...*

The explanation is as follows. For an anonymous function parameter, there are 3 implicit guaranteed arguments :

1. $ –> signifies the iteration object. This has the fields :
2. $.o, $.item –> Signifies the item of the collection , we call it the *ITEM*
3. $.c, $.context –> The context, or the collection itself , we call it the *CONTEXT*
4. $.i, $.index –> The index of the item in the collection, we call it the *ID* of iteration
5. Another case to case parameter is : $.p, $.partial –> Signifies the partial result of the processing , we call it *PARTIAL*

The ideas can be simply put into examples by taking this particular case :

```
1  l = list([1:5]) as {
2      printf('item : %s , index : %s , partial %s\n', $.o,$.i,$.p)
3  $.o*$.o  }
```

which produces this :

```
item : 1 , index : 0 , partial []
item : 2 , index : 1 , partial [1]
item : 3 , index : 2 , partial [1, 4]
item : 4 , index : 3 , partial [1, 4, 9]
```

Observe that partial is the partial result of the iteration, which would finally yield to the final result.

2.6.2 *Some Use Cases*

The data structure section would showcase some use cases. But we would use a utility function to showcase the use of this anonymous function. Suppose there is this function *minmax()* which takes a collection and returns the (min,max) tuple. In short :

```
1  #(min,max) = minmax(1,10,-1,2,4,11)
2  println(min) // prints -1
3  println(max) // prints 11
```

But now, suppose we want to find the minimum and maximum by length of a list of strings. To do so, there has to be a way to pass the comparison done by length. That is easy :

```
1  #(min,max) = minmax( "" , "aa" , "abc" , "aa", "bbbbb" ) as { size($.o ) }
2  println(min) // prints empty string
3  println(max) // prints bbbbb
```

2.7 AVAILABLE DATA STRUCTURES

2.7.1 *Range*

A range is basically an iterable, with start and end separated by colon : $[a : b]$. We already have seen this in action. "a" is inclusive while "b" is exclusive, this was designed the standard for loop in mind. There can also be an optional spacing parameter "s", thus the range type in general is $[a : b : s]$, as described below:

```
1  /*
2    when r = [a:b:s]
3    the equivalent for loop is :
4    for ( i = a ; i < b ; i+= s ){
5      ... body now ...
6    }
7  */
8  r1 = [0:10] // a range from 0 to 9 with default spacing 1
9  //a range from 1 to 9 with spacing 2
10 r2 = [1:10:2] //1,3,5,7,9
```

2.7.2 List

To solve the problem of adding and deleting item from an array, list were invented.

```
1  fll = [0,1,2,3,4] // fixed length list (not an array, but a list)
2  fll += 50 // no modification, Invalid Arithmetic Logical Operation error.
3  l = list ( 0,1,2,3 ) // a list
4  l += 10 // now the list is : 0,1,2,3,10
5  l -= 0 // now the list is : 1,2,3,10
6  x = l[0] // x is 1 now
7  l[1] = 100 // now the list is : 1,100,3,10
```

2.7.2.1 Regular List as Stack and Queue

```
1  l = list ( 0,1,2,3 ) // a list
2  // stack
3  l.peek() // 0 - but does not remove
4  l.pop() // 0
5  l.push(42) // [ 42,1,2,3 ]
6  // now as queue
7  l.removeLast() // 3
8  l.add(22) // [ 42,1,2,22 ]
```

2.7.2.2 Sorted List

```
1  sl = slist (2,1,4,10,5) // [ 1,2,4,5,10 ]
2  sl.first // 1
3  sl.last // 10
```

Adding to a sorted list is $\Theta(log(n))$ while indexing is $\Theta(n)$ where n is the size of the list. Finding item and removing them are constant time operations.

2.7.3 Set

A set is a collection of elements such that the elements do not repeat. Thus :

```
1  // now the set is : 0,1,2,3
2  s = set ( 0,1,2,3,1,2,3 ) // a set
3  s += 5 // now the set is : 0,1,2,3,5
4  s -= 0 // now the set is : 1,2,3
```

2.7.3.1   Ordered Set

In Set, elements are not present by their insertion order. To fix that problem, *oset()* exists :

```
1  os = oset(12,1,2,3, 10) // if we now print
2  println( os ) // you would find them in the same order...
```

It is a wrapper over LinkedHashSet.

2.7.3.2   Sorted Set

In Set, elements are not sorted by their values. To fix that problem, *sset()* exists :

```
1  ss = sset(12,1,2,3, 10) // if we now print
2  println( ss ) // you would find them in { 1,2,3,10,12 }
```

It is a wrapper over TreeSet.

2.7.4   *Dict*

A dictionary is a collection (a list of) (key,value) pairs. The keys are unique, they are the *keySet*(). Here is how one defines a dict:

```
1  d1 = { 'a' : 1 , 'b' : 2 } // a dictionary
2  d2 = dict( ['a','b'] , [1,2] ) // same dictionary
3  x = d1['a'] // x is 1
4  x = d1.a // x is 1
5  d1.a = 10 // now d1['a'] --> 10
```

2.7.4.1   Ordered Map

In Map, tuples are not present by their insertion order. To fix that problem, *odict()* exists :

```
1  od = odict() //
2  od[100] = 1
3  od[2] = 2
4  od[199] = 42
5  println( od ) // you would find them in the same order...
```

It is a wrapper over LinkedHashMap.

2.7.4.2   Sorted Dict

In Set, elements are not sorted by their keys. To fix that problem, *sdict()* exists :

```
1  sd = sdict()
2  sd[100] = 1
3  sd[10] = 2
4  sd[-10] = 42
5  println( sd ) // you would find them in in sorted key order
```

It is a wrapper over TreeMap.

2.7.5   *Multi Set, Group*

Many a times, it is necessary to group sets out of a collection. For example, consider the problem of how many unique integers are there in a collection, along with their frequencies. To solve this, ZoomBA has *mset()* :

```
1  l = [ 1,3,4,3,2,1,1,5]
2  ms = mset(l)
3  {1=3, 2=1, 3=2, 4=1, 5=1} // MultiSet
```

What it is depicting is, key 1 has occurred 3 times, 2 occurred once, etc etc This seats in somewhere between set and a collection, a frequency bucket. *mset()* and it's close cousin *group()* has other usage, which we will discuss later when we discuss comprehensions on collections.

2.7.6   *Mutability*

Data structures are not generally mutable in ZoomBA. What does that mean?

```
1  x = [1,2,3]
2  x + 10 // add some
3  println(x) // @[1,2,3] : x did not change
```

Thus, a variables value never gets changed, unless someone assigns back something to it. The only way a variable state can get change is through assignment. This is known as Immutability. See more of a discussion here.

The mutable additive operators : "+=" and "-=" are the ones which do not follow it, because they are also assignment operators. If the object is part of collection types, they would modify the left hand object itself, instead of creating a new instance of the object. Thus :

```
1   a = [1,2,3]
2   a + 10 // creates a new array object
3   s = set(1,2,3)
4   s += 42 // simply : s.add(42)
5   s -= 1 // s.remove(1)
6   s ++= [2,3,4] // s.addAll( list(2,3,4) )
7   s \= [3,4] // s.removeAll( list(3,4) )
8   m = {1:2, 3:4}
9   m += {5:6} // m.putAll( {5:6} )
10  m -= 3 // m.remove(3)
```

Notice the use of "++=" and "˜" to depict the addAll and removeAll operations.

FORMAL CONSTRUCTS

Formalism is the last thing that stays in a software developers mind nowadays. This is a result of mismanaging expectation of how software and coding is taught and practiced in industry. The majority view in the Industry is :

Software is a form of art, and is not science at all.

It is easy to showcase that this is the prevailing feeling. When the last time people actually did any mathematics to reach any conclusion like - what would be the optimal object hierarchy size if any? What should be the optimal pool size for an application? How one can optimally transform the data?

Most of these cases, a bit of formal thinking solves a lot of the problems that can come due to bad design, if there was a design in the first place (for most parts, there are none). That is, because software is nothing but formally applied computer science, and industry lacks computer science. What can be done by 100 can also be done by 10 people, and no, they do not have to be really really smart, they only have to be understanding the underlying structure. In reality software is :

Software is to Applied Computation is what Architecture is to applied sciences, i.e. Engineering.

Unless you have an idea of the underlying mechanism, most won't work, and those who would luckily work - would simply fail in a slightly longer timeframe. Moreover, user does not care about brilliant design and optimal code, user care about getting his/her problem solved, fast. It is business who should care about how to do things optimally and repeatedly, otherwise, the business model won't sustain.

This section would be specifically to understand what sort of formalism from computation we can use in practice to develop and test software to aid business. Because in the end, profit and revenue runs the show.

3.1 CONDITIONAL BUILDING BLOCKS

3.1.1 *Formalism*

In a formal logic like FOPL, the statements can be made from the basic ingredients, *"there exist"* and *"for all"*. We need to of course put some logical stuff like *AND, OR*, and some arithmetic here and there, but that is what it is.

If we have noted down the *sorting problem* in chapter 2, we would have restated the problem as :

> *There does not exist an element which is less than that of the previous one.*

In mathematical logic, *"there exists"* becomes : $\exists$ and *"for all"* becomes $\forall$. and logical not is shown as $\neg$, So, the same formulation goes in : let

$$S = \{0, 1, 2, ..., size(a) - 1\} = \{x \in \mathbb{N} | x < size(a)\}$$

then :

$$i \in S; \forall\, i\neg\, (\exists\, a[i]\ s.t.\ a[i] < a[i-1])$$

And the precise formulation of what is a sorted array/list is done in the second order logic. The problem of programming and validation generally is expressive in terms of higher order logic.

3.1.2 Relation between "there exists" and "for all"

There is a nice dual relationship between there exists $\exists$ and $\forall$. Given we have the negation operation defined, $\forall$ and $\exists$ are interchangeable. Suppose we ask :

> *Are every element in the collection C greater than 0?*

$$\forall x \in C\,;\, x > 0\,?$$

This can be reformulated as the negation of:

> *Does there exist any element in the collection C less than or equal to 0?*

$$\exists x \in C\,;\, x \leq 0\,?$$

Hence, the transformation law is :

$$\forall x \in C\,;\, P(x)\,? \iff \neg(\exists x \in C\,;\, \neg P(x)\,)?$$

3.1.3 There Exist In Containers

To check, if some element is, in some sense exists inside a container (list, set, dict, array, heap) one needs to use the *IN* operator, which is @.

```
1   l = [1,2,3,4] // l is an array
2   in = 1 @ l // in is true
3   in = 10 @ l // in is false
4   d = { 'a' : 10 , 'b' : 20  }
5   in = 'a' @ d // in is true
6   in = 'c' @ d // in is false
7   in = 10 @ d // in is false
8   in = 10 @ d.values() // in is true
9   /* division over a dictionary gives the keyset
10  where the value of the key is the operand */
11  in = size( d / 10 ) > 0   // in is true
12  s = "hello"
13  in = "o" @ s // in is true
14  /* This works for linear collections even */
15  m = [2,3]
16  in = m @ l // true
17  in = l @ l // true
18  // the not in operator is defined as such
19  10 !@ l // true : not in
20  1 !@ l // false : not in
```

This was not simply put in the place simply because we simply dislike *x.contains(y)*. In fact we do dislike the form of object orientation where there is no guarantee that *x* would be null or not. Worse, it is impossible to test for *null* always, such is the prevailing nature of the bad code in Industry. Formally, then :

```
1  // equivalent function
2  def in_function(x,y){ (x != null) && x.contains(y) }
3  // or one can use simply
4  y @ x // same result
```

What about positions? *startsWith* and *endsWith* or even *containsAsSubList* ? All these 3 are essentially pattern matching, with various anchor points for collections

```
1  l = [1,2,3,4]
2  l #^ 1 // l indeed starts with 1
3  l #^ [1,2]  // l indeed starts with [1,2 ]
4  l #^ [] // vacously true
5  l #$ 4 // l indeed ends with 4
6  l #$ [3,4]  // l indeed ends with [3,4 ]
7  l #$ [] // vacously true
8  [2,3 ] #@ l // true, indeed [2,3] exists inside l as a sub list
9  [1,4 ] #@ l // false, indeed [1,4]  does not exist inside l as a sub list
```

This can be extended to patterns where a collection matches full the iteration order of another collection:

```
1  l = [1,2,3,4]
2  m = [1,2,3,4]
3  k = [2,3,4,1]
4  l == k // true : Collection equality is  match in any order
5  k #= m // false
6  l #= m // true
```

How about pure regular expressions? There are two operators related to regular expressions, the *match* operator, and then *not match* operator. See the guide to regular expressions.

```
1  re = "^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?$"
2  s = "hello"
3  match = ( s =~ re ) // false
4  match = ( s !~ re ) // true
5  f = "12.3456"
6  match = ( f =~ re ) // true
7  match = ( f !~ re ) // false
```

Thus, the operator "=~" is the *match* operator, while "!~" is the *not match* operator. Regex can be specified verbatim:

```
1  me = 'cool!'
2  me =~ ##^c.*$# // true
3  me =~ ##c.*# // false
4  me =~ ##co#g // global match, true
5  me =~ ##Co#ig // ignore case, global match, true
```

The right side of the operator, the # operand returns a pattern. It is obvious that the 'g' is for global, 'i' is for ignore case. 'm' is used for being multiline.

Moreover, to substitute pattern, that is replace with a variable, start with a question mark.

```
1  me = 'cool!'
2  p = 'Co'
3  me =~ ##p#ig // false, p is verbatim
4  me =~ ##? p#ig // true, p is executed as script first, then substituted
```

In the same spirit - List sub sequences can be tested for match:

```
1  l = [1,2,3,4]
2  l =~ [1,3] // true
3  l =~ [2] // true
4  l =~ [2,10] // false
5  l =~ [] // true, vacously
```

So in this sense there is a symmetry around the *IN* class of operators.

3.1.4   *Size of Containers : empty, size, cardinality*

To check whether a container is empty or not, use the *empty*() function, just mentioned above. Hence:

```
1   n = null
2   e = empty(n) // true
3   n = []
4   e = empty(n) // true
5   n = list()
6   e = empty(n) // true
7   d = { : }
8   e = empty(d) // true
9   nn = [ null ]
10  e = empty(nn) // false
```

For the actual size of it, there are two alternatives. One is the *size*() function :

```
1   n = null
2   e = size(n) // -1
3   n = []
4   e = size(n) // 0
5   n = list()
6   e = size(n) // 0
7   d = { : }
8   e = size(d) // 0
9   nn = [ null ]
10  e = size(nn) // 1
```

Observe that it returns negative given input null. That is a very nice way of checking null. The other one is the *cardinal* operator :

```
1   n = null
2   e = #|n|  // 0
3   n = []
4   e = #|n|  // 0
5   n = list()
6   e = #|n|  // 0
7   d = { : }
8   e = #|d|  // 0
9   nn = [ null ]
10  e = #|nn|  // 1
```

This operator in some sense gives a measure. It can not be negative, so cardinality of *null* is also 0.

3.1.5   *There Exist element with Condition : index, rindex , exists*

Given we have if "*y in container x*" or not, what if when asked a question like : "*is there a x in Collection C such that predicate P(x) is true*" ? Formally, then, given a predicate $P(x)$, and a collection C, we ask :

$$\exists x \in C \ s.t. \ P(x) = True$$

This pose a delicate problem.

Notice this is the same problem we asked about sorting. Is there an element 'x' in C, such that the sorting order is violated? If not, the collection is sorted. This brings back the *index* function . We are already familiar with the usage of $index()$ function from chapter 2. But we would showcase some usage :

```
1   l = [ 1, 2, 3, 4, 5, 6 ]
2   // search an element such that double of it is 6
3   i = index(l) where { $.o * 2 == 6 } // i : 2
4   // search an element such that it is between 3 and 5
5   i = index(l) where { $.o < 5 && $.o > 3 } // i : 3
6   // search an element such that it is greater than 42
7   i = index(l) where {  $.o > 42 }// i : -1, failed
```

The way index function operates is: the statements inside the anonymous block are executed. If the result of the execution is true, the index function returns the index in the collection. If for none of the elements the anonymous block assumes the true value, it returns a failure by returning −1.

NOTE that in the ZoomBA negative indices are proper indices into a collection. Thus, a code like this is tantamount to be a disaster waiting to happen:

```
1   l = [ 1, 2, 3, 4, 5, 6 ]
2   // search an element
3   i = index(l) where { $.o > 42 } // i : -1
4   // now we have found the element, so :
5   x = l[i] // no, x is 6. we must check the index value for >=0
```

This brings to the problem of actually finding the element, which is solved using *find()* function.

```
1   l = [ 1, 2, 3, 4, 5, 6 ]
2   // search an element
3   mc = find(l) where { $.o > 3 } // result is MonadicContainer
4   // now we have found the element, so :
```

```
5  r = mc.nil ? mc.value : null // r is 4
```

Index function runs from left to right, and there is a variation *rindex()* which runs from right to left.

```
1  l = [ 1, 2, 3, 4, 5, 6 ]
2  // search an element such that it is greater than 3
3  i = index(l) :: { $.o > 3 } // i : 3, :: is the where clause
4  // search an element such that it is greater than 3
5  i = rindex(l) :: { $.o > 3 } // i : 5
6  // search an element such that it is greater than 42
7  i = rindex(l) :: {  $.o > 42 } // i : -1, failed
```

Thus, the *there exists* formalism is taken care by these operators and functions together.

The function *exists()* to ensure simpler usage on collections where index is kind of meaningless.

```
1  l = [ 1, 2, 3, 4, 5, 6 ]
2  // search an element such that it is greater than 3
3  // reads there exists item in collection where clause
4  exists(l) where { $.o > 3 } // true
```

3.1.6   *For All elements with Condition : select*

We need to solve the problem of *for all*. This is done by *select()* function . The way select function works is : executes the anonymous statement block, and if the condition is true, then select and collect that particular element, and returns a list of collected elements.

```
1  l = [ 1, 2, 3, 4, 5, 6 ]
2  // select all even elements
3  evens = select(l) where { $.o % 2 == 0 }
4  // for math folks
5  evens = select(l) :: { 2 /? $.o } // reads : where 2 divides the number
6  // select all odd elements
7  odds = select (l) :: { $.o % 2 == 1 }
```

3.1.7   *Partition a collection on Condition : partition*

Given a *select()*, we are effectively partitioning the collection into two halves, *select()* selects the matching partition. In case we want both the partitions, then we can use the *partition()* function .

```
1  l = [ 1, 2, 3, 4, 5, 6 ]
2  #(evens,odds) = partition(l) :: { $.o % 2 == 0 }
3  println(evens) // prints 2, 4, 6
4  println(odds) // prints 1, 3, 5
```

3.1.8   *Collecting value on Condition : select, from, concur*

Given a *select()* or *partition()*, we are collecting the values already in the collection. What about we want to change the values we are collecting, or what about the container we want to collect into?

These scenarios can be translated into:

```
1  collector = collection()
2  def condition(item){/* some predicate */}
3  def map(item){/* some mapping */}
4  // now the loop
5  for ( item : collection ){
6     if ( condition(item) ){
7       collector.add( map(item ) )
8     }
9  }
10 // use collector
```

Or, one can use the standard:

```
1  l = [ 1, 2, 3, 4, 5, 6 ]
2  evens = select(l) where { $.o % 2 == 0 } as { $.o ** 2 }
3  println(evens) // prints 4, 16, 36
```

where is the condition function, while *as* is the mapper function. You can pass the collector collection also:

```
1  select([0:5],set()) where { 2 /? $.o }
```

this, of course will collect the items in a set. While *select()* only supports two levels of composition, one predicate and one mapper, *from()* supports full functional composition:

```
1  from([0:11],set()) where { 3 /? $.o } as {  $.o % 3 }
```

A concurrent (multi-threaded) form of *from* is available as *concur*:

```
1  concur([0:11],set()) where { 3 /? $.o } as {  $.o % 3 }
```

As one can see there is no visible difference, only in the name. It works almost exactly like *from* but does the mapping in a multi-threaded way. One has to be careful to avoid *break* statement in the function - it has no meaning in an un co-ordinated non deterministic system.

3.1.9   *As vs Where*

By now, it is pretty clear that the anonymous functions take two forms. *where* and *as* - one is used to hint conditional (predicate) aspect of things, and another mapper aspect of things. *where* should be strictly used when we are looking at decision making. *as* should be strictly used when we are looking at mapping functions.

In *sort[ad]()* functions, the implementation differs internally based on what sort of anonymous function is being used. Observe a student object as follows. If we want to sort the list of student generated by a field, there is no point writing a custom comparator - we can very well give a mapping function that scalarifies the object into an auto comparable:

```
1  def Student : { role : 42 , name : "HHGG"  }
2  students = list ( [0:5 ) ) as {
3      new ( Student , role : random(1000), name : random( "\S+", 32 ) )
4    }
5  // now sort by role ?
6  sorta( students ) as { $.o.role }
7  // now sort by name ?
8  sorta( students ) as { $.o.name }
```

This is 1-1 with how a select query would chose to scalarify a tuple for ordering in *Order By* clause.

3.1.10   *Inversion of Logic*

As we have discussed there is a relation between $\exists$ and $\forall$, we can use them for practical applications. We would be using them everywhere, and as Kurt Lewin said :

> *There is nothing so practical as a good theory.*

So, we would start with all these examples in inverted logic. Observe that *select()* mandates a guaranteed runtime of $\Theta(n)$, while *index()* has a probabilistic runtime of $\Theta(n/2)$. So, we should generally choose *index()* over *select()*. For example, take the problem of *are all numbers in a list larger than 0*, it can be solve in two ways:

```
1  l = [10,2,3,10, 2, 0 , 9 ]
2  // forall method 1
3  size(l) == size ( select(l) :: { $.o > 0 } )
4  // forall method 2 : note the inversion of logic
5  empty ( select(l) :: { $.o <= 0 } )
6  // there exists method : note the inversion of condition from 1
7   !exists(l) :: { $.o <= 0 }
```

Please choose the 3rd one, that makes sense, takes less memory, and is optimal.

3.2   OPERATORS FROM SET ALGEBRA

3.2.1   *Set Algebra*

Set algebra , in essence runs with the notions of the following ideas :
1.   There is an unique empty set.
2.   The following operations are defined :

a. Set Union is defined as :

$$U_{AB} = A \cup B := \{x \in A \ or \ x \in B\} := U_{BA}$$

b. Set Intersection is defined as :

$$I_{AB} = A \cap B := \{x \in A \ and \ x \in B\} := I_{BA}$$

c. Set Minus is defined as :

$$M_{AB} := A \setminus B := \{x \in A \ and \ x \notin B\}$$

thus, $M_{AB} \neq M_{BA}$.

d. Set Symmetric Difference is defined as :

$$\Delta_{AB} := (A \setminus B) \cup (B \setminus A) := \Delta_{BA}$$

e. Set Cross Product is defined as :

$$X_{AB} = \{(a,b) \mid a \in A \ and \ b \in B\}$$

thus, $X_{AB} \neq X_{BA}$

3. The following relations are defined:

   a. Subset Equals :
   $$A \subseteq B \ when \ x \in A \implies x \in B$$

   b. Equals :
   $$A = B \ when \ A \subseteq B \ and \ B \subseteq A$$

   c. Proper Subset :
   $$A \subset B \ when \ A \subseteq B \ and \ \exists x \in B \ s.t. \ x \notin A$$

   d. Superset Equals:
   $$A \supseteq B \ when \ B \subseteq A$$

   e. Superset :
   $$A \supset B \ when \ B \subset A$$

3.2.2 *Set Operations on Collections*

For the sets, the operations are trivial. The only interesting departure from regular mathematical operator is "\" which is the operator for *collection (set) minus*. This is because operators like "+, −" are used to do element operations onto collections.

```
1  s1 = set(1,2,3,4)
2  s2 = set(3,4,5,6)
3  u = s1 | s2 // union is or : u = { 1,2,3,4,5,6}
4  i = s1 & s2 // intersection is and : i = { 3,4 }
5  m12 = s1 \ s2 // m12 ={1,2}
6  m21 = s2 \ s1 // m12 ={5,6}
7  delta = s1 ^ s2 // delta = { 1,2,5,6}
```

For the lists or arrays, where there can be multiple elements present, this means a newer formal operation. Suppose in both the lists, an element 'e' is present, in n and m times. So, when we calculate the following :

1. Intersection : the count of e would be $min(n,m)$.

2.   Union : the count of e would be $max(n,m)$.
3.   Minus : the count of e would be $max(0, n-m)$.

With this, we go on for lists:

```
1  l1 = list(1,2,3,3,4,4)
2  l2 = list(3,4,5,6)
3  u = l1 | l2 // union is or : u = { 1,2,3,3,4,4,5,6}
4  i = l1 & l2 // intersection is and : i = { 3,4 }
5  m12 = l1 \ l2 // m12 ={1,2,3,4}
6  m21 = l2 \ l1 // m12 ={5,6}
7  delta = l1 ^ l2 // delta = { 1,2,3,4,5,6}
```

Now, for dictionaries, the definition is same as lists, because there the dictionary can be treated as a list of key-value pairs. So, for one pair to be equal to another, both the key and the value must match. Thus:

```
 1  d1 = {'a' : 10, 'b' : 20 , 'c' : 30 }
 2  d2 = {'c' : 20 , 'd' : 40  }
 3  // union is or : u = { 'a' : 10, 'b' : 20, 'c' : [ 30,20] , 'd' : 40  }
 4  u = d1 | d2
 5  i = d1 & d2 // intersection is and : i = { : }
 6  m12 = d1 - d2 // m12 = d1
 7  m12 = d1 \ d2 // m12 = d1
 8  m21 = d2 - d1 // m12 = d2
 9  m21 = d2 \ d1 // m12 = d2
10  delta = d1 ^ d2 // delta = {'a' : 10, 'b' : 20,  'c' : [30,20] , 'd': 40}
```

For dictionaries the "\" is same as "−". Notice the symmetric difference in case the value of the key does not match. In that case the result is an ordered pair of *(left,right) values*.

3.2.3 *Collection Cross Product and Power*

The cross product, as defined, with the multiply operator:

```
1  l1 = [1,2,3]
2  l2 = ['a','b' ]
3  cp = l1 * l2
4  /* [[1, a], [1, b], [2, a], [2, b], [3, a], [3, b]] := cp */
```

Obviously we can think of power operation or exponentiation on a collection itself. That would be easy (we use ':=' to say *defined as*) :

$$A^0 := \{\} , A^1 := A , A^2 := A \times A$$

and thus :

$$A^n := A^{n-1} \times A$$

Collection power in general can not be negative. Here are some examples now:

```
1  b = [0,1]
2  gate_2 = b*b // [ [0,0],[0,1],[1,0],[1,1] ]
3  another_gate_2 = b ** 2 // same as b*b
4  gate_3 = b ** 3 // well, all truth values for 3 input gate
```

```
5  gate_4 = b ** 4 // all truth values for 4 input gate
6  b ** 0 // [] : power zero is empty collection
```

String is also a collection, and all of these are applicable for string too. But it is a special collection, so only power operation is allowed.

```
1  s = "Hello"
2  s2 = s**2 // "HelloHello"
3  s_1 = s**-1 // "olleH"
4  s_2 = s**-2 // "olleHolleH"
```

One interesting case is that of negative power. A negative power is a valid power for an *ordered collection* like an *array* or a *list*, negation implies a reversal of order in such a collection, if such thing is possible :

```
1   b = [0,1]
2   rb = b** -1 // [1 , 0 ]
3   s = "hello"
4   rs = s**-1 // "olleh"
5   rb = b** -2 // [[1, 0], [1, 1], [0, 0], [0, 1]]
6   s = "hello"
7   rs = s**-2 // "olleholleh"
8   // sorted list
9   sl = slist(3,1,2 ) // 1,2,3
10  sl ** -1 // 3,1,2
11  // ordered set
12  os = oset(1,2,3)
13  os ** -1 // {3,2,1}
14  // sorted set
15  ss = sset(1,3,2)
16  ss ** -1 // {3,2,1}
```

these are simply not there for curious purposes. In the examples section we will see how they ease out some interesting algorithms. A very practical use case of power operator for string is padding by zeros for a binary integer of a fixed size:

```
1  n = 23
2  bn = str(n,2) // convert into binary string
3  padded_8 = "0" ** (8 - #|bn| ) + bn // 00010111 : convert to 8 bit
4  padded_8 = str(n,2,8)  // 00010111 : convert to 8 bit
```

3.2.4 *Collection Relation Comparisons*

The operators are defined as such:
1. $A \subset B$ is defined as $A < B$
2. $A \subseteq B$ is defined as $A <= B$
3. $A \supset B$ is defined as $A > B$
4. $A \supseteq B$ is defined as $A >= B$
5. $A = B$ is defined as $A == B$

Note that when collections can not be compared at all, it would return false to showcase that the relation fails.

So, we go again with sets:

```
1   s1 = set(1,2,3)
2   s2 = set(1,3)
3   sub = s2 < s1 // true
4   sup = s1 > s2 // true
5   sube = ( s2 <= s1 ) // true
6   supe = (s1 >= s2) // true
7   s3 = set(5,6)
8   s1 < s3 // false
9   s3 > s1 // false
10  s1 != s3 // true
```

So, we go again with lists:

```
1   l1 = list(1,2,3,3,4)
2   l2 = list(1,3,2)
3   sub = l2 < l1 // true
4   sup = l1 > l2 // true
5   sube = ( l2 <= l1 ) // true
6   supe = (l1 >= l2) // true
7   l3 = list(5,6)
8   l1 < l3 // false
9   l3 > l1 // false
10  l1 != l3 // true
```

And finally with dictionaries:

```
1   d1 = {'a' : 10, 'b' : 20 , 'c' : 30 }
2   d2 = {'c' : 30 , 'a' : 10  }
3   sub = ( d2 < d1) // true
4   sup = ( d1 > d2) // true
```

3.2.5   *Mixing Collections*

One can choose to intermingle *set* with *list*, that promotes the *set* to *list*. Thus :

```
1   s = set(1,2,3)
2   l = list(1,3,3,2)
3   sub = s < l  // true
4   sup = l > s // true
5   s == l // false : promotes both to list and checks
6   u = l | s // u = [1,2,3,3 ]
```

3.2.6   *Collections as Tuples*

When we say $[1,2] < [1,2,3,4]$ we obviously mean as collection itself. But what about we start thinking as tuples? Does a tuple contains in another tuple? That is we can find the items in order? "@" operator solves that too :

```
1  l = [1,2]
2  m = [ [1,2],3,4]
3  in = l @ m // true
```

The *index*(), and *rindex*() generates the index of where the match occurred:

```
1  l = [1,2]
2  m = [1,2,3,4,1,2]
3  // read which : index is l in m?
4  forward = index( l , m) // 0
5  backward = rindex( l , m) // 4
```

Sometimes it is of importance to get the notion of *starts_with* and *ends_with*. There are two special operators :

```
1  // read m starts_with l
2  sw = m #^ l // true
3  // read m ends with l
4  ew = m #$ l // true
```

Note that this is also true for strings, as they are collections of characters. So, these are legal:

```
1  s = 'abracadabra'
2  prefix = 'abra'
3  suffix = prefix
4  i = index ( prefix, s ) // 0
5  r = rindex( suffix, s ) // 7
6  sw = s #^ prefix // true
7  ew = s #$ suffix // true
```

These are not fancy stuffs. These are necessary to move over from the stupidity that is known as *null based programming* where lots of efforts gets nullified by the stupidity of null. Observe, the same null thing:

```
1  s = 'abracadabra'
2  prefix = 'abra'
3  null #^ s // false
4  prefix #^ null // false
5  [null,null] #^ null // true
```

and thus, we can see where it does the value add. Python fails in this case, because none of it's syntaxes permits None type to be used as collection to yield True/False for *in* or *matches*.

There is this other operator called *InOrder* : #@ which tells whether a sub collection exists in order, as tuple in the larger collection when the larger collection is considered as tuple:

```
1  l = [0,1,2,3,4]
2  [1,2] #@ l // true
3  [2] #@ l // true
4  [2,1] #@ l // false
```

as usual, they are all null ready. The *in* : @ operator is different from *in order*.

There is another interesting operator # = *in order and equals* :

```
1  l = [0,1,2,3,4]
2  m = [0,1,3,2,4]
3  k = [0,1,2,3,4]
4  l == m   // true
5  l #= m   // false
6  l #= k   // true
7  l #= l   // true
8  // sub sequence
9  l =~ [1,3] // true
10 m =~ [0,2,4] // true
11 l =~ [4,1] // false
```

This is how most languages thinks about *equality* in list and arrays. In ZoomBA it is called *iteration equals*, or *order equals*.

Thus, in order is precisely what it is, if the items from the passed collection occurs as sub tuples (can be build by projection operations) of the parent collection. Formally, let's end this with a bit of formalism as we have started it.

Define a projection operation $\Pi_{i,j}(C)$ with $i \leq j$ which slice the collection C from index i to index j, or rather a *sub()* function. Let $|X|$ be the cardinality or size of the collection X. Then, the following are defined as such :

1. **In Order And Equals:**
   $A \#= B$ iff $|A| = |B|$; $\forall i \ \Pi_{i,i}(A) = \Pi_{i,i}(B)$.

2. **In Order :**
   $x \#@ C$ iff $\exists(i,j)$ such that $\Pi_{i,j}(C) = x$.

3. **Starts With :**
   $C \#\hat{} x$ iff $\exists j$ such that $\Pi_{0,j}(C) = x$.

4. **Ends With :**
   $C \#\$ x$ iff $\exists i$ such that $\Pi_{i,|C|-1}(C) = x$.

3.3   ASSERTIONS

It of utmost important that we do protective coding. What does that mean? That means, for any function, verify the inputs, and decides, what needs to be done. For the same purpose, there are some very specific functions which are in place.

3.3.1   *Existence of Variables*

There are one specific function which is in place: *is()* .

```
1  is( x )  // false, x does not exist in Context
2  x = 10
3  is( x )  // true, x exists
4  is( null )  // true, x exists and is null
5  del x // removes the variable name 'x'
6  is(x) // false, x does not exist any more
```

A related operator is *??* which is read as :

```
1  val = try_expression ?? default_expression
2  d = { 'a' : 11 }
3  val = d.y ?? 42 // val is 42
4  val = d.a ?? 42 // val is 11
```

If *try_expression* fails or is null, then assign val to default, else val is try expression.

3.3.2 *Verifying Truth about Expressions*

Sometimes, most of the time, we need to check if some expression is true or false. Most of the time, that might lead you do to throw exceptions or errors. ZoomBA handles it in a neat way.

There are 3 in built constructs one can try upon *assert()*, *panic()*, *test()*. Here are examples of the same:

```
1   x = 42
2   // expression true, so error will not be thrown
3   assert( x == 42, 'Why x is not 42?' )
4   // expression false, so error will not be thrown
5   panic( x != 42, 'Why x is not 42?' )
6   // expression false, so error will BE thrown
7   assert( x != 42, 'Why x is not 42?' )
8   // expression true, so error will BE thrown
9   panic( x == 42, 'Why x is not 42?' )
10  // true, so nothing will happen
11  test(x == 42, 'I am good' )
12  // false, so would log it to standard error
13  test(x != 42, 'I am not good' )
```

All of these functions are arbitrary valued arguments, with one mandatory argument, the expression which evaluates to true, false. Rest all are processed in, and passed to the runtime, if the assertion fails. Thus, all passed extra arguments can be used by runtime, thereby extending the functionality if it is intended to be.

There are guard blocks around each of them. One example is :

```
1   assert( '"x" does not exist' ) where { x }
```

What is the difference from the original form? If we use the basic expression form, then, we do get a variable error that 'x' does not exist. Which is entirely a different problem. The where clause is called a guard block, there, expression can be safely executed, to produce true,false. Obviously, for assert() and test() the clause must return false to fire error, and for panic() it is to be true.

4

Collections And Comprehension

Comprehension is a method which lets one create newer collection from older ones. In this chapter we would see how different collections can be made from existing collections.

4.1 using anonymous argument

4.1.1 *List*

The general form for list and arrays can be written (as from chapter 2):

```
1  def map(item) {/* does some magic */}
2  def comprehension(function, items){
3    c = collection()
4    for ( item : items ){
5      c_m = function(c)
6      c.add( c_m )
7    }
8    return c
9  }
10 // finally :
11 comprehension(map, items)
12 //or this way :
13 c = collection( items ) as { map($.o) }
```

Obviously *list*() generates *list* and *list.array*() generates *array*. Any collection structure can be used as a collector : *list, set, oset, sset, dict, sdict, odict, mset* . Hence, these are valid :

```
1  ln = list(1,2,3,4) as { $.o ** 2 } // [1,4,9,16 ]
```

So, the result of the anonymous block is taken as a function to map the item into a newer item, and finally added to the final collection.

One can use standard function also in case of anonymous function :

```
1  def square( inx, itm, partial, context ){
2    itm ** 2
3  }
4  ln = list(1,2,3,4) as square // same !
5  ln = list(1,2,3,4) as def(inx,itm) { itm ** 2 } // same!
```

4.1.2   Set

Set also follows the same behavioural pattern just like *list*(), but one needs to remember that the *set* collection does not allow duplicates. Thus:

```
1   s = set( 1,2,3,4,5 ) as { $.o % 3 } // [0,1,2]
```

Basically, the *map* function for the *set* construct, defines the key for the item. Thus, unlike its cousins *list*() and *array*(), the *set*() function may return a collection with size less than the input collection.

4.1.3   Dict

Dictionaries are special kind of a collection with (*key*, *value*) pair with unique keys. Thus, creating a dictionary would mean uniquely specifying the key, value pair. That can be specified either in a dictionary tuple way, or simply a pair way:

```
1   // range can be passed in for collection
2   d = dict([0:4]) as { t = { $.o : $.o **2 }  }
3   /* d = {0 : 0, 1 : 1, 2 : 4, 3:9 }  */
4   d = dict([0:4]) as { [ $.o , $.o **2 ]  }  // same as previous
```

There is another way of creating a dictionary, that is, passing two collections of same size, and making first collection as keys, mapping it to the corresponding second collection as values:

```
1   k = ['a','b','c' ]
2   v = [1,2,3]
3   d = dict(k,v)
4   /*  d = { 'a' : 1 , 'b' : 2, 'c' : 3 }  */
```

This is formally the operation called pairing:

$$D : (K, V) \rightarrow T \; ; \; t_i = (k_i, v_i) \; ; \; k_i \in K \; ; \; v_i \in V$$

4.1.4   Multi Set, Group

We got introduced to *mset()* sometimes back. One can think it is as a frequency counter for elements of a collection. But that is not what it does. *mset()* can also group items under a key, as demonstrated :

```
1   l = [0:10].list() // generate a list from this range
2   ms = mset( l ) as {  2 /? $.o } // group them under "even" and "odd"
3   // {false=[1, 3, 5, 7, 9], true=[0, 2, 4, 6, 8]} // HashMap
```

In essence, what it did is to 'group' the collections under multiple groups, all items in the same group must have same key.

A related problem is to run another function with the collection of each group. This is done by the *group()* function. For examples:

```
1  l = [0:10].list() // generate a list from this range
2  // group them under "even" and "odd",
3  //and then count the list of items in each group,
4  // making it the value under the key
5  g = group( l ) as {  2 /? $.o }  as {  size($.value) }
6  // results in :  {false=5, true=5} // HashMap
```

A *continue* inside the *group()* will remove the key from the final group, as follows:

```
1  l = list ( [0:100]) as { { "id" : random(100000) , "v" : random(100) }  }
2  // filter all items where id is mapped to more than 3 elements
3  g = group( l ) as {  $.id }  as {
4      sz = size($.value)
5      continue( sz > 3 )
6      $.value }
```

This lets one filter on the result of the grouping.

Another interesting operator is * for the dictionaries. Imagine the dictionary can be considered as a vector with key being the components. Thus we can come up with a rudimentary group operation such that $d_1 * d_2$ defines as taking dot product of the keys (intersection of the keys) while the value would be simply a pair of values from d_1, d_2 as follows:

$$r_{ij} = \{k \rightarrow (d_i[k], d_j[k])\}$$

Thus the sample is as follows:

```
1  a = { "a" : 10, "b" : 20 , "d" : 100 }
2  b = {  "b" : 30 , "c" : 40 }
3  r_ab = a * b // dict { "b" : [20,30] }
4  r_ba = b * a // dict { "b" : [30,20] }
```

What sort of Map we can use for *mset* and *group* ? We can use ordered map or a sorted map too. This can be easily specified as follows:

```
1  l = [1,2,3,4,5,6]
2  mso = mset(col=l,order=true) // ordered map
3  mss = mset(col=l,sorted=true) // sorted map
4  // or even
5  ms = mset(col=l, acc=sdict() ) // pass the accumulator
```

This way, we can reuse collector accumulator from any previous steps.

4.1.5 *Stack And Queue*

Collection *list()* can be used as both Stack and the Queue. This is because the *ZList* wrapper which wraps around any JVM list actually implements Java Deque .

For example :

```
1  // use as queue
2  q = list()
3  for ( x : [0:10] ) {  q.add(x) } // enqueue
4  q.remove() // we get back 0
5  // use as stack
6  s = list()
7  for ( x : [0:10] ) {  s.push(x) } // push to stack
8  s.pop() // we get back 10
```

You can read more about how *Deque* can act as both.

4.1.6   *JVM Arrays*

Collection *collection.array(type)* can be used to create arrays: For example :

```
1   x = [0,1,2,3]
2   //@[ 0,1,2,3 ] // ZArray
3   x.array('int')
4   //0,1,2,3 // int[]
5   //x.array('double')
6   //0.0,1.0,2.0,3.0 // double[]
7   x.array('float')
8   //0.0,1.0,2.0,3.0 // float[]
9   x.array('long')
10  //0,1,2,3 // long[]
11  x = [true, false]
12  //@[ true,false ] // ZArray
13  x.array('bool')
14  //true,false // boolean[]
```

4.2   CONSTRUCTS ALTERING THE ITERATION FLOW

In normal iterative languages, there are *continue* and *break*. The ideas are borrowed from them, and improved to give a declarative feel onto it.

4.2.1   *Continue and Break*

The idea of continue would be as follows :

```
1   for ( i : items ){
2      if ( condition(i) ){
3          // execute some code
4          continue
5      }
6      // do something else
7   }
```

That is, when the condition is true, execute the code block, and then continue without going down further (not going to do something else). The idea of the break is :

```
1   for ( i : items ){
2       if ( condition(i) ){
3       /* execute some code */
4       break
5       }
6       // do something else here
7   }
```

That is, when the condition is true, execute the code block, and then break without proceeding with the loop further. Evidently they change the flow of control of the iterations.

4.2.2   *Continue*

As we can see, the *condition*() is implicit in both *break* and *continue*, in ZoomBA this has become explicit. Observe that:

```
1  for ( i : items ){
2      continue( condition(i) ){ /* continue after executing this */}
3      // do something else
4  }
```

is equivalent of what was being shown in the previous subsection. As a practical example, lets have some fun with the problem of *FizzBuzz*, that is, given a list of integers, if *n* is divisible by 3 print *Fizz*, if something is divisible by 5, print *Buzz*, and for anything else print the number *n*. A solution is :

```
1  for ( n : integers ){ // note that "x /? y" implies x divides y ?
2      continue( n /? 15 ){ println('FizzBuzz') }
3      continue( n /? 3  ){ println('Fizz') }
4      continue( n /? 5  ){ println('Buzz') }
5      println(n)
6  }
```

Obviously, the block of the continue is optional. Thus, one can freely code the way it was mentioned in the earlier subsection.

4.2.3 *Break*

As mentioned in the previous subsection, *break* also has the same features as continue.

```
1  for ( i : items ){
2      break( condition(i) ){ /* break loop after executing this */}
3      // do something else
4  }
```

is equivalent of what was being shown in the previous subsection. As a practical example, lets find if two elements of two lists when added together generates a given value or not. Formally :

$$\exists (a, b) \in A \times B \; ; \; s.t. \; a + b = c$$

```
1  A = [ 1, 4, 10, 3, 8 ]
2  B = [ 2, 11, 6 , 9 ]
3  c = 10
4  for ( p : A*B  ){
5      break( p[0] + p[1] ==  c ){ printf( '%d %d\n' , p[0] , p[1] ) }
6  }
```

Obviously, the block of the break is optional.

4.2.4 *Substitute for Select*

One can readily use the *break* and *continue* in larger scheme of comprehensions. Observe this :

```
1  l_e1 = select([0:10]) :: { $.o % 2 == 0 }
2  l_e2 = list([0:10]) -> { continue($.o % 2 != 0) ; $.o }
```

Both are the same list. Thus, a theoretical conversion rule is :

```
1  ls = select(collection) where { <condition> }
2  lc = list(collection) as { continue( ! ( <condition> ) ) ; $.o }
```

now, the *ls == lc*, by definition. More precisely, for a generic *select* with *where* clause:

```
1  ls = select(collection) where {  <condition body>  }
2  lc = list(collection) as { continue( !(<condition>) ) ; <body> }
```

Obviously, the *list* function can be replaced with any *collection* type : *set*, *dict*.
    Similarly, *break* can be used to simplify conditions :

```
1  l_e1 = select([0:10]) :: { $.o % 2 == 0 && $.o < 5 }
2  l_e2 = list([0:10]) -> { break( $.o > 5 ) ;
3         continue($.o % 2 != 0) ; $.o }
```

Note that Break body is inclusive:

```
1  l = list([0:10]) as { break( $.o > 3 ) ; $.o }
2  /* l = [ 0, 1,2,3 ] */
3  l = list([0:10]) as { break( $.o > 3 ) { $.o } ; $.o }
4  /* l = [ 0, 1,2,3, 4] */
```

So, with a body, the value from the body is included into the collection.

4.2.5   *Uses of Partial*

One can use the *PARTIAL* for using one one function to substitute another, albeit loosing efficiency. Below, we use list to create *effectively* a set:

```
1  l = list( 1,1,2,3,1,3,4 ) as { continue( $.o @ $.p ) ; $.o }
2  /* l = [ 1,2,3, 4] */
```

But it gets used properly in more interesting of cases. What about finding the prime numbers using the Sieve of Eratosthenes? We all know the drill imperatively:

```
1  def is_prime( n ){
2      primes = set(2,3,5)
3      for ( x : [6:n+1] ){
4          x_is_prime = true
5          for ( p : primes ) {
6              break( x % p == 0  ){ x_is_prime = false }
7          }
8          if ( x_is_prime ){
9            if ( x == n ) return true
10           primes += x
11         }
12     }
13     return false
14 }
```

Now, a declarative style coding :

```
1  def is_prime( n ){
2      primes = list( [2:n+1] ) as {
3      // store the current number
4          me = $.o
5          // check : *me* is not prime – using partial set of primes
6          not_prime_me = exists( $.p ) where {  me % $.o == 0 }
7          // if not a prime, continue
```

```
8            continue( not_prime_me )
9            me // collect me, if I am prime
10       }
11     // simply check if n is last of the primes
12     ( n == primes[-1] )
13  }
```

Observe that, if we remove the comments, it is a one liner. It really is. Hence, declarative style is indeed succinct, and very powerful. The one reason why we need to create the *me* variable is due to the implicit parameter $. The implicit loop variable also as a field $ which access the parent's loop parameter. Thus, the same code can be written in a line :

```
1  n == (select([2:n+1]) where{ exists($.p) where{ $.o /? $.$.o } })[-1]
```

4.3 COMPREHENSIONS USING MULTIPLE COLLECTIONS : JOIN

In the previous sections we talked about the comprehensions using a single collection. But we know that there is this most general purpose comprehension, that is, using multiple collections. This section we introduce the $join()$ function.

4.3.1 *Formalism*

Suppose we have multiple sets $S_1, S_2, ... S_n$. Now, we want to generate this collection of tuples

$$e = <e_1, e_2, ..., e_n> \in S_1 \times S_2 \times ... \times S_n$$

such that the condition (predicate) $P(e) = true$. This is what join really means. Formally, then :

$$J(P, e) := \{e \in S_1 \times S_2 \times ... \times S_n \mid P(e)\}$$

4.3.2 *As Nested Loops*

Observe this, to generate a cross product of collection A, B, we must go for nested loops like this:

```
1  for ( a : A ){
2     for ( b : B ) {
3        printf ( '(%s,%s)', a,b )
4     }
5  }
```

This is generalised for a cross product of A, B, C, and generally into any cross product. Hence, the idea behind generating a tuple is nested loop.

Thus, the $join()$ operation with $condition()$ predicate with a $mapper()$ essentially is :

```
1  collect = collection()
2  for ( a : A ){
3     for ( b : B ) {
4        tuple = [a,b]
5        if ( condition ( tuple ) ) { collect += map(tuple) }
6     }
7  }
```

4.3.3 *Join Function*

Observe that the free join, that is a join without any predicate, or rather join with predicate defaults to true is really a full cross product, and is available using the power operator :

```
1  A = ['a','b']
2  B = [ 0, 1 ]
3  j1 = A * B // [ [a,0] , [a,1] ,[b,0], [b,1] ]
4  j2 = join( A , B ) // [ [a,0] , [a,1] ,[b,0], [b,1] ]
```

But the power of *join*() comes from the predicate expression one can pass in the anonymous block. Observe now, if we need 2 permutations of a list of 3 : $^3P_2$:

```
1  A = ['a','b', 'c' ]
2  // generate 2 permutations, when $.o is collection
3  // one can access the items by index like this
4  p2 = join( A , A ) where { $.0 != $.1 }
```

4.3.4 *Permutations*

In the last subsection we figured out how to find 2 permutation. The problem is when we move beyond 2, the condition can not be aptly specified as $.0 \neq $.1. It has to move beyond. So, for $^3P_3$ we have :

```
1  A = ['a','b', 'c' ]
2  // generate 3 permutations
3  p2 = join( A , A,  A ) where { #|set($.o)| == #|$.o| }
```

which is the declarative form for permutation, given all elements are unique. This can be simply created by the pre-defined *perm()* function :

```
1  A = ['a','b', 'c' ]
2  // generate 3 permutations
3  p2 = perm( A, 2  ) //  From collection A, generate 2 permutations
4  // @[ a,b ],@[ b,a ],@[ a,c ],@[ c,a ],@[ b,c ],@[ c,b ] //
       CombinatoricsIterable
5  // hence,
6  perm(collection, number) // generates an iterable
```

4.3.5 *Combinations*

What about combinations then? Difference between permutation and combination is, in combination, order does not matter, hence $(1, 2)$ is same combination as $(2, 1)$, while they are two distinct permutations.

```
1  A = ['a', 'b', 'c' ]
2  c2 = join( A, A ) where {
3      c = sorta( list($.o) )
4      c !@ $.p &&  #|set(c)| == #|c|
5  }
```

which is the declarative form for combination, given all elements are unique. This can be simply created by the pre-defined *comb()* function :

```
1  A = ['a','b', 'c' ]
2  // generate 3 combinations
3  c2 = comb( A, 2  ) //  From collection A, generate 2 combinations
4  @[ a,b ],@[ a,c ],@[ b,c ] // CombinatoricsIterable
5  comb(collection, number) // generates an iterable
```

4.3.6   Searching for a Tuple

In the last section we were searching for a tuple t from two collections A, B, such that $c = t.0 + t.1$. We did that using power and break, now we can do slightly better:

```
1  A = [ 1, 4, 10, 3, 8 ]
2  B = [ 2, 11, 6 , 9 ]
3  c = 10
4  v = join(A,B) :: { break( $.0 + $.1 == c) }
5  /* v := [[1,9]] */
```

which is the declarative form of the problem.

4.3.7   SQL Style Optimization

Imagine someone trying to do this:

```
1  l1 = list ( [0:1000] ) as { { "id" : random(10000000) , "x" : random(100) } }
2  l2 = list ( [0:1000] ) as { { "id" : random(10000000) , "y" : random(100) } }
3  #( t, o ) = #clock{
4    // now do join over something
5    g1 = mset( l1 ) as { $.id }
6    g2 = mset( l2 ) as { $.id }
7    g1 * g2  // would not work - but hypothetical code here
8  }
```

Essentially this is joining with an attribute 'id'. This can be done with a massive complexity of $\Theta(MN)$ as follows:

```
1  join( l1, l2 ) where { $.l.id == $.r.id } as { $.l | $.r }
```

This pose a problem. Hence we do have a sql optimised join which relies on keys:

```
1  a = list ( [0:300] ) as {  { "id" : $.i + 1 ,  "a" : random(500) } }
2  b = list ( [0:300] ) as {  { "id" : $.i + 1 ,  "b" : random(500) } }
3  c = list ( [0:300] ) as {  { "id" : $.i + 1 ,  "c" : random(500) } }
4
5  r = join( a, b, c ) as {  $.o.id } where {
6    $.0.a > 100 && $.1.b < 100 && $.2.c > 200
7  } as {  ($.0 | $.1 | $.2)
```

Notice the first *as* block. Ordering matters. First as block is the key to be used for a *hashed join*. This takes care of the problem of complexity.

4.3.8   All Possible Sub Collections

Suppose, given a collection C, we want to find out all possible sub collection inside that collection. This tantamount to what is known as Power Set or 2^C. One way to solve this problem

is to imagine that each sub collection from the original collection is created by a bitmap with 1 means select that index, and 0 means do not select the index. If the $|C| = n$, then clearly 2^n of those sub collections possible. This can be coded as in:

```
1   def all_sub_collection( col ){
2     power_col = list()
3     n = size(col)
4     for ( b : [0 : 2 ** n ] ){
5       bit_map = str(b,2) // binary rep of integer b
6       bit_map = '0' ** ( n - size( bit_map) ) + bit_map
7       sub_col = list()
8       for ( i : [0:n] ){
9           if ( bit_map[i] == _'1' ){
10              sub_col += col[i]
11          }
12      }
13      power_col.add( sub_col )
14    }
15    println( power_col )
16  }
```

This is too much of non declarative hodge-podge. This can be simply coded in as :

```
1   a =[0, 1, 2]
2   // generates all possible combination sequences possible using collection
3   for ( t : powerset(a) ) { println(t) }
```

4.3.9 *Projection on Collections*

Sometimes it is needed to create a sub-collection from the collection. This, is known as projection (choosing specific columns of a row vector).

While project works on a range *(from,to)*, the generic idea can be expanded. What if we want to create a newer collection from a collection using a *range* type? We can :

```
1   a =[0, 1, 2, 3, 4]
2   a[0:2] // @[0, 1, 2]
```

Now, given a set of indices, one can define a generic project operation, but that is very easy to do using collection comprehensions :

```
1   a = [0, 1, 2, 3, 4]
2   indices = [0,2]
3   p_a = list(indices) as { continue( $.i != $.o ) ; a[$.o] }
```

This simply selects those indices in order from the collection *a*. In fact, defining *project()* or *tuple()* is easy :

```
1   a = [0, 1, 2, 3, 4]
2   indices = [from_index,to_index]
3   p_a = select(a) where {  $.i >= indices.0 && $.i < indices.1 ) }
```

CHAPTER 5

Types and Conversions

Types are not much useful for general purpose programming, save that they avoid errors. Sometimes they are necessary, and some types are indeed useful because they let us do many useful stuff. In this chapter we shall talk about these types and how to convert one to another.

The general form for type casting can be written :

```
1   val = type_function(value, optional_default_value = null )
```

How does this work? The system would try to cast *value* into the specific type. If it failed, and there is no default value, it would return *null*. However, if default is passed, it would return that when conversion fails. This neat design saves a tonnage of *try ... catch* stuff.

5.1 INTEGER FAMILY

This section is dedicated to the natural numbers family. We have :

1. bool : Boolean
2. char : Character family, automatic size.
3. int : Integer family, automatic size.
4. INT : BigInteger

Every type is under the hood is an object in ZoomBA, there is really no primitive here.

5.1.1 *Boolean*

The syntax is :

```
1   val = bool(value, optional_default_value = null )
2   val = bool(value, optional_matching_values[2])
```

Observe both in action :

```
1   val = bool("hello") // val is null
2   val = bool("hello",false) // val is false
3   val = bool('hi', ['hi' , 'bye' ] ) // val is true
4   val = bool('bye', ['hi' , 'bye' ] ) // val is false
```

5.1.2   Character

This is rarely useful - and the syntax is :

```
1   val = char(value, optional_default_value = null )
```

Usage is :

```
1   val = char("hello") // val is 'h'
2   val = char(1000, 'a') // converting from int -- val is 'a'
3   val = char(_'x', 'a') // val is 'x'
4   val = char(97, 'x') // converting from int -- val is 'a'
```

5.1.3   Integer

This is very useful and the syntax is :

```
1   val = int(value, optional_default_value = null )
```

Usage is :

```
1   val = int("hello") // val is null
2   val = int("hello",0) // val is 0
3   val = int(_"a",0) // converting from char - ordinal - val is 97
4   val = int('42') // val is 42
5   val = int(42) // val is 42
6   val = int ( 10.1 ) // val is 10
7   val = int ( 10.9 ) // val is 10
8   val = int('100',2, null ) // val is 4, ( string, base, default )
9   val = int(time()) // val is current epoch time in ms
```

5.1.4   Arbitrary Large Integer

This is sometimes required, and the syntax is :

```
1   val = INT(value, base=10, default_value = null )
```

Usage is :

```
1   val = INT("hello") // val is null
2   val = INT('hi',10,42 )// base 10, default 42, val is 42
3   val = INT('42') // val is 42
4   val = INT(54,13 ) // val is 42
5   val = INT(time()) // val is current epoch in nano sec :-)
```

5.2 RATIONAL NUMBERS FAMILY

This section is dedicated to the floating point numbers family. We have :

1. float : minimal container for floating point.
2. FLOAT : BigDecimal

5.2.1 *Float*

The syntax is :

```
1  val = float(value, optional_default_value = null )
```

Usage is :

```
1  val = float("hello") // val is null
2  val = float("hello",0) // val is 0.0
3  val = float('42') // val is 42.0
4  val = float(42) // val is 42.0
5  val = float ( 10.1 ) // val is 10.1
6  val = float ( 10.9 ) // val is 10.9
```

Note that, ZoomBA will automatically shrink floating point data into double, given it can fit the precision in :

```
1  val = 0.01 // val is a double type, automatic
```

5.2.2 *FLOAT*

This is sometimes required, and the syntax is :

```
1  val = FLOAT(value,default_value = null )
```

Usage is :

```
1  val = FLOAT("hello") // val is null
2  val = FLOAT('hi', 42 )// val is 42.0
3  val = FLOAT('42') // val is 42.0
4  val = FLOAT(42.00001 ) // val is 42.00001
```

5.3 GENERIC NUMERIC FUNCTIONS

5.3.1 *num()*

For generic numeric conversions, there is *num()* function, whose job is to convert data types into proper numeric type, with least storage. Thus :

```
1  val = num(value,default_value)
```

Usage is :

```
1  val = num("hello") // val is null
2  val = num('hi', 42 )// val is 42 int
3  val = num('42.00') // val is 42 int
4  val = num(42.00001 ) // val is 42.00001 double
5  val = num(42.000014201014 ) // val is double
6  // val is Real
7  val = num(42.00001420101410801980981092101 )
```

5.3.2 *size()*

size() function finds out digits length for an integer, in a given base:

```
1  x = 100
2  size(x) // 100 in 'unary' base : imagine 100s of 1's written
3  size(x,2) // 7, base 2
4  size(x,10) // 3 , base 10
5  size(x,16) // 2 , base 16
```

5.3.3 *Float to Integers*

To convert floating points to integers or pure rationals there are 3 functions : *floor(), ceil(), round()* as shown below:

```
1   // floor of f: max( all integers less than of equal to f )
2    x = floor(1.1) // x = 1
3    x = floor(0.0) // x = 0
4    x = floor(-1.5) // x = -2
5    // ceil of f: min ( all integers greater than or equal to f )
6    y = ceil(1.1) // x = 2
7    y = ceil(-1.1) // x = -1
8    y = ceil(-1.0) // x = -1
9    // now round function, with number of digit after decimal precision
10   round(0.5) // 0 digits, produces 1
11  round(0.51239,3) // 0.512
12  round(0.51239,4) // 0.5124
```

These functions takes care of the sign of the number also, and adjusted.

5.3.4 *Signs and Absolute*

Sign function for any number is defined as :

```
1  def _sign_(x ){
2     if ( x < 0 ) return -1
3     if ( x > 0 ) return 1
4     0 // else return 0
5  }
6  // or we can use :
7  sign(x) // same as _sign_ function
```

Absolute of a number is given by either the *size()* or the operator #|x| to be read as mod-x.

```
1  x = -100
```

```
2  #|x| // 100, absolute value of x
3  size(x) // 100, same
4  #|null| // 0, mod of nothing is 0
5  size(null) // -1, shows that the container is un-initialized
```

5.3.5 *log()*

log() function takes logarithm of a number in Napier's base, by default, or if base is provided, on that base:

```
1  x = 100
2  log(x) // 4.605170185988091368035982909368728 // BigDecimal
3  log(100,10) // 2 Integer
```

5.4 THE CHRONO FAMILY

Handling date and time has been a problem, that too with timezones. ZoomBA simplifies the stuff. We have Java 8 DateTime to handle date/time:

5.4.1 *time()*

This is how you create a DateTime LocalDateTime:

```
1  val = time([ value, date_format , time_zone] )
```

With no arguments, it gives the current date time:

```
1  today = time()
```

The default date format is $yyyyMMdd$, so :

```
1  dt = time('20160218') // 2016-02-18T00:00:00.000+05:30
```

For all the date formats on dates which are supported, see DateTimeFormat.
Take for example :

```
1  dt = time('2016/02/18', 'yyyy/MM/dd' )
2  // dt := 2016-02-18T00:00:00.000+05:30
3  dt = time('2016-02-18', 'yyyy-MM-dd' )
4  // dt := 2016-02-18T00:00:00.000+05:30
```

5.4.2 *Adjusting Timezones*

The *at()* function of the *ZDate* object handles the timezones properly. If you want to convert one time to another timezone, you need to give the time, and the timezone:

```
1  dt = time()
2  dt_honolulu = dt.at( 'Pacific/Honolulu' )
3  // dt_honolulu := 2016-02-17T17:23:02.754-10:00
4  dt_ny = dt.at(  'America/New_York' )
```

```
5  // dt_ny := 2016-02-17T22:23:02.754-05:00
```

There is a *date()* function to convert into a *Date* object in older Java, as well as timezone:

```
1  t = time()
2  dt = t.date() // convert into Date
3  dt_honolulu = t.date( 'Pacific/Honolulu' ) // timezone and covert
```

5.4.3   *Comparison on Chronos*

All these date time types are freely mixable, and all comparison operations are defined with them. Thus :

```
1  t1 = time()
2  thread().sleep(1000)  // wait for some time
3  t2 = time()
4  // now compare
5  c = ( t1 < t2 ) // true
6  c = ( t2 > t1 ) // true
```

Tow dates can be equal to one another, but not two instances, that is a very low probability event, almost never. Thus, equality makes sense when we know it is date, and not instant :

```
1  t1 = time('19470815') // Indian Day of Independence
2  t2 = time('19470815') // Indian Day of Independence
3
4  // now compare
5  c = ( t1 == t2 ) // true
```

5.4.4   *Arithmetic on Chronos*

Dates, obviously can not be multiplied or divided. That would be a sin. However, dates can be added with other reasonable values, dates can be subtracted from one another, and *time*() can be added or subtracted by days, months or even year. More of what all date time supports, see the manual of LocalDateTime.

To add time to a date, there is another nifty method :

```
1  d = time('19470815') // Indian Day of Independence
2  time_delta_in_millis = 24 * 60 * 60 * 1000 // next day
3  nd = time( d.time + time_delta_in_millis )
4  // nd := Sat Aug 16 00:00:00 IST 1947
```

Same can easily be achieved by the Durations string :

```
1  d = time('19470815') // Indian Day of Independence
2  nd = d + 'P1D' // 1947-08-16T00:00:00.000+05:30
```

And times can be subtracted :

```
1  d = time('19470815') // Indian Day of Independence
2  nd = d + 'P1D' // 1947-08-16T00:00:00.000+05:30
```

```
3  diff = nd - d
4  //diff := PT24H ## Duration between two dates!
5  diff.toMillis // long number of millisecs
```

So we can see it generates Duration gap between two chrono instances.

5.5 STRING : USING STR

Everything is just a string. It is. Thus every object should be able to be converted to and from out of strings. There are types of strings in ZoomBA:

5.5.1 *Types of Strings*

There are 3 types of strings:

```
1  general_string = "I am a string"
2  exec_string = #"2 + 3" // produces string 5
3  sub_string = #"variable substitution with #{exec_string}"
4  // append scripts folder path to the string
5  relocable_string = _/"some_path/some/other/path"
6  char = _"I" // character literal
```

Converting an object to a string representation is called Serialization, and converting a string back to the object format is called *DeSerialization*. This is generally done in ZoomBA by the function *str()*.

5.5.2 *Null*

str() never returns null, by design. Thus:

```
1  s = str(null)
2  s == 'null' // true
```

5.5.3 *Integer*

For general integers family, *str()* acts normally. However, it takes overload in case of $INT()$ or *BigInteger* :

```
1  bi = INT(42)
2  s = str(bi) // '42'
3  s = str(bi,2) // base 2 representation : 101010
4  s = str(bi,2,8) //8 bit base 2 representation : 00101010
```

5.5.4 *Floating Point*

For general floating point family, *str()* acts normally. However, it takes overload, which is defined as :

```
1  precise_string = str( float_value, num_of_digits_after_decimal )
```

To illustrate the point:

```
1  d = 101.091891011
2  str( d, 0 ) // 101
3  str(d,1) // 101.1
4  str(d,2) // 101.09
5  str(d,3) // 101.092
```

5.5.5 *Chrono*

Given a chrono family instance, *str*() can convert them to a format of your choice. These formats have been already discussed earlier, here they are again: SimpleDateFormat.

The syntax is :

```
1  formatted_chrono = str( chrono_value , chrono_format )
```

Now, some examples :

```
1  t = time()
2  str( t ) // default is 'yyyyMMdd' : 20160218
3  str( t , 'dd - MM - yyyy' ) // 18 - 02 - 2016
4  str( t , 'dd-MMM-yyyy' ) // 18-Feb-2016
```

5.5.6 *Collections*

Collections are formatted by str by default using ','. The idea is same for all of the collections, thus we would simply showcase some:

```
1  l = [1,2,3,4]
2  s = str(l) // '1,2,3,4'
3  s == '1,2,3,4' // true
4  s = str(l,'#') // '1#2#3#4'
5  s == '1#2#3#4' // true
```

So, in essence, for *str*(), serialising the collection, the syntax is :

```
1  s = str( collection [ ,  seperation_string ] )
```

5.5.7 *Generalised* toString()

This brings to the point that we can linearise a collection of collections using *str*(). Observe how:

```
1  l = [1,2,3,4]
2  m = [ 'a' , 'b' , 'c' ]
3  j = l * m // now this is a list of lists, really
4  s_form = str(j, '&') as { str($.o,'#')  } // lineraize
5  // This generates  1#a&1#b&1#c&2#a&2#b&2#c&3#a&3#b&3#c&4#a&4#b&4#c
```

5.5.8 *JSON*

Given we have a complex object comprise of primitive types and collection types, the *jstr()* function returns a JSON representation of the complex object. This way, it is very easy to inter-operate with JavaScript type of languages.

```
1  d = { 'a' : 10 , 'b' : 20  }
2  s = jstr(d) // as json string
3  /*  { "a" : 10 , "b" : 20 }  */
4  // pretty print json string using ZoomBA native capability
5  jstr(d,false)
6  jstr(d,true) // pretty print, using Jsoner capability
```

5.5.9 *Yaml*

Given we have a complex object comprise of primitive types and collection types, the *ystr()* function returns a Yaml representation of the complex object.

```
1  d = { 'a' : 10 , 'b' : 20  }
2  s = ystr(d) // as Yaml string
3  /* Yaml String ...
4  a: 10
5  b: 20
6  */
```

5.6 PLAYING WITH TYPES : REFLECTION

It is essential for a dynamic language to dynamically inspect types, if at all. Doing so, is termed as reflection. In this section, we would discuss different constructs that lets one move along the types.

5.6.1 *type() function*

The function *type*() gets the type of an object. In essence it gets the *getClass*() of any POJO. Notice that the class object is loaded only once under a class loader, so equality can be done by simply "==". Thus, the following constructs are of importance:

```
1  d = { : }
2  c1 = type(d) // c1 := java.util.HashMap
3  i = 42
4  c2 = type(i) // c2 := java.lang.Integer
5  c3 = type(20) // c3 := java.lang.Integer
6  c3 == c2 // true
7  c1 == c2 // false
```

5.6.2 *The === Operator*

From the earlier section, suppose someone wants to equal something with another thing, under the condition that both the objects are of the same type. Under the tenet of ZoomBA which reads : *"find a type at runtime, kill the type"* , we are pretty open into the type business :

```
1  a = [1,2,3]
2  ca = type(a) // ZArray
3  b = list(1,3,2)
4  cb = type(b) // ZList type
5  a == b // true
6  ca == cb // false
7  // type equals would be :
8  ca == cb  && a == b // false
```

But that is a long haul. One should be succinct, so there is this borrowed operator from JavaScript, known as "===" which let's its job in a single go :

```
1  a = [1,2,3]
2  b = list(1,3,2)
3  a === b // false
4  c = [3,1,2]
5  c === a // true
```

5.6.3   *The* isa *operator*

Arguably, finding a type and matching a type is a tough job. Given that we need to care about who got derived from whatsoever. That is an issue a language should take care of, and for that reason, we do have *isa* operator.

```
1   null isa null // true
2   a = [1,2,3]
3   a isa [] // true : a is a type of ZArray?
4   a isa [0] // true : a is a type of ZArray?
5   b = list(1,3,2)
6   b isa 'list' // true : b is a type of list ?
7   a isa b // false
8   {:} isa 'map'
9   true isa 'bool'
10  1 isa 'int'
11  1.0 isa 'float'
12  "abc".toCharArray() isa 'array'
```

This also showcase the issues of completely overhauling the type structure found in a JVM. The first two lines are significantly quirky.

There is a distinctly better way of handling *isa*, by using *patternmatcher* strings, strings that starts with ## and has a type information as expression match: e.g. ##*map#ig* :

```
1  b = list(1,3,2)
2  b isa ##list#ig // true : b is a type of list ?
3  {:} isa ##map#ig // true
4  4.2 isa ##double##ig // the double
5  s = set(1,2,3)
6  s isa ##set#ig // true
7  t = time()
8  t is a ##date#ig // true : for ZDate
```

This regex also is case insensitive, and matches from anywhere, so be careful.

5.6.4   *Field Access*

Once we have found the fields to access upon, there should be some way to get to the field, other than this :

```
1  d = time()
2  d.date().fastTime  // some value
```

There is obvious way to access properties as if the object is a dictionary, or rather than a property bucket:

```
1  d = time().date()
2  d['fastTime'] == d.fastTime  // true
```

This opens up countless possibilities of all what one can do with this. Observe however, monsters like spring and hibernate is not required given the property injection is this simple.

5.6.5   Nested Field Access

For a simple field like this, it is good. But what happens when we need to access to nested properties? For example, take this :

```
1  m = { "a" : { "b" : 10 }, "c" : { "b" : 20 } }
```

Of course, it can actually be a proper java object (we are simply using JSON map). How to access upto the level 'm.a.b' ? Or rather, how to get back the list of all nested fields "b" ? This is the precise reason ZoomBA has *xpath()* and *xelem()* functions.

One can actually parameterize the access "m.a.b" using the *xpath()* function:

```
1  m = { "a" : { "b" : 10 }, "c" : { "b" : 20 } }
2  // from "m" get me the path b, followed by a property
3  v = xpath(m, "/a/b" )  // v := 10
4  v = xpath(m, "/a/x" )  // v := null
```

What about there are multiple matches? This is done by optional third argument sending a boolean 'true' :

```
1  // from "m" get me the path b, followed by a property, as list, true
2  v = xpath(m, "//b", true )  // v := [10, 20]
```

But this pose a little problem of the setting property side of the things. Moreover, the power to traverse the object tree wanes off with only values. For this, there is the *xelem()* function:

```
1  // from "m" get me the path b, followed by a property, as list, true
2  p = xelem(m, "/a/x" ) // p := /.[@name='a'][@name='x'] //
       DynamicPropertyPointer
```

Now we can readily set any value pointed by 'p':

```
1  p.value = 42
2  // now if we print m :
3  println(m) // prints {"a" : {"b" : 10, "x" : 42}, "c" : {"b" : 20} }
```

xelem() also takes another optional parameter to get back a list of pointers:

```
1  // from "m" get me the path b, followed by a property, as list, true
2  pa = xelem(m, "//b", true )
3  // pa := [ /.[@name='a'][@name='b'],/.[@name='c'][@name='b'] ]
```

And now we can go about merry ways of getting and setting values in them.

REUSING CODE

The tenet of ZoomBA is : *"write once, forget"*. That essentially means that the code written has to be sufficiently robust. It also means that, we need to rely upon code written by other people.

How does this work? The system must be able to reuse *any* code from Java SDK, and should be able to use any code that we ourselves wrote. This chapter would be elaborating on this.

6.1 THE IMPORT DIRECTIVE

6.1.1 *Syntax*

Most of the languages choose to use a non linear, tree oriented import. ZoomBA focuses on minimising pain, so the import directive is linear. The syntax is simple enough :

```
1  import 'import_path' as unique_import_identifer
```

The directive imports the *stuff* specified in the *import_path* and create an alias for it, which is :
unique_import_identifer, Thus, there is no name collision at all in the imported script. This *unique_import_identifer* is known as the namespace.

6.1.2 *Examples*

Observe, if you want to import the class *java.lang.Integer* :

```
1  import 'java.lang.Integer' as Int // works, string literal
2  import java.lang.Integer as Int // works, standard form
3  x = 'java.lang.Integer' // well, programmatic access
4  import x as Int // works, again. Try avoiding it, please.
```

This directive would import the class, and create an alias which is *Int*. To use this class further, now, we should be using :

```
1  Int.parseInt('20') // int : 20
2  Int.valueOf('20') // int : 20
```

In the same way, one can import an ZoomBA script :

```
1   // w/o any extension it would find the script automatically
2   import 'from/some/folder/awesome' as KungFuPanda
3   // call a function in the script
4   KungFuPanda.showAwesomeness()
```

6.2 using existing java classes

6.2.1 *Load Jar and Create Instance*

The example of outer class, or rather a proper class has been furnished already. So, we would try to import a class which is not there in the $CLASS\_PATH$ at all.

```
1   // put all dependencies of xmlbeans.jar in there
2   success = load('path/to/xmlbeans_jar_folder') // true/false
3   import 'org.apache.xmlbeans.GDate' as AGDate
```

Once we have this class now, we can choose to instantiate it, See the manual of this class here:

```
1   //create the class instance : use new()
2   gd1 = new ( AGDate, date() )
3   // 2016-02-18T21:13:19+05:30
4   // or even this works
5   gd2 = new ( 'org.apache.xmlbeans.GDate' , date() )
6   // 2016-02-18T21:13:19+05:30
```

And thus, we just created a Java class instance. This is how we call Java objects, in general from ZoomBA.

Calling methods now is easy:

```
1   cal = gd1.getCalendar() // calls a method
2   /* 2016-02-18T21:13:19+05:30 */
```

Note that thanks to the way ZoomBA works, a method of the form $getXyz$ is equivalent to a field call xyz so :

```
1   cal = gd1.calendar // calls the method but like a field!
2   /* 2016-02-18T21:13:19+05:30 */
```

6.2.2 *Import Enum*

Enums can be imported just like classes. However, to use one, one should use the *enum*() function:

```
1   // creates a wrapper for the enum
2   rms = enum('java.math.RoundingMode')
3   rms.0 // UP
4   rms.UP // UP
```

The same thing can be achieved by :

```
1  // gets the value for the enum
2  v = enum('java.math.RoundingMode',
3          'UP' )
4  v = enum('java.math.RoundingMode', 0 )
```

6.2.3 *Import Static Field*

Import lets you import static fields too. For example :

```
1  // put all dependencies of xmlbeans.jar in there
2  import 'java.lang.System.out' as OUT
3  // now call println
4  OUT.println("Hello,World") // prints it!
```

However, another way of achieving the same is :

```
1  // put all dependencies of xmlbeans.jar in there
2  import 'java.lang.System' as SYS
3  // now call println
4  SYS.out.println("Hello,World") // prints it!
5  // something like reflection
6  SYS['out'].println("Hello,World") // prints it!
```

6.2.4 *Import Inner Class or Enum*

Inner classes can be accessed with the "$" separator. For example, observe from the SDK code of HashMap, that there is this inner static class *EntrySet*. So to import that:

```
1  // note that $ symbol for the inner class
2  import 'java.util.HashMap$EntrySet' as ES
```

6.3 using zoomba scripts

We have already discussed how to import an ZoomBA script, so in this section we would discuss how to create a re-usable script.

6.3.1 *Creating a Script*

We start with a basic script, let's call it *hello.zm* :

```
1  /* hello.zm */
2  def say_hello(arg){
3      println('Hello, ' + str(arg) )
4  }
5  s = "Some one"
6  say_hello(s)
7  /* end of script */
```

This script is ready to be re-used.

6.3.2 *Relative Path*

Suppose now we need to use this script, from another script, which shares the same folder as the "hello.zm". Let's call this script *caller.zm*. So, to import "hello.zm" in *caller.zm* we can do the following :

```
1  import './hello' as Hello
```

but, the issue is when the runtime loads it, the relative path would with respect to the runtimes run directory. So, relative path, relative to the caller would become a mess. To solve this problem, relative import is invented.

```
1  import _/'hello' as Hello
2  Hello.say_hello('Awesome!" )
```

In this scenario, the runtime notices the "_/", and starts looking from the directory the *caller.zm* was loaded! Thus, without any $PATH$ hacks, the ZoomBA system works perfectly fine.

6.3.3 *Calling Functions*

Functions can be called by using the namespace identifier, the syntax is :

```
1  import 'some/path/file' as NS
2  NS.function(args,... )
```

If one needs to call the whole script, as a function, that is also possible, and that is done using the *execute*() function:

```
1  import 'some/path/file' as NS
2  NS.execute(args,... )
```

We will get back passing arguments to a function in a later chapter. But in short, to call *hello.zm*, as a function, the code would be :

```
1  import _/'hello' as Hello
2  Hello.execute()
3  // or if you like succinct stuff
4  Hello()// yep, good to go...
```

Just as python has the implicit variable *__name__* ==' *__main__*' to identify if a script is running as 'main' script or not, ZoomBA has the implicit variable $@SCRIPT$. This stores the current script. Thus :

```
1  @SCRIPT.main // true if it is main script, false it is not
2  @SCRIPT.location // location of the currently running script
```

Thus, say we have two files 'a.zm' and 'b.zm' as follows:

```
1  // a.zm
2  import 'b' as B
3  B.say_something("hello!") // this is how you call a function
```

```
4
5  // b.zm
6  def say_something(arg){
7    println("I am from B!" + arg )
8  }
9  // body will be executed only when the script is main
10 if ( @SCRIPT.main ){
11   say_something(@ARGS)
12 }
```

Note the omission of '.zm' while importing.

When we run this code, we see : 'I am from B!hello!' This is how we call functions from other imported modules.

Point to be noted that the body of the imported script gets executed while importing, thus, any stray method calls not protected by the '@SCRIPT.main' functionality will be executed. For example, if we have the 'b.zm' to be modified by :

```
1  // b.zm
2  println("I am being called - from B!")
3  def say_something(arg){
4    println("I am from B!" + arg )
5  }
6  // body will be executed only when the script is main
7  if ( @SCRIPT.main ){
8    say_something(@ARGS)
9  }
```

Now, if we run the 'a.zm' file we see the line "I am being called - from B!". This has very interesting implications. It means all executable blocks will be executed, and all variables which are associated with the imported scripts block, will be initialised and accessible from the importing module. For all practical purpose, an imported module acts same as an object :

```
1  // a.zm
2  import 'b' as B
3  println( B.TI ) // this is how you access free variables defined on imported
         module
4
5  // b.zm
6  TI = time() // time of import ?
7  if ( @SCRIPT.main ){
8    println("From B : " + TI)
9  }
```

Naturally, we can extend an module import with initialisation of arguments, which will not be trivial, and hence, there is no need to invent Objects over imported modules.

Declarative Functional Style

Declarative Functional style is a goal for ZoomBA, in essence it boils down to some tenets:

1. Embrace **functional style** - a bit:
    a. Functions as first class citizens, they can be used as variables.
    b. Avoid modifying objects by calling methods on them. Only method calls returning create objects.
    c. Immutability of sorts, and referential transparency.
2. Avoid being imperative of sorts - known as being **Declarative** :
    a. Avoid conditional statements, replace them with alternatives.
    b. Avoid explicit iteration, replace them with higher order functions.
    c. Attain CSK complexity - write minimal code to solve a problem

See this SO Link for more ideas.

As we shall see there is a built-in facility to do almost anything (Pareto Optimal) in ZoomBA.

7.1 functions : in depth

As the functional style is attributed to functions, in this section we would discuss functions in depth.

7.1.1 *Function Types*

ZoomBA has 3 types of functions.

1. Explicit Functions : This functions are defined using the *def* keywords. They are the *normal* sort of functions, which everyone is aware of. As an example take this :

```
1  def my_function( a, b ){ a + b } // named function
2  plus = def(a,b){ a+b } // anonymous function assigned to a variable
3  succ = def(n){ n+1 } // same
4  pred = def(n){ n-1 } // same
5  d = { 's' :  def(n){ n+1 } , 'p' : def(n){ n-1 } } // treat them regular
6  a = [def(n){ n+1 } , def(n){ n-1 }] // natural
```

2. Anonymous Functions : This functions are defined as a side-kick to another function, other languages generally calls them Lambda functions, but they do very specific task, specific to the host function. All the collection comprehension functions takes them :

```
1  l = list([0:10] ) as { $.o ** 2}
2  def sq(x) { x ** 2 }
3  l = list([0:10] ) as sq // same
4  // same with a name-less function ( against anonymous )
5  l = list([0:10]) as def(_index, _item,_partial,_context) {   _item** 2 }
6  // note that , all arguments are optional, so :
7   l = list([0:10]) as def() { item = @ARGS[1] ; item **2  } // works too
```

3. Implicit Functions : This functions are not explicitly defined functions, they are the whole script body, to be treated as functions. Importing a script and calling it as a function qualifies as one :

```
1  import 'foo' as FOO
2  FOO()
```

7.1.2 *Default Parameters*

Every defined function in ZoomBA is capable of taking default values for the parameters. For example :

```
1  // default values passed
2  def my_function( a = 40 , b = 2 ){ a + b }
3  // call with no parameters :
4  println ( my_function() ) // prints 42
5  println ( my_function(10) ) // prints 12
6  println ( my_function(1,2) ) // prints 3
```

Note that, one can not mix the default and non default arbitrarily. That is, all the default arguments must be specified from the right side. Thus, it is legal :

```
1  // one of the default values passed
2  def my_function( a, b = 2 ){ a + b }
```

But it is not :

```
1  // default values passed in the wrong order
2  def my_function( a = 10 , b ){ a + b }
```

7.1.3 *Named Arguments*

In ZoomBA, one can change the order of the parameters passed, provided one uses the named arguments. See the example:

```
1  def my_function( a , b  ){ printf('(a,b) = (%s ,%s)\n' ,a , b ) }
2  my_function(1,2)  // prints (a,b) = (1,2)
3  my_function(b=11,a=10) //  prints (a,b) = (10,11)
```

Note that, named args can not be mixed with unnamed args, that is, it is illegal to call the method this way :

```
1  // only one named values passed
2  my_function(b=11,10) //  illegal
```

7.1.4 Arbitrary Number of Arguments

Every ZoomBA function can take arbitrary no of arguments. To access the arguments, one must use the *@ARGS* construct, as shown :

```
1  // this can take any no. of arguments
2  def my_function(){
3      // access the arguments, they are collection
4      s = str ( @ARGS , '#' )
5      println(s)
6  }
7  my_function(1,2,3,4)
8  /* prints 1#2#3#4 */
```

This means that, when a function expects n parameters, but is only provided with $m < n$ parameters, the rest of the expected parameters not passed is passed as *null*.

```
1  // this can take any no. of arguments, but named are 3
2  def my_function(a,b,c){
3      // access the arguments, they are collection
4      s = str ( @ARGS , '#' )
5      println(s)
6  }
7  my_function(1,2)
8  /* prints 1#2#null */
```

In the same way when a function expects n parameters, but is provided with $m > n$ parameters, the rest of the passed parameters can be accessed by the *@ARGS* construct which was the first example.

7.1.5 Arguments Overwriting

How to generate permutations from a list of object with ^nP_r ? Given a list l of size 3 We did $^3P_3$:

```
1  l = ['a','b','c' ]
2  perm_3_from_3 = join(l,l,l) where { #|set($.o)| == #|$.o| }
3  l = ['a','b','c' ,'d' ]
4  perm_4_from_4 = join(l,l,l,l) where { #|set($.o)| == #|$.o| }
5  perm_2_from_4 = join(l,l) where { #|set($.o)| == #|$.o| }
```

Thus, how to generate the general permutation? As we can see, the arguments to the permutation is always varying. To fix this problem, *argument overwriting* was invented. That, in general, all the arguments to a function can be taken in runtime from a collection.

```
1  l = ['a','b','c' ,'d' ]
2  // this call overprintlns the args :
3  perm_2_from_4 = join( @ARGS = [l,l] ) where{ #|set($.o)| == #|$.o| }
```

Thus, to collect and permutate r elements from a list l is :

```
1  perm_r = join(@ARGS = list([0:r]) as {l}) where{ #|set($.o)| == #|$.o| }
```

and we are done. This is as declarative as one can get.

7.1.6 *Recursion*

It is customary to introduce recursion with factorial. We would not do that, we would introduce the concept delving the very heart of the foundation of mathematics, by introducing Peano Axioms. Thus, we take the most trivial of them all : Addition is a function that maps two natural numbers (two elements of N) to another one. It is defined recursively as:

```
1  /* successor function */
2  def s( n ){
3      if ( n == 0 ) return 1
4      return  ( s(n-1) + 1 )
5  }
6  /* addition function */
7  def add(a,b){
8    if ( b == 0 ) return a
9    return s ( add(a,b-1) )
10 }
```

These functions do not only show the trivial addition in a non trivial manner, it also shows that the natural number system is recursive. Thus, a system is recursive if and only if it is in 1-1 correspondence with the natural number system. Observe that the function *add()* does not even have any addition symbol anywhere! This is obviously cheating, because one can not use minus operator to define plus operation. A better less cheating solution is to use function composition and power operator : $a + b := s(...s(s(a)))$ b times, that is $s^b(a)$, which we would discuss later.

Now we test the functions:

```
1  println(s(0)) // prints 1
2  println(s(1)) // prints 2
3  println(s(5)) // prints 6
4  println(add(1,0)) // prints 1
5  println(add(1,5)) // prints 6
6  println(s(5,1)) // prints 6
```

Now, obviously we can do factorial :

```
1  def fact(n){
2      if ( n <= 0 ) return 1
3      return n * fact( n - 1 )
4  }
```

7.1.7 *LRU Cache*

The advent of recursion meant draining of stack and computational power. This can be mitigated by memoizing the results of compute. Of course one can create a map to cache the output result against serialised form of the input but that is too much code.

For example:

```
1  def fib(n){
2    if ( n <=1 ) return n
3    return fib(n-1) + fib(n-2)
4  }
```

we can optimise it as:

```
1  __mem = dict()
2  def fib(n){
3    if ( n <=1 ) return n
4    if ( n @ __mem ) return __mem[n]
5    __mem[n] =  fib(n-1) + fib(n-2)
6  }
```

The dict will keep on expanding.

But ZoomBA can do a bit better, with a LRU cache, one can simply set the method level cache as follows:

```
1  #(t1, o) = #clock{  fib(20) }
2  printf("W/O Caching : %f %n", t1)
3  @SCRIPT.fib.cache("all")
4  #(t2, o) = #clock{  fib(20) }
5  printf("With Caching : %f %n", t2)
6  assert ( t1 > t2  , "Method Caching Did not help!"
```

7.1.8 Closure

A closure is a function, whose return value depends on the value of one or more variables declared outside this function. Consider the following piece of code with anonymous function:

```
1  multiplier = def(i) { i * 10 }
```

Here the only variable used in the function body, $i * 0$, is i, which is defined as a parameter to the function. Now let us take another piece of code:

```
1  multiplier = def (i) {  i * factor }
```

There are two free variables in multiplier: i and *factor*. One of them, i, is a formal parameter to the function. Hence, it is bound to a new value each time multiplier is called. However, *factor* is not a formal parameter, then what is this? Let us add one more line of code:

```
1  factor = 3
2  multiplier = def (i) {  i * factor }
```

Now, factor has a reference to a variable outside the function but in the enclosing scope. Let us try the following example:

```
1    println ( multiplier(1) ) // prints 3
2    println ( multiplier(14) ) // prints 42
```

Above function references factor and reads its current value each time. If a function has no external references, then it is trivially closed over itself. No external context is required.

7.1.9 Partial Function

Partial functions are functions which lets one implement closure, w/o getting into the global variable way. Observe :

```
1   // this shows the nested function
2   def func(a){
3       // here it is : note the name-less-ness of the function
4       r = def(b){ // which gets assigned to the variable "r"
5           printf("%s + %s ==> %s\n", a,b,a+b)
6       }
7       return r // returning a function
8   }
9   // get the partial function returned
10  x = func(4)
11  // now, call the partial function
12  x(2)
13  //finally, the answer to life, universe and everything :
14  x = func("4")
15  x("2")
```

This shows nested functions, as well as partial function.

7.1.10   *Functions as Parameters : Lambda*

From the theory perspective, lambdas are defined in Lambda Calculus. As usual, the jargon guys made a fuss out of it, but as of now, we are using lambdas all along, for example :

```
1   list(1,2,3,4) as { $.o ** 2 }
```

The {$.*o* ∗ ∗2} is a lambda. The parameter is implicit albeit, because it is meaningful that way. However, sometimes we need to pass named parameters. Suppose we now want to create a function composition, first step would be to apply it to a single function :

```
1   def  apply ( param , a_function ){
2       a_function(param)
3   }
```

So, suppose we want to now apply arbitrary function :

```
1   apply( 10, def(a){ a** 2 }  )
```

And now, we just created a lambda! The result of such an application would make apply function returning 100.

7.1.11   *Composition of Functions*

To demonstrate a composition, observe :

```
1   def compose (param,  a_function , b_function ){
2       // first apply a to param, then apply b to the result
3       // b of a
4       b_function ( a_function( param ) )
5   }
```

Now the application :

```
1   compose( 10, def(a){ a ** 2 } ,  def(b){ b - 58 } )
```

As usual, the result would be 42!

Now, composition can be taken to extreme ... this is possible due to the arbitrary length argument passing :

```
1  def compose(value, func_list){
2      for ( f : func_list ){
3          value = f(@ARGS=[value])
4      }
5      value // return it
6  }
```

Formally this is a potential infinite function composition! Thus, this call :

```
1  // note the nameless-ness :)
2  r = compose(6, [def (a){ a** 2} , def (b){ b + 6 }] )
3  println(r)
```

generates, the answer to life, universe, and everything, as expected.

7.1.12 *Operation on Function*

Functions support composition to generate a new function : The operator for function composition :

$$\# < f|g > (x) := g(f(x))$$

Let's have a simple demonstration :

```
1  def f(){
2      println(str(@ARGS,'#'))
3      @ARGS[0] ** 2
4  }
5  def g(){
6      println(str(@ARGS,'#'))
7      @ARGS[0] - 2
8  }
9  h = #< f | g >  // g of f
10 println( h(10) )
```

The result is :

```
10
100
98
```

What is happening under the hood? A method is being "created" on the fly that is the *composition* of the methods. Also note that we can do exactly the same with anonymous methods too:

```
1  h = #< {  $.0 ** 2 } | { $.0 - 2 }  > // data pipeline
2  println( h(10) ) // 98
```

We must understand the argument transfer and wrapping around is pretty automatic. But in rare cases where we want to be explicit about that we want to *explode* the return value into a list itself - we can use the explode operator "*" as follows:

```
1  h =  #< { $.0.toCharArray }* | minmax * | {$.1} >
2  println( h("Hello!")) // the letter 'o' comes in
```

As we can see, we can mix expression as well as predefined functions freely.

7.1.13 *Eventing*

All ZoomBA functions are Eventing ready. What are events? For all practical purposes, an event is nothing but a hook before or after a method call. This can easily be achieved by the default handlers for ZoomBA functions. See the object Event.

Here is a sample demonstration of the eventing:

```
1  def my_func(){
2      printf('ARGS : %s %n', str(@ARGS,'#'))
3  }
4  def event_hook(event){
5      printf('%s %s %n', event.method.name , event.when )
6  }
7  my_func.before += event_hook
8  my_func.after += event_hook
9  my_func('hello', 'world')
```

When one runs it, this is what we will get :

```
my_func BEFORE
ARGS : hello#world
my_func AFTER
```

7.2 STRINGS AS FUNCTIONS : CURRYING

All these idea started first with Alan Turing's Machine, and then the 3rd implementation of a Turing Machine, whose innovator Von Neumann said data is same as executable code. Read more on : Von Neumann Architecture. Thus, one can execute arbitrary string, and call it code, if one may. That brings in how functions are actually executed, or rather what are functions.

7.2.1 *Rationale*

The idea of Von Neumann is situated under the primitive notion of alphabets as symbols, and the intuition that any data is representable by a finite collections of them. The formalisation of such an idea was first done by Kurt Godel, and bears his name in Gödelization.

For those who came from the Computer Science background, thinks in terms of data as binary streams, which is a general idea of Kleene Closure : $\{0, 1\}^*$. Even in this form, data is nothing but a binary String.

Standard languages has String in code. In 'C' , we have "string" as "constant char*". C++ gives us std:string , while Java has "String". ZoomBA uses Java String. But, curiously, the whole source code, the entire JVM assembly listing can be treated as a String by it's own right! So, while codes has string, code itself is nothing but a string, which is suitable interpreted by a machine, more generally known as a Turing Machine. For example, take a look around this :

```
(zoomba)println('Hello, World!')
```

This code printlns 'Hello, World!' to the console. From the interpreter perspective, it identifies that it needs to call a function called println, and pass the string literal "Hello, World!" to it. But observe that the whole line is nothing but a string.

This brings the next idea, that can strings be, dynamically be interpreted as code? When interpreter reads the file which it wants to interpret, it reads the instructions as strings. So, any string can be suitable interpreted by a suitable interpreter, and thus data which are strings can be interpreted as code.

7.2.2 *Minimum Description Length*

With this idea, we now consider this. Given a program is a String, by definition, can program complexity be measured as the length of the string proper? Turns out one can, and that complexity has a name, it is called : Chatin Solomonoff Kolmogorov Complexity.

The study of such is known as Algorithmic Information Theory and the goal of a software developer becomes to println code that reduces this complexity.

As an example, take a look about this string :

```
(zoomba)s = 'abababababababababababababababab'
=>abababababababababababababababab  ## String
(zoomba)#|s|
=>34 ## Integer
```

Now, the string 's' can be easily generated by :

```
(zoomba)s1 = 'ab'**17
=>abababababababababababababababab  ## String
(zoomba)s == s1
=>true  ## Boolean
```

Thus, in ZoomBA, the minimum description length of the string 's' is : 8 :

```
(zoomba)code="'ab'**17"
=>'ab'**17  ## String
(zoomba)#|code|
=>8  ## Integer
```

7.2.3 *Examples*

First problem we would take is that of comparing two doubles to a fixed decimal place. Thus:

```
(zoomba)x=1.01125
=>1.01125
(zoomba)y=1.0113
=>1.0113
(zoomba)x == y
=>false
```

How about we need to compare these two doubles with till 4th digit of precision (rounding) ? How it would work? Well, we can use String format :

```
(zoomba)str("%.4f",x)
=>1.0113
(zoomba)str("%.4f",y)
=>1.0113
```

But wait, the idea of precision, that is ".4" should it not be a parameter? Thus, one needs to pass the precision along, something like this :

```
(zoomba)c_string = "%%.%df"  ## This is the original one
=>%%.%df
## apply precision, makes the string into a function
(zoomba)p_string = str(c_string,4)
=>%.4f
## apply a number, makes the function evaluate into a proper value
(zoomba)str(p_string,y)
=>1.0113  # and now we have the result!
```

All we really did, are bloated string substitution, and in the end, that produced what we need. Thus in a single line, we have :

```
(zoomba)str(str(c_string,4),x)
=>1.0113
(zoomba)str(str(c_string,4),y)
=>1.0113
```

In this form, observe that the comparison function takes 3 parameters :
1.   The precision, int, no of digits after decimal
2.   Float 1
3.   Float 2

as the last time, but at the same time, the function is in effect generated by application of partial functions, one function taking the precision as input, generating the actual format string that would be used to format the float further. These sort of taking one parameter at a time and generating partial functions or rather string as function is known as Currying, immortalized the name of Haskell Curry, another tribute to him is the name of the pure functional language Haskell.

Now to the 2nd problem. Suppose the task is given to verify calculator functionality. A Calculator can do '+' , '-', '*' , ... etc all math operations. In this case, how one proposes to println the corresponding test code? The test code would be, invariably messy :

```
1  if ( operation == '+' ) {
2      do_plus_check();
3  } else if ( operation == '-' ) {
4      do_minus_check();
5  }
6  // some more code ...
```

In case the one is slightly smarter, the code would be :

```
1  switch ( operation ){
2      case '+' :
3          do_plus_check(); break;
4      case '-' :
5          do_minus_check(); break;
6      ...
7  }
```

The insight of the solution to the problem is finding the following :

We need to test something of the form we call a "binary operator" is working "fine" or not:

$$operand_1 \ < operator > \ operand_2$$

That is a general binary operator. If someone can abstract the operation out - and replace the operation with the symbol - and then someone can actually execute that resultant string as code (remember JVMA?) the problem would be solved. This is facilitated by the back-tick operator (executable strings) :

```
(zoomba)c_string = #'#{a} #{op} #{b}'
=>#{a} #{op} #{b}
(zoomba)a=10
=>10
(zoomba)c_string = #'#{a} #{op} #{b}'
=>10 #{op} #{b}
(zoomba)op='+'
=>+
(zoomba)c_string = #'#{a} #{op} #{b}'
=>10 + #{b}
```

```
(zoomba)b=10
=>10
(zoomba)c_string = #'#{a} #{op} #{b}'
=>20
```

7.2.4   Reflection

Calling methods can be accomplished using currying.

```
1   import 'java.lang.System.out' as out
2   def func_taking_other_function( func ){
3       #"#{func}( 'hello!' )"
4   }
5   func_taking_other_function('out.println')
```

The *func* is just the name of the function, not the function object at all. Thus, we can use this to call methods using reflection.

7.2.5   Referencing

Let's see how to have a reference like behaviour in ZoomBA.

```
(zoomba)x = [1,2]
@[1, 2]
(zoomba)y = { 'x' : x }
{x=@[ 1,2 ]}
(zoomba)x = [1,3,4]
@[1, 3, 4]
(zoomba)y.x // x is not updated
@[1, 2]
```

Suppose you want a different behaviour, and that can be achieved using Pointers/References. What you want is this :

```
(zoomba)x = [1,2]
@[1, 2]
(zoomba)y = { 'x' : 'x'  }
{x=x}
(zoomba)#'#{y.x}' // access as in currying stuff
@[1, 2]
(zoomba)x = [1,3,4]
@[1, 3, 4]
(zoomba)#'#{y.x}' // as currying stuff, always updated
@[1, 3, 4]
```

So, in effect we are using a dictionary to hold name of a variable, instead of having a hard variable reference, thus, when we are dereferencing it, we would get back the value if such a value exists! In fact ZoomBA does have a much more clean *reference pointer* implementation which allows full reactive programming in ZoomBA.

```
1   ref_ex = #def{ x + y } // a ZoomBA Observable
2   x = 10 // now define x and y
3   y = 32 // there
4   println(ref_x) // 42
5   x = 100 // now change again
6   println(ref_x) // 132
```

Interestingly, all *assert,panic,test* methods has all their parameters passed as ZoomBA Observables. We will talk about this in a later chapter.

7.3 AVOIDING CONDITIONS

As we stated in the tenets of declarative functional programming was to avoid conditional statements. In the previous section, we have seen some of the applications, how to get rid of the conditional statements. In this section, we would see in more general how to avoid conditional blocks.

7.3.1 *Theory of the Equivalence Class*

Conditionals boils down to $if - else$ construct. Observe the situation for a valid date in the format of $ddMMyyyy$.

```
1   is_valid_date(d_str){
2      num = int(d_str)
3      days = num/1000000
4      rest = num % 1000000
5      mon = rest /10000
6      year = rest % 10000
7      max_days = get_days(mon,year)
8      return (   0 < days && days <= max_days  &&
9                 0 < mon && mon < 13 &&
10                year > 0 )
11  }
12  get_days(month,year){
13     if ( month == 2  and leap_year(year){
14        return 29
15     }
16     return days_in_month[month]
17  }
18  days_in_month = [31,28,31,... ]
```

This code generates a decision tree. The leaf node of the decision tree are called Equivalent Classes, their path through the source code are truly independent of one another. Thus, we can see there are 4 equivalence classes for the days:

1. Months with 31 days
2. Months with 30 days
3. Month with 28 days : Feb - non leap
4. Months with 29 days : Feb -leap

And the $if - else$ simply ensures that the correct path is being taken while reaching the equivalent class. Observe that for two inputs belonging to the same equivalent class, the code path remains the same. That is why they are called equivalent class.

Note from the previous subsection, that the days of the months were simply stored in an array. That avoided some unnecessary conditional blocks. Can it be improved further? Can we remove all the other conditionals too? That can be done, by intelligently tweaking the code. Observe, we can replace the ifs with this :

```
1   get_days(month,year){
2      max_days = days_in_month[month] +
3      ( (month == 2 and leap_year(year) )? 1 : 0 )
4   }
```

But some condition can never be removed even in this way.

7.3.2 *Dictionaries and Functions*

Suppose we are given the charter to verify a sorting function. Observe that we have already verified it, but this time, it comes with a twist, sorting can be both ascending and descending.

So, the conditions to verify ascending/descending are :

```
1  sort_a = !exists(collection) where { $.i > 0 and $.c[$.i - 1 ] > $.o }
2  sort_d = !exists(collection) where { $.i > 0 and $.c[$.i - 1 ] < $.o }
```

Note that both the conditions are the same, except the switch of > in the ascending to < in the descending. How to incorporate such a change? The answer lies in the dictionary and currying :

```
1  op  = { 'ascending' : '>' , 'descending' : '<'  }
2  sorted = !exists(collection) where {
3      $.i > 0 and #'$.c[$.i - 1 ] #{op} $.o' }
```

In this form, the code written is absolutely generic and devoid of any explicit conditions. Dictionaries can be used to store functions.

7.3.3 *An Application : FizzBuzz*

We already aquatinted ourselves with FizzBuzz. here, is a purely conditional version:

```
1   /* Pure If/Else */
2   def fbI(range){
3     for ( i : range ){
4         if ( i % 3 == 0 ){
5             if ( i % 5 == 0 ){
6                 println('FizzBuzz')
7             } else {
8                 println('Fizz')
9             }
10        }else{
11            if ( i % 5 == 0 ){
12                println('Buzz')
13            }else{
14                println(i)
15            }
16        }
17    }
18  }
```

However, it can be written in a purely declarative way. Here is how one can do it, Compare this with the other one:

```
1   def fbD(range){
2     d = { 0 : 'FizzBuzz' ,
3           3 : 'Fizz' ,
4           5 : 'Buzz' ,
5           6 : 'Fizz',
6          10 : 'Buzz' ,
7          12 : 'Fizz' }
8     for ( x : range ){
9         r = x % 15
10        continue ( r @ d ){ println( d[r] ) }
11        println(x)
```

```
12    }
13  }
```

7.4   AVOIDING EXPLICIT MUTABLE ITERATIONS

The last tenet is avoiding explicit mutable (*for,while*) loops, and thus in this section we would discuss how to avoid loops. We start with some functions we have discussed, and some functionalities which we did not.

7.4.1   *Range Objects in Detail*

The *for* loop becomes declarative, the moment we put a range in it. Till this time we only discussed part of the range type, only the numerical one. We would discuss the *Date* and the *Symbol* type range.

A date range can be established by :

```
1  d_start = time()
2  d_end = d_start + "PD10"
3  // a range from current to future
4  d_range = [d_start : d_end ]
5  // another range with 2 days spacing
6  d_range_2 = [d_start : d_end : 2 ]
```

and this way, we can iterate over the dates or rather time. The spacing is to be either an integer, in which case it would be taken as days, or rather in string as the ISO-TimeInterval-Format. The general format is given by : $PyYmMwWdDThHmMsS$.

The other one is the sequence of symbols, the Symbol range :

```
1  s_s = "A"
2  s_e = "Z"
3  // symbol range is inclusive
4  s_r = [s_s : s_e ]
5  // str field holds the symbols in a string
6  s_r.str == 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' // true
```

All the range types have some specific functions.

```
1  r = [0:5]
2  // a list of the elements
3  r.list()
4  // reverse of the same [a:b]
5  r.reverse() // [4, 3, 2, 1, 0]
6  // inverse of the range : reversed into [b:a]
7  r.inverse() // [5:0:-1]
```

There is the 'XRange'. X Range stands for extended range, which goes beyond standard size of JVM. This can be created by using long types for one or more end points of a range :

```
1  xr = [0L:50000000000000000000L] // yep, huge range
2  xrr = [0D:10D:0.00000001D]  // precise range
```

It is obvious that *list*() and related functions may or may not be possible with *XRange*.

7.4.2 *The Fold Functions*

The rationale of the fold function is as follows :

```
1  def do_something(item){ /* some code here */ }
2  for ( i : items ){
3      do_something(i)
4  }
5  // or use fold
6  fold(items) as { do_something($.o) }
```

The idea can be found here in a more elaborative way. We must remember that the partial exists, and the general fold function is defined as : $fold()$ folds from left, so it is also called $lfold()$.

```
1  // folds from left side of the collection
2  lfold(items[, seed_value ]) as { /* body */ }
3  // folds from right side of the collection
4  rfold(items[, seed_value ]) as { /* body */ }
```

First we showcase what right to left means in case of $rfold()$:

```
1  items = [0,1,2,3,4]
2  rfold( items ) as { printf( '%s ', $.o) }
```

This prints :

```
4 3 2 1 0
```

Let us showcase some functionalities using fold functions, we start with factorial :

```
1  fact_n = lfold( [2:n+1] , 1 ) as { $.p * $.o  }
```

We move onto find Fibonacci :

```
1  fib_n = fold([0:n+1],[0,1]) as { #(p, c) = $.p; [c, p + c]  }
```

We now sum the elements up:

```
1  sum_n = rfold([2:n+1], 0) as { $.p + $.o }
```

Generate a set from a list :

```
1  my_set = lfold( [1,2,1,2,3,4,5], set() ) as { $.p += $.o }
```

Finding min/max of a collection :

```
1  l = [1,8,1,2,3,4,5]
2  #(min,max) = lfold( l , [l.0, l.0] ) as {
3                  #(m,M)  = $.p
4                  continue( $.o > M ){ $.p = [ m, $.o] }
5                  continue( $.o < m ){ $.p = [ $.o, M] }
6                  $.p // return it off, when nothing matches
7              }
8  // (1,8)
```

Finding match index of a collection :

```
1  inx = fold( [1,2,1,2,3,4,5] , -1 ) as {
2    break ( $.o  > 3 ){  $.p = $.i } ; $.p
3  }
```

So, as we can see, the *fold* functions are the basis of every other functions that we have seen. We show how to replace select function using fold:

```
1  // selects all elements less than than equal to 3
2  selected = fold( [1,2,1,2,3,4,5] , list() ) as {
3              continue ( $.o  > 3 )
4              $.p += $.o
5          }
```

To do an *index*() function :

```
1  // first find the element greater than 3
2  selected = fold( [1,2,1,2,3,4,5] , -1){
3              break ( $.o  > 3 ) { $.i }
4              $.p }
```

And to do an *rindex*() function :

```
1  // first find the element less than 3
2  selected = rfold( [1,2,1,2,3,4,5] , -1){
3              break ( $.o  < 3 ) { $.i }
4              $.p }
```

7.4.3 *Matching*

There is *if*, and there is *else* and switch-case seems to be missing in ZoomBA. Fear not, the omniscient *#match* exists. To demonstrate :

```
1  // match  item
2  item = 10
3  selected = switch(item){
4              case 1 : "I am one"
5              case @$ < 5 :  "I am less than 5"
6              /* Default match is @$ itself, note that :
7              inside match you must use block comments */
8              case @$ : "yes, I could not match anything!"
9          }
```

Observe these important things about match:

1. Match works with case expression after colon, evaluating to *true* or expression using *Object.equals(expr,item)*. Thus, string o won't match 'o'. This is a design decision chosen to be compatible with switch case. To match such stuff, use @$ == *expr* , which would use ZoomBA expression evaluation.

2. The only comments which are allowed are block comments.

3. The @$ signifies the item, and thus a *case* like case @$ must be the last item.

4. Any expression is permitted in the case statement, *switch* returns the body value of the matching case. Thus, unlike ordinary switch-case, it returns a proper value. Thus, it is more succinct to put it at the end of a choice function.

7.4.4 *Sequences*

Iteration is key component of multitude of algorithms. Let's start with Factorial, Fibonacci, which are recursively defined. We did find out how to do it with basic function style. Observe, the recursion pattern $F_{n+1} = nF_n$, with base case $F(0) = 1$.

This is done by the *seq()* function in ZoomBA, which generates the iterator:

```
1  factorial = seq( 1 ) as { size( $.item ) *  $.item[-1]  }
2  for ( i : [ 1 : 10  ] ){
3    // Java Iterator: hasNext() and next() :: prints factorials of 1 to 9
4      printf( '%d %d %n' , i, factorial.next )
5  }
6  println( factorial.history ) // shows the history till this point
```

And that is how we can generate a never ending sequence of factorials. One has to remember that a sequence is a never ending iterator.

Moreover, we can see the 'history'. Formally, *seq()* is a function that is capable of computing fixed point iteration : $X_{n+1} = F(X_n, X_{n-1}, ...)$, each of these X_{n-i} can be tracked by the *$.item* term, or history.

While Factorial only use one past history term to calculate next, Fibonacci, uses 2:

$$Fib_n = Fib_{n-1} + Fib_{n-2}$$

with base cases $Fib(0) = Fib(1) = 1$ and a straightforward mapping is possible using sequences:

```
1  fib = seq( 1 , 1 ) as {  $.item[-1] + $.item[-2]  }
2  for ( i : [ 2 : 10  ] ){
3    printf( '%d %d %n' , i, fib.next )
4  }
5  println( fib.history )
```

It is customary to showcase at least 2 different ways of creating *primes* sequence which generates the next prime. First the *intuitive* way :

```
1  primes = seq(2,3) as {
2    for(test_prime=$.p[-1]+2;exists($.p)::{ $.o /? test_prime };test_prime+=2);
3    test_prime
4  }
```

And now, a bit succinct way :

```
1  primes = seq ( 2,3 ) ->{
2     prime_cache = $.prev
3     last = prime_cache[-1] + 1
4     // https://en.wikipedia.org/wiki/Bertrand%27s_postulate
5     end = last * 2
6     last + index([last:end:2])::{!exists(prime_cache)::{$.item /? $.$.item}}
7  }
```

It concludes the sequences.

Input and Output

IO is of very importance for software testing and business development. There are two parts to it, the generic IO, which deals with how to read/write from disk, and to and from url. There is another part of the data stuff, that is structured data. ZoomBA let's you do both, and those are the topic of this chapter.

8.1 reading

8.1.1 *read() function*

Reading is done by this versatile *read*() function. The general syntax is :

```
1  value = read(source_to_read)
```

The source can actually be a file, an UNC path, an url. Note that given it is a file, *read*() reads the whole of it at a single go, so, be careful with the file size.

```
1  value = read('my_file.txt') // value contains all the text in my_file.txt
2  lines = read('my_file.txt', true) // lines contains lines in the file
3  bytes = read('my_file.txt', 'b') // bytes contains all bytes of the file
4  next_line = read() // next line in stdin
5  all_text = read(false) // entire text at stdin as string
6  all_lines = read(true) // entire stdin as list of lines [strings]
```

To demonstrate – here is a command line interface loop:

```
1  while (  null !=( l = read() ){  /* do processing with l */ }
```

Here is the proverbial cat command:

```
1  println(read(false)) // stdin
2  println(read(file_name)) // cat the file_name
```

And that should tell you how much brevity the language wants to aspire to.

8.1.2 *Reading from url : HTTP GET*

Read can be used to read from an url. Simply :

```
1  resp = read('http://www.google.co.in')
2  /* value contains response of the google home page */
3  resp.body() // string of HTML
4  resp.bytes // response bytes
5  resp.code // responce code
6  resp.headers // response headers
```

To handle the timeouts, it comes with overloads:

```
1  // url, connection timeout, read timeout, all in millisec
2  resp = read('http://www.google.co.in', 10000, 10000 )
3  /* resp contains all the response data from google */
4  // it supports named parameters :
5  resp = read('path' = 'https://www.google.co.in',
6     'conTo'=1000, 'readTo'=1000, headers={:} )
7  // conTo : Connection Timeout, readTo : Read Timeout,
8  // headers : http headers you want to send
```

To generate the url to *get*, the fold function comes handy:

```
1  _url_ = 'https://httpbin.org/get'
2  params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
3  // str: is the namespace alias for 'java.lang.String'
4  data = str(params.entries,'&') as {
5     str('%s=%s',$.key,$.value)
6  }
7  response = read( str("%s?%s", _url_ , data ) )
```

And that is how you do a restful api call, using *get*.

8.1.3 *Reading All Lines*

The function *file*() reads all the lines, and puts them into different strings, so that it returns a list of strings, each string a line. With no other arguments, it defers the reading of next line, so it does not read the whole bunch together.

```
1  for ( line : file('my_file.txt') ){
2     println(line) // proverbial cat program
3  }
```

In short, the *file*() method yields an iterator. But with this, it reads the whole file together as list of string lines:

```
1  ll = read ( 'my_file.txt' , true )
```

This is again a synchronous call, and thus, it would be advisable to take care.

8.2 WRITING

8.2.1 *write(), println(), printf() function family*

We are already familiar with the write function. With a single argument, it writes the data back to the standard output. The function is aliased as *print()* which is same as *write()*:

```
1  println('Hello,World!') ; // prints it
2  printf("%d", 10) // formatted print
3  eprintln("This goes to error stream!" )
4  eprintf("This goes to error : %d", 10 ) // formatter
```

Given that the first argument does not contain a "%" symbol, the write functionality also writes the whole string to a file with the name :

```
1  // creates my_file.txt, writes the string to it
2  write('my_file.txt', 'Hello,World!')
3  // appends to the file
4  write('my_file.txt', 'Another line appended', true)
5  // write binary to file
6  write('my_binary_file.bin', 'This is string data as payload'.getBytes() )
7  // write to path, data, no append, if path does not exist : create,
8  write('/a/b/c/file.txt', 'This is string data as payload', false, true )
```

The formatting guide for *printf()* can be found here.

8.3 FILE PROCESSING

We talked about a bit of file processing in *read* and *write*. But if we want to read a file line by line, then we have to use something else.

8.3.1 *open() function*

In some rare scenarios, to write optimal code, one may choose to read, write, append on files. The idea is then, using *open*() function. The syntax is :

```
1  fp = open( path [, mode_string ] )
```

The mode strings are defined as such: There are 3 modes :
1. Read : default mode : specified by "r"
2. Write : write mode : specified by "w"
3. Append : append mode : specified by "a"
4. Binary : binary mode : specified by "b"

8.3.2 *Reading*

To read, one must open the file in read mode. This returns a java BufferedReader object. Now, something like readLine can be called to read the file line by line, in a traditional way to implement the *cat* command :

```
1  // open in read mode : get a reader
2  fp = open( path ,"r" )
3  while ( null!=( line = fp.readLine() ) ){
4      // call the standard java functions
5      println(line)
6  }
7  // close the reader
8  fp.close()
```

8.3.3 Writing

To write, one must open the file in write or append mode. This returns a java PrintStream object. Now, something like println() can be called to write to the file :

```
1  // open in read mode : get a reader
2  fp = open( "hi.txt" ,"w" )
3  // call the standard java functions
4  fp.println("Hi")
5  // close the Stream
6  fp.close()
```

8.4 web io

8.4.1 open() for web

open() function is used to open a http or https connection:

```
1  // open web connection
2  wcon = open( "https://www.google.co.in" )
3  // wcon has multiple fields:
4  wcon.readTo // Read timeout
5  wcon.conTo // Connection timeout
```

8.4.2 get() , post()

Once we have *open()* a http or https connection, one can get or post into it by appropriate functions :

```
1  resp = wcon.get("/", {:} ) // path, headers to send
2  resp = wcon.post("/", {:}  ) // path, header map
3  resp = wcon.post("/", {:}, body ) // path, headers, body
```

8.4.3 Sending to url : send() function

To write back to url, or rather for sending anything to a server, there is a specialised send() function.

```
1  _url_ = 'https://httpbin.org'
2  path = "/post"
3  wcon = open(_url_)
4  params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
5  // path ,  protocol,  headers, body
```

```
6  resp = wcon.send( post  , "POST"  , {:} , jstr(params) )
```

The SOAP XML stuff can be easily done with *send*. See the example soap body we will send to :

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <GetWeather xmlns="http://www.webserviceX.NET">
            <CityName>%s</CityName>
            <CountryName>%s</CountryName>
        </GetWeather>
    </soap:Body>
</soap:Envelope>
```

And this is how we *send* the SOAP:

```
1  template_pay_load = read('samples/soap_body.xml')
2  pay_load = str( template_pay_load , 'Hyderabad', 'India' )
3  headers = { "SOAPAction" : "http://www.webserviceX.NET/GetWeather" ,
4             "Content-Type" : "text/xml; charset=utf-8" }
5  wcon = open( "http://www.webservicex.net")
6  resp = wcon.send( "/globalweather.asmx" , "POST" , headers , pay_load )
7  x = xml(resp.body() )
8  sr = x.element("//GetWeatherResult")
9  write(sr.text)
```

8.5 working with json

8.5.1 *What is JSON?*

JSON is being defined formally in here. The idea behind JSON stems from : *JavaScript Object Notation*. There are many ways to define an object, one of them is very practical :

Objects are property buckets.

That definition ensures that the ordering layout of the properties does not matter at all. Clearly :

$$\{ "x" : 10, "y" : 42 \} == \{ "y" : 42, "x" : 10 \}$$

Thus, we reach the evident conclusion, JSON are nothing but Dictionaries. Hence in ZoomBA they are always casted back to a dictionary.

8.5.2 *json() function*

Suppose the json string is there in a file:

```
1  /* sample.json file  */
2  {
3    "zoomba": {
4      "dbName": "zoomba",
5      "url": "jdbc:postgresql://localhost:5432/zoomba",
6      "driverClass" : "org.postgresql.Driver",
7      "user": "zoomba",
8      "pass": ""
9    },
10   "some2": {
```

```
11        "dbName": "dummy",
12        "url": "dummy",
13        "driverClass" : "class",
14        "user": "u",
15        "pass": "p"
16     },
17  }
```

To read the file (or the text which has json) :

```
1  jo = json('sample.json', true)
```

The *json*() function can read also from the string argument. The return result object is either a Dictionary object, or an array object, based on what sort of json structure was passed.

8.5.3 *Accessing Fields*

Now, to access any field:

```
1  jo.zoomba.dbName // "zoomba"
2  jo.some2.driverClass // "class"
```

In effect, one can imagine this is nothing but a big property bucket. For some, there is this thing called JSONPath, which is clearly not implemented in ZoomBA, because that is not a standard. In any case, given json is a string, to check whether a text exists or not is a regular string search. Given whether a particular value is in the hierarchy of the things or not, that is where the stuff gets interesting.

For parameterised access, Currying is a better choice. Better still is *xpath(), jxpath()* formulation:

```
1  prop_value = 'zoomba.dbName'
2  value = #'jo.#{prop_value}' // same thing, "zoomba"
3  xpath(jo, '/zoomba/dbName') // same
```

In this way, one can remove the hard coding of accessing JSON objects.

8.5.4 *Yaml Processing*

ZoomBA supports yaml processing by the same way:

```
1  jo = yaml(string) // loads the yaml string into json object
2  jo = yaml('file.yaml', true ) // loads the yaml file into json object
```

Only important distinction is, for Yaml processing, the parser does not support automatic type conversion into primitive type, thus, type hints are needed.

8.6 working with xml

Please, do not work with xml. There are many reasons why xml is a terrible idea. The best of course is :

> *XML combines the efficiency of text files with the readability of binary files – unknown*

But thanks to many big enterprise companies - it became a norm to abused human intellect - the last are obviously Java EE usage in Hibernate, Springs and Struts. Notwithstanding the complexity and loss of precise network bandwidth - it is a very popular format. Thus - against my better judgment we could not avoid XML.

8.6.1 *xml() function*

Suppose, we have an xml in a file sample.xml :

```
<slideshow
title="Sample Slide Show"
date="Date of publication"
author="Yours Truly" >
<!-- TITLE SLIDE -->
<slide type="all">
  <title>Wake up to WonderWidgets!</title>
</slide>
<!-- OVERVIEW -->
<slide type="all">
    <title>Overview</title>
    <item>Why <em>WonderWidgets</em> are great</item>
    <item/>
    <item>Who <em>buys</em> WonderWidgets</item>
</slide>
</slideshow>
```

Call *xml*() To load the xml file (or a string) into an ZXml object :

```
1  x = xml('sample.xml', true , 'UTF-8' ) // x is a ZXml object
```

The encoding string (UTF-8) is optional. Default is UTF-8. The full syntax of *xml()* is :

```
1  // x is a ZXml object
2  x = xml( source [, is_source_file? false , encoding=UTF-8, validate? false )
```

8.6.2 *Converting to Other Formats*

Given an Xml object, there are handy ways to convert them into other formats.

```
1  // x is a ZXml object
2  x = xml( ... )
3  json_object = json(x) // converts xml to json object
4  json_string = jstr(x) // converts xml to json string
5  yaml_string = ystr(x) // converts xml to yaml string
```

8.6.3 *Accessing Elements*

Given we now have the ZXml object, we should be able to find elements in it. The elements can be accessed in two different ways. The first is, using the xml as a tree with *root* and *children*. That is easy, for example, for the previous json to xml generated :

```
1  // XmlMap of the x
2  y = xml(x)
3  y.root.children[0].name //glossary
4  y.root.children[0].children[0].text // example glossary
```

For the slideshow xml:

```
1  // XmlMap of the x
2  x = xml('sample.xml',true) // slideshow xml
3  x.root.name //slideshow
4  // access attributes
5  x.root.children[0].attr.type // "all"
6  x.root.children[0].attr['type'] // "all"
```

The other way of accessing elements, is to use the *element*() function, which uses XPATH :

```
1  // XmlMap of the x
2  x = xml('sample.xml', true ) // slideshow xml
3  // get one element
4  e = x.element("//slide/title")// first title element
5  e.text // "Wake up to WonderWidgets!"
6  e = x.element("//slide[2]/title" ) // second element
7  e.text // "Overview"
```

In the same way, for selecting multiple elements, *elements*() function is used :

```
1  // XmlMap of the x
2  x = xml('sample.xml') // slideshow xml
3  // get all the elements called titles
4  es = x.elements("//slide/title" ) // a list of elements
5  // print all titles ?
6  fold(es) as {  println( $.text ) }
```

8.6.4   *XPATH Formulation*

For evaluation of xpath directly, there is *xpath*() function defined. For example, to find the text of any element, one can use the following :

```
1  // XmlMap of the x
2  x = xml('sample.xml',true) // slideshow xml
3  // text of the title element
4  s = x.xpath("//slide/title/text()" )
```

For a list of all xpath functions see the specification in w3c.
To check if a condition exists or not, one can use the *exists*() function.

```
1  // XmlMap of the x
2  x = xml('sample.xml',true) // slideshow xml
3  // text of the title element, exists?
4  b = x.exists("//slide/title/text()" ) // bool : true
5  b = x.exists("//slide/title" ) // bool : true
6  b = x.exists("//foobar" ) // bool : false
```

Obviously, one can mix and match, that is, give a default to the *xpath*() function :

```
1  // string : text comes
2  t = x.xpath("//slide/title/text()" , 'NONE_EXISTS' )
3  // string : 'NONE_EXISTS'
4  t = x.xpath("//foobar/text()",  'NONE_EXISTS' )
```

8.7 GENERIC XPATH, XELEMENT

In any declarative flavoured language, it is imperative that automatic discovery of fields and dynamic accessing of fields to be supported. For XML, people do have xpath, while for JSON a non standard json path is coming recently to the foray. ZoomBA uses apache jxpath to simplify the whole problem.

JXPath can be used for any complex objects, including Maps and Collections. The syntax is trivially same like xpath.

8.7.1 *xpath()*

To access value of an element we use the function *xpath*() :

```
v = xpath( object, xpath , [all_items=false] )
// this is how it works out
d = {'x' : 42 }
v = xpath(d,'/x', false ) // 42 a value
v = xpath(d,'/x', true ) // [ 42 ] List
```

There is no way to pass a default, when in doubt, use all switch to true, to ensure that when a field is not there, it returns an empty list.

8.7.2 *xelem()*

To access an element using an xpath we use the function *xelem*() :

```
v = xelem( object, xpath , [all_items=false] )
// this is how it works out
d = {'x' : [ 0,1,2] }
e = xelem(d,'/x', false ) // /.[@name='x'] // DynamicPropertyPointer
e.value // [0,1,2]
```

See more about DynamicPropertyPointer.

As usual, pass true to generate a list.

8.8 DATAMATRIX

A DataMatrix is an abstraction for a table with tuples of data as rows. Formally, then a data matrix $M = (C, R)$ where C a tuple of columns, and R a list of rows, such that $\forall r_i \in R$, r_i is a tuple of size $|r_i| = |C|$. In specificity, $r_i[C_j] \rightarrow r_{ij}$, that is, the i'th rows j'th cell contains value which is under column C_j.

In software such abstraction is natural for taking data out of data base tables, or from spreadsheets, or comma or tab separated values file.

8.8.1 *matrix() function*

To read a data matrix from a location one needs to use the *matrix*() function :

```
m = matrix(data_file_path  [, column_separator_string = '\t'
                [, header_is_there_boolean = true ]] )
```

For example, suppose we have a tab delimited file "test.tsv" like this:

Number	First Name	Last Name	Points	Extra
1	Eve	Jackson	94	
2	John	Doe	80	x
3	Adam	Johnson	67	
4	Jill	Smith	50	xx

To load this table, we should :

```
1  m = matrix("test.tsv", '\t', true ) // loads the matrix
2  m.columns // the set of columns
3  m.rows // the list of data values
```

The field *columns* is a Set of string values, depicting the columns. *rows*, however are the list of data values.

8.8.2 *Accessing Data*

A data matrix can be accessed by rows or by columns. For example :

```
1  x = m.rows[0][1] // x is "Eve"
2  x = m.rows[1][2] // x is "Doe"
```

Now, using column function $c()$

```
1  y = (m.c(1))[1] // y is "Eve"
2  y = (m.c(2))[1] // y is "Doe"
```

8.8.3 *Tuple Formulation*

Every row in a data matrix is actually a tuple. This tuple can be accessed using the *tuple()* function which returns a specific data structure called a Tuple:

```
1  t = m.tuple(1) // 2nd row as a Tuple
2  t.0 // 2
3  t["First Name"] // John
4  t.Points // 80
```

This tuple object is the one which gets returned as implicit row, when select operations on matrices are performed.

8.8.4 *Project and Select*

The project operation is defined here. In essence it can be used to slice the matrix using the columns function $c()$. The column function $c()$ takes an integer, the column index of slicing, and can take aggregated rows, which are a list of objects, which individually can be :

1. An integer, the row index
2. A range: $[a : b]$, the row indices are between a to b.

Thus :

```
1  m.c(0,1) // selects first column, and a cell of 3rd row
2  m.c(0, [0:2]) // selects first column, and two cells : 1st and 2nd row
```

But that is not all we want. We want to run SQL like select query, where we can specify the columns to pass and appear. That is done using *select*() function. It takes the anonymous parameter as an argument. The other normal arguments are objects which individually can be :

1. An integer, the column index to project
2. A String, the column name to project
3. A range: $[a : b]$, the column indices to project, between a to b.

Thus :

```
1  x = m.select(0,2)
2  // x  := [[1, Jackson], [2, Doe], [3, Johnson], [4, Smith]]
```

Same thing can be accomplished by :

```
1  x = m.select('Number' , 'Last Name' )
2  // x  := [[1, Jackson], [2, Doe], [3, Johnson], [4, Smith]]
```

Changing the order of the columns results in a different Tuple, as it should :

```
1  x = m.select(2 , 0 )
2  // x :=  [[Jackson, 1], [Doe, 2], [Johnson, 3], [Smith, 4]]
```

As always, *select*() works with condition, so :

```
1  x = m.select(2 , 0 , where {  $.Number > 2  } )
2  // x :=  [[Johnson, 3], [Smith, 4]]
```

which demonstrates the conditional aspect of the *select*() function.

The tuples generated can be transformed, while being selected. For example, we can change the Last Name to be lowercased :

```
1  x = m.select(2 , 0 , where {
2          ( $.Number > 2 ){
3              $[2] = $[2].toLowerCase()
4          }
5      } ) // x :=  [[johnson, 3], [smith, 4]]
```

8.8.5 *The matrix() function*

One can create a matrix out of an existing one. Observe that, using project and select generates only the list of data values, not a new matrix itself. If one wants to create a matrix out of an existing one :

```
1  m2 = m.matrix(2 , 0 ,  where ( $.Number > 2 ){
2              $[2] = $[2].toLowerCase()
3          }
4      )
```

This generates a new matrix named *m2*, with columns starting from "Last Name" and "Number". The rows values are exactly the same as the select query done.

8.8.6 Keys

Given we have 2 matrices $(m1, m2)$, we can try to figure out if they differ or not. A Classic case is comparing two database tables from two different databases. A very naive code can be written as such :

```
1  def match_all_rows(m1,m2){
2      count = 0
3      for ( r1 : m1.rows ){
4        // linearise the left tuple
5        st1 = str(r1,'#')
6        for ( r2 : m2.rows ){
7            // linearise the right tuple
8            st2 = str(r2,'#')
9            // they match
10           if ( r1 == r2 ){ count += 1 }
11       }
12     }
13     ( count == size(r1.rows) && count == size(r2.rows) )
14 }
```

The complexity of this algorithm, with respect to the size of the rows of the matrices is : $\Theta(n^2)$, given n is the rows size of the matrices. Given both the matrices have m columns, the complexity would become $\Theta(n^2 m^2)$, a huge amount considering a typical $(n, m) := (10000, 10)$.
We can obviously improve this setting:

```
1  def match_all_rows(m1,m2){
2      // exactly m * n
3      l1 = list(m1.rows ) as { str($.o,'#') }
4      // exactly m * n
5      l2 = list(m2.rows ) as { str($.o,'#') }
6      // exactly n
7      diff = l1 ^ l2
8      empty(diff) // return
9  }
```

The new complexity comes to $\Theta(nm)$. The thing we are implicitly using is known as key. We are not really using the key, but, stating that key uniquely represents a row. The criterion for a set of attributes to be qualified as key can be described as :

```
1  // a set of attribute indices
2  key_attrs = [ index_1 , index_2 , ...  ]
3  def is_key( m, key_attrs){
4      s = set(m.rows) as {
5            r = $.o // store the row
6            // generate a list by projecting the attributes
7            l = fold(key_attrs) as { r[$.o] }
8            // linearise the tuple
9            str(l,'#')
10        }
11     size(s) == size(m.rows)
12 }
```

What happens when the keys are non unique? That boils down to the list, but then we should be able to remember, which rows have the same keys, or rather what all rows are marked by the same key. So, we may have a keys function, assuming keys are not unique :

```
1  // a set of attribute indices
2  key_attrs = [ index_1 , index_2 , ...   ]
3  def key( m, key_attrs, sep){
4      keys = fold (m.rows, dict() ) as {
5            r = $.o // store the row
6            // generate a list by projecting the attributes
7            l = fold(key_attrs) as { r[$.o] }
8            // linearise the tuple
9            k = str(l, sep)
10           if ( k @ $.p ){ // the key already exists
11               $.p[k] += $.i // append the current row index to the rows
12            }else{
13               // key does not exist, create one and then append
14               $.p[k] = list($.i) // key is the row id
15            }
16           $.p
17       }
18  }
```

This idea is already in-built in the DataMatrix, and is known as the *keys*() function :

```
1  m.keys{  /* how to generate one */ }( args... )
2  m.keys( columns_for_keys )
```

So, to simply put, *keys*() generate a key dict for the matrix, as shown below, using the matrix in (8.8.1) :

```
1  // the key is the column Number
2  m.keys( 'Number' )
3  // see what the keys are ?
4  m.keys /* {4?=[3], 3?=[2], 2?=[1], 1?=[0]}  */
```

We can generate a non unique key too, just to show the practical application :

```
1  // the key is the column Number modulo 2: [0,1]
2  m.keys( as {  $.Number % 2  } )
3  // see what the keys are ?
4  m.keys /* {0=[1, 3], 1=[0, 2]}  */
```

8.8.7 *Aggregate*

Once we establish that the keys function did not generate unique keys, one may want to consolidate the rows to ensure that the key points to unique rows. This is done by the *aggregate*() function. This function can be surmised to generate a new matrix from the old matrix, where the same keys pointing to different rows must be aggregated based on columns.

```
1  def aggregate(m,columns , aggregate_function ){
2    agg_rows = list()
3    for ( key : m.keys ){
4        row = list()
5        rows_list = m.keys[key]
6        for ( c : columns ){
7            // aggregate the rows on a single column
8            col_data = m.c( c, rows_list )
9            agg_data = aggregate_function( col_data )
10           row += agg_data // add a replacement for all the rows
11       }
12       // add to the newer row
13       agg_rows += row
14   }
15   // now we have the new matrix data rows
16 }
```

Interacting with Environment

Environment is needed to run any system. In this chapter we discuss about that environment, which is aptly called the Operating System. We would discuss about how to make calls to the operating system, how to create threads, and how to operate in linear ways in case of error happened, and how to have random shuffling generated.

9.1 PROCESS AND THREADS

9.1.1 *system() function*

To run any operating system command from ZoomBA, use the *system*() function. It comes in two varieties, one where the whole command is given as single string, and in another the commands are separated as words :

```
1  status = system(command_to_os)
2  status = system(word1, word2,...)
```

The return status is the exit status of the command. It returns 0 for success and non zero for failure. For example, here is the sample :

```
1  status = system("whoami") // system prints "noga"
2  /* status code is 0 */
```

In case of unknown command, it would throw an error:

```
1  status = system("unknown command")
2  /* error will be thrown */
```

In case of command is found, but command execution reported an error, it status would be non zero:

```
1  status = system("ls -l thisDoesNotExist")
2  /* os reprots
3  ls: thisDoesNotExist: No such file or directory
4  status := 1
```

```
5  */
```

In the same way, multiple words can be put into system, which is of some usefulness when one needs to pass quoted arguments :

```
1  // the result of "ls -al ." is not shown at all, use popen()
2  status = system("ls" , "-al" , "." )
3  /* status code is 0 */
```

9.1.2 *popen() function*

If you want to work with processes, use *popen()* function.

```
1  p = popen("ls" , "-al" , "." )
2  p.waitOn(10000) // timeout
3  p.status().out // stdout
4  p.status().err // stderr
5  p.status().code // exit code
```

popen() comes with multiple command line arguments, as in :

```
1  p = popen(args=["ls" , "-al", "/" ],
2      redirect={ "out" : "out_file" , "err" : "err_file" , "in" : "in_file"  }
         )
```

This way, one can redirect input, output and error. A call to *popen()* w/o any input yields a dummy abstraction of the current running process, which then can be used to extract the *pid* by :

```
1  cur = popen( )  // dummy current ZProcess
2  cur.process.pid // current process id
```

9.1.3 *thread() function*

thread() function, creates a thread. It also runs it immediately. The syntax is simple :

```
1  t = thread( args_to_be_passed ) as {  /* thread function body  */  }
2  current_thread = thread() // gets current thread
```

Thus, simply, use *thread*() to create a thread:

```
1  t = thread( "Hello", "world" ) as {
2     printf('My Id is %d\n', $.i )
3     printf('My args are: %s\n', str($.c,'\n' ) )
4     printf('I am %s\n', $.o )
5   }
6  /* java isAlive() maps to alive */
7  while ( t.alive ){
8     cur_thread = thread() // returns current thread
9     cur_thread.sleep(1000)
10 }
```

This would produce some kind of output as below:

```
My Id is 10
My args are Hello
world
I am Thread[Thread-1,5,main]
```

It is possible to send named arguments in the thread block as follows:

```
1  t = thread( first = "Hello", second = "world" ) as {
2     printf('My args are: first %s second %s %n',  first, second )
3     }
```

This way it is quite easy to handle the parameters passed into a thread.

9.1.4 *fiber() function*

Java 21 have green threads, or virtual threads. If we are running in Jva 21 JRE or beyond, *fiber()* function can be used to spawn a virtual/green thread. Syntax is exactly as same as that of thread function.

```
1  t = fiber( first = "Hello", second = "world" ) as {
2     printf('My args are: first %s second %s %n',  first, second )
3     }
```

9.1.5 *task() function*

In case one just chose to create a thread, but does not run it, one should use *task()* function. Syntax is same as thread, but it does not run it immediately.

```
1  t = task( args_to_be_passed ) as {  /* thread function body  */  }
```

9.1.6 *batch() function*

In case people would be wondering where the *task()* function would be used, it can be used in a threadpool scheduler.

Such a scheduler is exposed as:

```
1  l = list([0:10]) as { task($.o) as{ printf("%d ", $.c.0); $.c.0 + 1 } }
2  lr = batch( l )
3  s = sum( lr ) as { $.o.value }
```

batch() function has two optional parameters:

```
1  lr = batch( list_of_functions [, pool_size , time_out] )
```

9.1.7 *concur() function*

Once one gets the idea of thread and parallel tasks, it is not hard to imagine that the basal collection processing can actually be done via the same using some form of *map reduce*.

This gets done by the *concur()* function:

```
1   x = from( [0:100] ) where { $.o % 2 == 1  } as  { $.o ** 2 }
2   y = concur( [0:100] , set() ) where { $.o % 2 == 1 } as { $.o ** 2 }
3   assert( x == y, "Concurrent is different than sequential" )
```

concur() has the exact same syntax as *from*() function, just that it spawns parallel threads to process the data. For small data range less than 1 million, there would be nearly no cost saving.

9.1.8   *poll() function*

Observe the last example where we used a while to wait for a condition to be true. That would generate an infinite loop if the condition was not true. So, many times we need to write this custom waiter, where the idea is to wait till a condition is satisfied. Of course we need to have a timeout, and of course we need a polling interval. To make this work easy, we have *poll*().

The syntax is :

```
1   poll ( [ timeout-in-ms = 3000, [ polling-interval-in-ms = 100] ]) [ where {
        condition-body-of-function } ]
```

So these are explanations :

```
1   poll() // this is ok : simple default wait
2   poll (4000) // ok too !
3   poll (4000, 300 ) // not cool : 300 is rejected , no condition!
```

A very mundane example would be :

```
1   i = 3
2   poll( 1000, 200 ) where { i-= 1 ; i == 0 }// induce a bit of wait
3   // this returns true : the operation did not timeout
4   i = 100
5   poll( 1000, 200 ) where { i+= 1 ; i == 0 } // induce a bit of wait
6   // this returns false : timeout happened
```

and, finally, we replace the while statement of the thread example using *poll*() :

```
1   t = thread() as {
2      /* do anything one needs */
3      }
4   /* java isAlive() maps to alive */
5   poll( 10000, 300 ) where { ! t.alive }
```

9.1.9   *Atomic Operations*

Given there are threads, being atomic is of importance. The definition of being atomic is :

Either the block gets totally executed, or it does not at all.

Observe the problem here :

```
1   $count = 0
2   // create a list of threads to do something?
3   l = list( [0:10] ) as { thread() as { $count+= 1 } } // increase count...
```

```
4  poll ( 10000, 50 ) where {
5    running = select(l) where { $.o.isAlive } ; empty(running)  }
6  println($count) // what is the value here?
```

You would expect it to be 10, but it would be any number $x \leq 10$, because of the threading and non-atomic issue. To resolve it, put it inside an atomic block (#atomic{}) :

```
1  $count = 0
2  // create a list of threads to do something?
3  l = list( [0:10] ) as { thread() as { #atomic{ $count+= 1 } } }
4  // above tries to increase count...
5  poll( 10000, 50 ) where {
6    running = select(l) as { $.isAlive } ; empty(running)
7    }
8  println($count) // the value is 10
```

and now, the count is always 10.
Atomic can be used to revert back any changes in the variables :

```
1   x = 0
2   #atomic{
3      x = 2 + 2
4   }
5   #atomic{
6      x = 10
7      println(x) // prints 10
8   /* error, no variable as "p', x should revert back to 4 */
9      t = p
10  }
11  println(x) // x is 4
```

9.1.10   *The clock Block*

Timing is of essence. If one does not grab hold of time, that is the biggest failure one can be. So, to measure how much time a block of code is taking, there is *clock*() plugin function. The syntax is simple :

```
1  // times the body
2  #(time_taken_in_sec_double, result ) = #clock{ /* body  */}
```

Thus this is how you tune a long running code:

```
1  m1 = 0
2  #(t,r) = #clock{
3     // load a large matrix, 0.2 million rows
4     m1 = matrix('test.csv', ',' , false)
5     0 // return 0
6  }
7  println(t) // writes the timing in seconds ( double )
8  println(r) // writes the result : 0
```

The clock block never throws an error. That is, if any error gets raised in the block inside, the error gets assigned to the result variable.

```
1  #(t,r) = #clock{  x = foobar_not_defined_var }
2  // r would be ZoomBA variable exception
```

Note that the timing would always be there, no matter what. Even in the case of an error, the timing variable would be populated.

9.2 HANDLING OF ERRORS

We toyed with the idea of try-catch, and then found that : Why Exception Handling is Bad? ; GO does not have exception handling. And we came to believe it is OK not to have exceptions in a language that has no business becoming a system design language. In a glue language like ZoomBA, we should code for everything, and never ever eat up error like situations. For this specific need, Asserts are in place. In particular - the multi part arguments and the return story - should ensure that there is really no exceptional thing that happens. After all, trying to build a fool-proof solution is never full-proof. And one of the return values should then be asserted.

9.2.1 *error() function*

But we need errors to be raised. This is done by the *error*() function.

```
1  error( args... ) where [ { /* body, when evaluates to true raise error */} ]
```

The function *error*() can have multiple modes, for example this is how one raise an error with a message :

```
1  error( "error message")
```

This is how one raise an error on condition :

```
1  error( condition , "error message")
```

when *condition* would be true, error would raised. If the condition is false, *error*() returns *false*. In case we want to execute complex logic, and guards it against failure:

```
1  error("error message") where { /* error condition */ }
```

9.2.2 *Multiple Assignment*

GoLang supports multiple assignments. This feature is picked from GoLang. Observe the following :

```
1  #(a,b) = [ 0 , 1 , 2 ] // a = 0, b = 1
2  #(,a,b) = [ 0 , 1 , 2 ] // a = 1, b = 2
```

The idea is that one can assign a collection directly mapped to a tuple with proper variable names, either from left or from right.

Note the quirky "," before *a* in the 2nd assignment, it tells you that the splicing of the array should happen from right, not from left.

9.2.3 *Error Assignment*

A variant of multiple assignment is error assignment. The idea is to make the code look as much linear as possible. The issue with the *try*() function is that, there is no way to know that error was raised, actually. It lets one go ahead with the default value. To solve this issue, there is multiple return, on error.

```
1  import 'java.lang.Integer' as Int
2  #(n ? e)  =  Int.parseInt('42') //n := 42 int, e := null
3  // error will be raised and caught
4  #(n ? e) = Int.parseInt('xx42') //n := null, e:= NumberFormatException
```

Observe the "?" before the last argument *e*, to tag it as the error container. Thus, the system knows that the error needs to be filled into that specific variable. Thus, one can write almost linear code like this:

```
1  import 'java.lang.Integer' as Int
2  #(n ? e)  =  Int.parseInt(str_val)
3  empty(e) || bye('failure in parsing ', str_val )
```

The *bye*() function, when called, returns from the current calling function or script, with the same array of argument it was passed with.

9.3   ORDER AND RANDOMNESS

9.3.1 *Lexical matching : tokens()*

In some scenarios, one needs to read from a stream of characters (String) and then do something about it. One such typical problem is to take CSV data, and process it. Suppose one needs to parse CSV into a list of integers. e.g.

```
1  s = '11,12,31,34,78,90' // CSV integers
2  l = select( s.split(',') )  where { !empty( int($.o) )  } as { int($.o) }
3  // l := [11, 12, 31, 34, 78, 90] // above works
```

But then, the issue here is : the code iterates over the string once to generate the split array, and then again over that array to generate the list of integers, selecting only when it is applicable. Clearly then, a better approach would be to do it in a single pass, so :

```
1  s = '11,12,31,34,78,90'
2  tmp = ''
3  l = list()
4  for ( c : s ){
5      continue ( c == ',' ){
6          l += int(tmp)
7          tmp = ''
8      }
9      tmp += c
10  }
11  l += int(tmp)
12  println(l)
```

However, this is a problem, because we are using too much coding. Real developers abhor extra coding. The idea is to generate a state machine model based on lexer, in which case, the best idea is to use the *tokens*() function :

```
1  tokens( <string> , <regex> ,
2  match-case = [true|false] )
3  // returns a matcher object
4  tokens ( <string> , <regex> , match-case = [true|false] ) as { anon-block }
5  // calls the anon-block for every  matching group
```

Thus, using this, we can have:

```
1  // what to do : string : regex
2  l = tokens(s, '\d+') as { int($.o) }
```

and we are done. That is precisely how code is meant to be written.

9.3.2   *hash() function*

Sometimes it is important to generate hash from a string. To do so :

```
1  h = hash('abc')
2  // h  := 900150983cd24fb0d6963f7d28e17f72
```

It defaults to "MD5", so :

```
1  hash( 'MD5' ,'abc')
2  // h := 900150983cd24fb0d6963f7d28e17f72
```

They are the same. One can obviously change the algorithm used :

```
1  hash([algo-string , ] <string> )
```

There are these two specific algorithms that one can use to convert to and from base 64 encoding. They are "e64" to encode and "d64" to decode. Thus, to encode a string in the base 64 :

```
1  h = hash('e64', 'hello, world')
2  //h := aGVsbG8sIHdvcmxk
3  //And to decode it back :
4  s = hash('d64', 'aGVsbG8sIHdvcmxk' )
5  // s:= "hello, world"
6  // url encode and decode
7  s = hash('eu', string)
8  hash('du', s) // decode
```

9.3.3   *Uniq*

An interesting function is *uniq()* which is essentially same as the Unix standard uniq command. It returns an iterator that groups all *similar* items together:

```
1  l = [1,2,2,2,4,1,1]
2  lu = list ( uniq(l) ) // [[1], [2,2,2],[4],[1,1]]
```

As we can see it did not group across the list, it only groups in the stream of items that followed.

A very nice use case is of course the run length encoding of a list by compressing the list as a tuple of *(freq,item)* as follows:

```
1  l = [1,2,2,2,4,1,1]
2  lr = list ( uniq(l) ) as {  [ #|$.o|, $.o.0  ] } // [[1,1], [3,2],[1,4],[2,1]]
```

We can of course do better in terms of run length encoding:

```
1  l = [1,2,2,2,4,1,1]
2  lr = list ( uniq(l) ) as { ( #|$.o| == 1) ?  $.o.0  :   [ #|$.o|, $.o.0  ] }
3  // [1, [3,2],4,[2,1]]
```

It is important to note that *uniq()* also takes a *hash* method, when hashes match items are considered to be same:

```
1  l = [1,2,2,2,4,1,1]
2  lr = list ( uniq(l)  as { 2 /? $.o } )   // [[1], [2,2,2,4], [1,1]]
```

9.3.4 *Anonymous Comparators*

Ordering requires to compare two elements. The standard style of Java comparator is to code something like this :

```
1  def compare( a, b ) {
2      if ( a < b ) return -1
3      if ( a > b ) return  1
4      return 0
5  }
6  // or in short, with sign function :
7  def compare(a,b) {  sign(a - b) }
```

Obviously, one needs to define the relation operator accordingly. For example, suppose we have student objects :

```
1  student1 = { 'name' : 'Anakin' , 'id' : 42  }
2  student2 = { 'name' : 'Mace' , 'id' : 24  }
```

And we need to compare these two. We can choose a particular field, say "id" to compare these :

```
1  def compare( a, b ) {
2      if ( a.id < b.id ) return -1
3      if ( a.id > b.id ) return  1
4      return 0
5  }
```

Or, there is another way to represent the same thing :

```
1  def compare( a, b ) {
2      ( a.id < b.id )
3  }
```

In ZoomBA compare functions can be of both the types. Either one choose to generate a triplet of $(-1, 0, 1)$, or one can simply return *true* when $left < right$ and false otherwise. The triplet is not precisely defined, that is, when the comparator returns an integer :

1. The $result < 0 \implies left < right$
2. The $result = 0 \implies left = right$
3. The $result > 0 \implies left > right$

when it returns a boolean however, the code is :

```
1  cmp = comparator(a,b)
2  if ( cmp ) {
3    println( 'a<b' )
4  }else{
5    println( 'a >= b' )
6  }
```

In the subsequent section anonymous comparators would be used heavily. The basic syntax of these functions are :

```
1  order_function(args... ) where { /* anonymous comparator block */ }
2  order_function(args... ) as { /* anonymous scalar comparator block */ }
```

As always $.0 is the left argument (a) and $.1 is the right argument (b). Thus, one can define the suitable comparator function to be used for comparison.

Note that *where* comparators can return either *bool* or $-1, 0, 1$ values. These methods take 2 parameters. Now *as* or mapper comparators are field or scalar comparators, and they take a single entry as input.

Specifically there are bunch of *predefined* comparators :

```
1  func(...) where cmp_njJ  // null first, then smallest to largest Java Style
2  func(...) where cmp_nJj  // null first, then largest to smallest Java Style
3  func(...) where cmp_jJn  // null last, else smallest to largest Java Style
4  func(...) where cmp_Jjn  // null last, else largest to smallest Java Style
```

The trouble with these *Java Style Cmp* functions are they can not handle mixing of types. Specifically numbers. A mixed numeric list will raise error. Hence *ZoomBA* style comparators exists:

```
1  func(...) where cmp_nzZ  // null first, then smallest to largest Zmb Style
2  func(...) where cmp_nZz  // null first, then largest to smallest Zmb Style
3  func(...) where cmp_zZn  // null last, else smallest to largest Zmb Style
4  func(...) where cmp_Zzn  // null last, else largest to smallest Zmb Style
```

9.3.5 *Sort Functions*

Sorting is trivial:

```
1  cards = ['B','C','A','D' ]
2  sa = sorta(cards) // ascending
3  // sa := [A, B, C, D]
4  sd = sortd(cards) //descending
5  // sd := [D, C, B, A]
```

Now, sorting is anonymous block (function) ready, hence we can sort on specific attributes. Suppose we want to sort a list of complex objects, like a Student with Name and Ids. And now we are to sort based on "name" attribute:

```
1  students = [ {'roll' : 1, 'name' : 'X' } ,
2               {'roll' : 3, 'name' : 'C' } ,
3               {'roll' : 2, 'name' : 'Z' } ]
4  sa = sorta(students) where { $[0].name < $[1].name }
5  /* sa := [{roll=3, name=C}, {roll=1, name=X},
6            {roll=2, name=Z}] */
```

Obviously we can do it using the roll too:

```
1  sa = sorta(students) where { $[0].roll < $[1].roll }
2  // sa := [{roll=1, name=X}, {roll=2, name=Z}, {roll=3, name=C}]
```

Note that with *as* formulation we are supposed to find a field or a scalar value which would be used to sort the collection:

```
1  sa = sorta(students) as { $.name  }
2  // [{roll=3, name=C}, {roll=1, name=X},  {roll=2, name=Z}]
3  sa = sorta(students) as { $.roll  }
4  // [{roll=1, name=X}, {roll=2, name=Z}, {roll=3, name=C}]
```

This is equivalent of Pythons column key for sorting.

What about we want to consider multiple fields? Obviously, we can pass our own separate function for it:

```
1  def my_cmp_func(x,y){
2    if ( x.name < y.name ) return -1
3    if ( x.name > y.name ) return  1
4    if ( x.roll < y.roll ) return -1
5    if ( x.roll > y.roll ) return  1
6    return 0
7  }
8  sa = sorta(students) where my_cmp_func
```

or, we could have written the same using *where*. Same is, in fact highly possible with the *as* function too:

```
1  sa = sorta(students) as {  str('%s-%s', $.name, $.roll)  }
```

The trick is to think of a scalar that can solve our comparison problem.

9.3.6 *Heap*

A heap is an entirely different matter. Heap stores the 'k' largest or smallest entries in a collection that is given to it. This structure let's you find top 'k' or bottom 'k' as you wish. *heap()* function in ZoomBA creates the Heap.

```
1  h = heap(3) // It is a min-heap, 3 elements
2  h ++= [0:10].asList  // add stuff up : h<0|[ 0,1,2 ]> // ZHeap
3  h.top // 0 // Integer
```

In case we want to create a max-heap :

```
1  h = heap(3,true) // It is a max-heap, 3 elements
2  h ++= [0:10].asList  // add stuff up : h<9|[ 7,8,9 ]> // ZHeap
3  h.top // 9 // Integer
```

One can, of course index into a heap :

```
1  h[0] // 7
2  h[-1] // 9 : last element
```

And finally, one can always use a comparator to create the Heap, because that comparison is needed to compare elements:

```
1  l = [  [1,2] , [4,1] , [2,10 ], [0,40] , [-1,-3] ]
2  h = heap(2) as {  $.o.1  } // right element
3  h ++= l // consume all of it
4  h.top // @[-1,-3] is the top element
```

9.3.7 *Priority Queue*

While Heap is a neat structure for top-k, it is a specialisation of Priority Queue. In heap there are only some 'k' items while in Priority Queue, there can be as many. ZoomBA implementation wraps around Java Priority Queue.

```
1  pq = pqueue() [ where { /* comparator function */ } ]
```

For example :

```
1  l = [10,1,23,12,15,0,3]
2  pq = pqueue()
3  for ( l ) { pq += $ } // consume
4  pq.peek // 0
5  while ( !empty(pq) ){ println(pq.poll) } // 0 1 3 10 12 15 23
```

9.3.8 *sum() function*

Sometimes it is of importance to generate *sum* of a collection. That is precisely what *sum()* achieves :

```
1  l = [0,-1,-3,3,4,10 ]
2  s = sum(l)
3  /*  sum := 9  */
```

Obviously, the function takes anonymous function as argument, thus, given we have :

```
1  x = [ 2,3,1,0,1,4,9,13]
2  s = sum(x) as { $.o * $.o }
3  /* 281 */
```

Thus, it would be very easy to define on what we need to sum over or min or max. Essentially the anonymous function must define a scalar to transfer the object into.

9.3.9 *minmax() function*

It is sometimes important to find min and max of items which can not be cast into numbers directly. For example one may need to find if an item is within some range or not, and then finding min and max becomes important. Thus, we can have :

```
1  x = [ 2,3,1,0,1,4,9,13]
2  #(m,M) = minmax(x)
3  // [0, 13]
```

This also supports anonymous functions,thus :

```
1  students = [ {'roll' : 1, 'name' : 'X' } ,
2               {'roll' : 3, 'name' : 'C' } ,
3               {'roll' : 2, 'name' : 'Z' } ]
4  #(m,M) =  minmax(students) where { $[0].name < $[1].name }
5  // [{roll=3, name=C}, {roll=2, name=Z}]
```

9.3.10 *shuffle() function*

In general, testing requires shuffling of data values. Thus, the function comes handy:

```
1  cards = [ 'A', 'B' , 'C' , 'D' ]
2  // inline shuffling
3  happened = shuffle(cards)
4  // happened := true , returns true/false stating if shuffled
5  cards
6  // order changed : [D, A, C, B]
```

9.3.11 *The random() function*

The *random*() function is multi-utility function. With no arguments it returns a SecureRandom :

```
1  sr = random() // ZRandom instance, a SecureRandom implementation
```

random() function can be used in selecting a single value at random from a collection :

```
1  l = [0,1,2,3,4]
2  rs = random(l) // rs is a randomly selected element
```

It can also be used to select a random sub collection from a collection, we just need to pass the number of elements we want from collection :

```
1  a = [0,1,2,3,4]
2  rs = random(a,3) // pick 3 items at random without replacement
3  // works on strings too....just to array
4  rs = random("abcdefghijklmpon".toCharArray ,10)
```

Given a seed data type, it generates the next random value from the same data type. Behold the behaviour:

```
1   rand_bool_value = random(true) // generates random boolean
2   random_within_0_9 = random(10) // a random integer between 0 to 9
```

Random instance supports multitudes of other randomized functions like :

```
1    r = random() // get the random instance
2    r.string("\\d+", 10 ) //string matching regex with min size 10
3    r.string("[abc]+", 10, 20 ) // string matching regex with min size 10 , max 20
4    // now various numeric ones
5    r.num() // next random integer
6    r.num(0.0) // next double
7    seed = 1
8    r.num(seed.longValue()) // gets a long random number
9    r.num( "2" ) // 2 size large integer as random
10   r.num(true) // Next Gaussian
```

9.4   ERRORS IN ZOOMBA

ZoomBA was written as a glue language. Therefore, utmost care is taken to report any error, in a precise manner. For reference, we have used this file called *t.zm* for all the examples.

9.4.1   *Unknown Variable Error*

Let's start with a very simple error :

```
1   println(x) // x is not defined
```

You would see the error comes in as :

```
zoomba.lang.core.types.ZException$Variable: x :   -->
/Codes/zoomba/image_scraper/t.zm at line 1, cols 9:9
|- println --> /Codes/zoomba/image_scraper/t.zm at line 1, cols 1:10
```

This is the most common type of error one is going to get.

9.4.2   *Unknown Property Error*

In a dynamic language, this is the next error one gets:

```
1   obj = { 'a' : 10 }
2   println( obj.x ) // but there is no property called 'x' !!!
```

You would see the error comes in as :

```
zoomba.lang.core.types.ZException$Property:
 Object {a=10} does not have property 'x' of class java.lang.String :
 --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 14:14
|- println --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 1:16
```

Note that the error also tries to showcase what all properties the object has.

9.4.3   Function Errors

This happens, when you are calling a function that does not exist:

```
1  val = 10
2  func( val ) // where is this function?
```

The error says that there is no such function:

```
zoomba.lang.core.types.ZException$Function: func :
--> /Codes/zoomba/image_scraper/t.zm at line 2, cols 1:11
|- func --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 1:11
```

It is simply stating that the function does not exist. Now, there can be issue with parameter passing, as in below:

```
1  def func( x ){
2     println(x)
3  }
4  func() // parameter is not passed!
```

Related error is:

```
zoomba.lang.core.types.ZException$Function:
parameter 'x' is not assigned!: in method : func :
 --> /Codes/zoomba/image_scraper/t.zm at line 4, cols 1:6
|- func --> /Codes/zoomba/image_scraper/t.zm at line 4, cols 1:6
```

Wait a bit here. It is OK to have a function where we use implicit args @*ARGS*, but not ok to have a parameter and then not using it.

Another related errr is method mismatch, which happens because ZoomBA is a dynamic language:

```
1  s ="foobar"
2  s.foo()
```

Clearly, no such method *foo()* exists in the String class and hence, the error comes to be:

```
zoomba.lang.core.types.ZException$Function: foo : no such method! :
foo : in class : class java.lang.String :
 --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 3:7
|- foo --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 3:7
```

What happens when a method exists, but the parameters do not match?

```
1  s ="foobar"
2  s.toString("boohahah")
```

Clearly, no such method *toString(String)* exists in the String class and hence, the error comes to be:

```
zoomba.lang.core.types.ZException$Function: toString : args did not match any known method! :
toString : in class : class java.lang.String :
 --> /Codes/zoomba/image_scraper/t.zm at li
ne 2, cols 3:20
|- toString --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 3:20
```

9.4.4   Stack Trace

While we are at functions, functions can and do call other functions.

```
1  def funky(z){
2     println(y)
3  }
4
5  def func( x ){
6     funky( x )
7  }
8  func(42)
```

In this case, ZoomBA shows the stack trace:

```
zoomba.lang.core.types.ZException$Variable: y :
--> /Codes/zoomba/image_scraper/t.zm at line 2, cols 11:11
|- println --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 3:12
|- funky --> /Codes/zoomba/image_scraper/t.zm at line 6, cols 3:12
|- func --> /Codes/zoomba/image_scraper/t.zm at line 8, cols 1:8
```

Even a for iterated loop comes under this scheme, that is because for-iterator is actually a function, calling Java's *hasNext()* and *next()* over an iterator.

9.4.5   Arithmetic Errors

Naturally, there can be arithmetic errors, when some operator we use on some object, that does not support it. Starting with the trivial :

```
1  q = 0
2  p = 42
3  println( p/q )
```

which is classic divide by zero, culminates in:

```
zoomba.lang.core.types.ZException$ArithmeticLogicOperation:
Invalid Arithmetic Logical Operation [/] :
 --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 12:12
 --> ( Can not do operation ( DIVISION ) : ( 42 ) with ( 0 ) ! )
|- println --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 1:14
```

9.4.6   LValue Error

What happens when we try to assign value to a constant term?

```
1  12 = 42
```

this culminates in :

```
zoomba.lang.core.types.ZException$Variable:
Assignment not possible into ASTNumberLiteral :
--> /Codes/zoomba/image_scraper/t.zm at line 1, cols 1:2
```

9.4.7   Stack OverFlow

Pretty common to do:

```
1  def fact( n ){
2      if ( n == 1 ) return 1
```

```
3      n * fact( n  ) // missed the : n - 1
4  }
5  fact(2)
```

this will produce :

```
zoomba.lang.core.types.ZException$Function: StackOverFlow: in method : fact :
 --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
|- fact --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
|- fact --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
|- fact --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
|- fact --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
|- fact --> /Codes/zoomba/image_scraper/t.zm at line 3, cols 9:18
.... <many lines of repeat>
```

9.4.8 *Null Error*

From Java standpoint, pretty common to reference null:

```
1  x = null
2  println( x.method() ) // imagine there is actually something
```

this will auto-panic :

```
Panic --> [  --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 12:19 ]
caused by : zoomba.lang.core.types.ZAssertionException: null reference : x
|- println --> /Codes/zoomba/image_scraper/t.zm at line 2, cols 1:21
```

Null is treated differently, hence, a Panic gets generated, because it is curable.

9.5 DEBUGGING

It is essential to have debug support in a language. ZoomBA is no exception. ZoomBA supports debugging. To do so, we must use the instrumented build variant.

Then we should alias this :

```
zdb='java --add-opens java.base/jdk.internal.loader=ALL-UNNAMED -Dzoomba.debug=4242 -jar zoomba.lang.core-INST
```

Now we are ready to debug any source code over TCP port 4242. We would talk about these two files during the debug:

main file : import_test.zm

```
1  import _/"tbi" as T
2
3  T.hello("boo")
4  println( T.__FOO__ )
5  T.__FOO__
```

imported file : tbi.zm - to be imported

```
1  import java.lang.System as SYS
2
3  __FOO__ = 42
4
5  def hi(a){
6    printf( '%s --> %s %n' , a, SYS.getProperty('user.dir') )
7  }
8
9  def hello(arg){
10   println ( "hi " + arg )
```

```
11    println ( hi )
12    hi(arg)
13  }
```

Now to run the debugger we execute:

```
zdb import_test.zm # this is what we execute, rest are server log
Debug Server Starting at : 4242
Waiting for debug client connection...
```

And we see the execution is waiting for some debug client to connect. As, the debug protocol runs with raw strings being transferred between 2 machines; our debug client will be netcat command or *nc*.

So we simply execute the following to get:

```
nc localhost 4242 # this is what we execute, rest are response from the server
bp  - sets  break point
cb  - clears  break point
ex  - executes expression in debugger
st  - shows current stack trace
lb  - list all break points
lm  - list all loaded modules
ls  - list source code
lv  - list context variables
lg  - list global variables
c   - continues in debugger
h   - shows this help

(zdb)
```

It is at this point the debugger is ready to take commands from the client.

9.5.1 *Breakpoint*

We indulge by adding a breakpoint. To add a breakpoint one must pass the module load index, as well as the line number (based on 1) where we want to put a breakpoint.

```
bp // 1:3
true
(zdb)
```

This puts the breakpoint in the file import_test.zm at line no 3. Note the curious use of // to separate out command words. That is lazy but effective way to collect commands. Now to continue running we use :

```
(zdb)
c
continue!
(zdb)
class zoomba.lang.parser.ASTMethodNode --> .../import_test.zm:3:3 to 14
```

And if we do, we reach the breakpoint immediately. Now we can inspect various things really, or choose to list the loaded modules *lm* (scripts):

```
(zdb)
lm
.../import_test.zm : 1
.../tbi.zm : 2
(zdb)
```

So it says, it loaded 2 modules at this point and listed down the load indexes in order of load. Now, say we want to put a breakpoint in the *tbi.zm* and that too in the *hello()* method. So, we are going to do just that:

```
(zdb)
bp // 2:10
true
(zdb)
```

```
c
continue!
(zdb)
class zoomba.lang.parser.ASTMethodNode --> .../tbi.zm:10:3 to 25
```

And we are broken into debugger again!

9.5.2 *Viewing Stacks*

At this point looking at stack trace is a good idea, so let's just do that:

```
(zdb)
st
|- hello --> .../import_test.zm:3:3 to 14
(zdb)
```

Ah. So indeed *hello()* got called from *import_test*. Perhaps we now want to check the variables? That is easy with *lv* :

```
(zdb)
lv
hi
@ME
hello
SYS
__FOO__
(zdb)
```

And now we just list out the breakpoints we have with *lb* :

```
(zdb)
lb
1:3
2:10
(zdb)
```

And finally, we clear the break points, all of them:

```
(zdb)
cb // all
true
(zdb)
```

And now simply continue and close the debug session!

```
(zdb)
c
continue!
(zdb)
Process Exited!
```

And that concludes our debug session!

9.6 CODE COVERAGE

We use the same instrumented build to do code coverage as follows:

```
zmc='java --add-opens java.base/jdk.internal.loader=ALL-UNNAMED -Dzoomba.intercept=true -jar zoomba.lang.core-
```

And once we are done, we can now run any script with this command:

```
zmc import_test.zm
```

This produces a coverage file :

```
ls *.txt
2024_10_21_11_22_12_326_coverage.txt
```

As you can see, the coverage has the time stamp upto ms in the name itself. Now, if you open the file:

```json
1   {
2       "blocks":{
3           ".../import_test.zm":{
4               // no blocks were found in this file
5           },
6           ".../tbi.zm":{
7               "5:10 to 7:1":true,
8               "9:15 to 13:1":true
9           }
10      },
11      "lines":{
12          ".../import_test.zm":{
13              1:true,
14              3:true,
15              4:true,
16              5:true
17          },
18          ".../tbi.zm":{
19              1:true,
20              3:true,
21              5:true,
22              6:true,
23              7:true,
24              9:true,
25              10:true,
26              11:true,
27              12:true,
28              13:true,
29              15:true
30          }
31      },
32      "methods":{
33          ".../import_test.zm":{
34
35          },
36          ".../tbi.zm":{
37              "5:1 to 7:1":true,
38              "9:1 to 13:1":true
39          }
40      }
41  }
```

Things which are covered would be marked as true. ZoomBA coverage works on lines, methods, and blocks.

OBJECT ORIENTATION

Objects are essential, programmatically. It can be argued that a map or a dictionary can work perfectly well instead of "fully blown" objects in multitude of languages.

The idea of hiding states is bad, because it assumes the incompetency of a programmer, rather all of them. A philosophy which assumes philosophers are incompetent is contradictory onto itself.

However, that is a philosophical debate, and is not needed to be played down. There is no denial of facts, that modern computing world is object oriented. Thus, ZoomBA has objects, even if we never intended it to have "language specific" objects in ZoomBA.

10.1 INTRODUCTION TO CLASSES

10.1.1 *Defining and Creating*

A class is defined by the keyword *def* , just like a method definition. The difference is, the method definition must have a () after the keyword, while the class definition does not. Thus :

```
1  def Hello{}
```

creates a class called *Hello*. Obviously one needs to create a class, that is created by the *new()* function :

```
1  // reflection style creation
2  h1 = new ( 'Hello' )
3  // hard coded creation, Hello resolves to current script
4  h2 = new ( Hello )
```

That is all it takes to create classes in ZoomBA. We sincerely thought about the Pythonic way of resolving class name as the creation of new classes, and then decided to go against it. In the pythonic way, it becomes impossible to tell, whether or not this is a class constructor call, or rather a generic function call. Observe this :

```
1  /* a function call or object creation?
2  In ZoomBA, function call. In Python, anything! */
3  h = Hello()
4  // clear object creation in ZoomBA
5  h = new ( Hello )
```

and this is why we do believe the Pythonic way is ambiguous. The ZoomBA tenet of minimisation of code footprint must also abide by the tenet of maintainability. Moreover, ZoomBA is prototype based, hence none of the ZoomBA objects are actually a JVM class objects. They are, in the truest sense, dictionaries holding functions and properties.

10.1.2 *Instance Fields*

Objects are property buckets. I choose to design ZoomBA precisely this way, the JavaScript way. Pitfalls are obvious, and see a nice discussion here. Those are not a problem for us here much, and subsequent discussion would show why. In any case, object without any fields or rather say properties are meaningless objects. Any object can be given any property at runtime. Observe the following :

```
1  // object creation in ZoomBA
2  h = new ( Hello )
3  h.greet_string = "hello, world" // create a field : greet_string
4  println( h.greet_string ) // access the field at runtime
```

Every field is public, and we are Pythonic. As the proverb goes *"Never design a tool for fools, while you were designing version 1.0, the fools upgraded themselves to version 2.0"* . The Java naked field access without any getter and setter is 1.5 times faster than that of the getters and setters.

But we all agree that is a bad way of randomly assigning properties to an object. Thus, we want to somewhat put properties inside the object in the time of creation.

One easy way out is as follows:

```
1  def SomeClass : {  'a' : 42, b : false  }
2   // now, instance is based on SomeClass
3  instance = new ( SomeClass )
```

One should also note that, both normal and string literals are allowed, much like JavaScript. This is the easiest way to create a prototype based object. In the runtime, we can actually change the values, if we desire:

```
1  instance = new ( SomeClass ,  b = true )
2  // now a is 42, but b is true
```

But this is not what we really want. We want something which can programmatically initialize the instance, while it is being prepared. That is next.

10.1.3 *Constructor*

This brings to us the notion of constructor. There is no constructor in ZoomBA, nothing actually constructs the class. There is obviously an instance initialiser. This is defined as such :

```
1  def Hello{
2      def $$(){
3          $.greet_string = 'Hello, World!'
4      }
5  }
6  h = new ( Hello )
7  println( h.greet_string )
```

Now, this *$$()* is a specific function, which is the instance initialiser. The special identifier *$* in the context in a class also is a specific identifier, identifying the instance of the class being passed, this is pythons *self*. One obviously can pass arguments to the constructor :

```
1  def Hello{
2      def $$(greetings='Hello, World'){
3          $.greet_string = greetings
4      }
5  }
6  // the ordinary folks
7  h1 = new ( Hello )
8  println( h1.greet_string )
9  // from the Penguins of Madagascar
10 h2 = new ( Hello   , "Hey, Higher Mammal! Do you read?" )
11 println( h2.greet_string )
```

10.1.4 *Instance Methods*

This brings the question of how do we *encapsulate* the state of the object. Sometimes, you do not want to let people access stuff. You want to handle the properties of an instance yourself. Those are done with the instance methods, constructor method is one special type of instance method:

```
1  def Hello{
2      def $$(greetings='Hello, World'){
3          $.greet_string = greetings
4      }
5      def greet(){
6          println( $.greet_string )
7      }
8  }
9  h1 = new ( Hello )
10 h1.greet()
```

Obviously one can pass arguments to any instance method:

```
1  def Hello{
2     def $$(greetings='Hello, World'){
3         $.greet_string = greetings
4     }
5     def greet(person="Everyone"){
6         println( 'To @%s : %s\n', person ,  $.greet_string )
7     }
8  }
9  h1 = new ( Hello )
10 h1.greet("ZoomBA")
```

10.2 statics & prototype based inheritence

To have class level properties and methods, which gets initialised only once, we must have *static* constructs, which are indistinguishable from prototype.

10.3 operators

ZoomBA supports operator overloading. The following operators and functions can be overloaded :

operator	method
toString	$str
hashCode	$hc
equals	$eq
compareTo	$cmp
+	$add
-	$sub
*	$mult
/	$div
+=	add
-=	sub
*=	$mult$
/=	div
**	$pow
unary -	$neg
		$or
&	$and
^	$xor
|=	or
&=	and
^=	xor

10.3.1   *Example : Complex Number*

As an example, we present the class Complex number, which overloads lots of operators :

```
1  def Complex {
2      def $$ (x=0.0,y=0.0){
3          $.x = Q(x)
4          $.y = Q(y)
5      }
6      def $str(){
7          str('(%f,i %f)', $.x, $.y)
8      }
9      def $eq(o){
10         ( $.x == o.x && $.y == o.y )
11     }
12     def $hc(){
13          31 * $.x.hashCode() + $.y.hashCode()
14     }
15     def $neg(){
16        new (Complex , -$.x , -$.y)
17     }
18     def $add(o){
19        new (Complex , $.x + o.x , $.y + o.y )
20     }
21     def $sub(o){
22        new (Complex , $.x - o.x , $.y - o.y )
23     }
24     def $mul(o){
25        new (Complex , $.x * o.x  - $.y * o.y , $.x * o.y + $.y * o.x )
26     }
27     def $div(o){
28        mod = ( o.x **2  + o.y **2 )
29        x = ( $.x * o.x  + $.y * o.y )/mod
30        y = ( $.y * o.x  - $.x * o.y )/mod
31        new (Complex , x, y )
32     }
33  }
34  // test it out :
35  c0 = new (Complex )
36  c12 = new (Complex , 1, 2)
37  c21 = new (Complex , 2, 1)
38  // test some?
39  println( -c12 )
40  println ( c0 == -c0 )
```

PRACTICAL ZOOMBA

Examples should be abundant, and this chapter is massively inspired from Programming Pearls. The idea behind this chapter is many examples that happens practically, and how to solve the problems in least possible code, in the best possible way. Although the whole book is pretty abundant in examples - this chapter has its own advantages.

11.1 A NOTE IN STYLE

Standards are for humans, and not otherwise. Being said that : A guideline is a guideline and is not a law of a country or nature that one abides by. Being said that, it is more of the matter of taste, readability, and understandability and something called reproducibility of a bug. Therefore the notion of test flow is of utmost importance. Test cases should fail when app failed, and should pass only when app is working. This tenet drives the guidelines of coding. Therefore, the following subsections are for both the author and the reviewer.

11.1.1 *Comments*

Comments are not after thought. For any standard library use, comment why and what we are trying to accomplish. ZoomBA has a minimalistic syntax, so it is better to explain why we are doing what we are doing. This is a NO GO :

```
1  assert ( size( @ARGS ) >= 2 ,  'boom!' )
2  #(some_param, some_other) = @ARGS
```

This, for example is a GO:

```
1  assert ( size( @ARGS ) >= 2 ,  'Problem : Args are less than 2!' )
2  #(some_param, some_other) = @ARGS  // storing the args into two params as tuple
```

11.1.2 *Naming Conventions*

Naming are to be done by very simple Pareto optimality. If looking at the name, out of 10, 8 developers can not figure out what it is for, then that name is useless. Change it, continually till 80% developers can understand what something means.

One can use *caml* casing or can use underscore, but mixing them in a single file produces readability problem. So, please do not use it. *thisIsABigName* and *this_is_a_big_name* both are acceptable, as long as they are not in the same file. But, *iamamoron* is not acceptable as any name.

When in doubt, please use expanded names rather than the cryptic ones. Don't be shy of using variables, because variable assignment and get value is fast. By design, it is not possible to write a ZoomBA code beyond 42 lines, unless you are using it for System design, which you should not.

11.1.3 *Explicit Conditionals and Iteration*

ZoomBA goes for provable, correct coding, rather than code that feels like "Let there be Code!". Therefore, it is advised to use the **declarative functional style** where it suits. If it means twice the runtime, sometimes, it is OK to have twice the runtime. In short, we generally do not want this :

```
1  // sum up even and odd numbers in a collection col
2  tot_even = 0
3  tot_odd = 0
4  for ( x : col ) {
5    if ( 2 /? x ) {
6        tot_even += x
7    }   else {
8        tot_odd += x
9    }
10 }
```

Instead, this is preferable :

```
1  #( total_even, total_odd ) = fold ( col, [0,0] ) as  {
2    // generate an index mapping which collector to be used
3    inx = $.item % 2
4    $.partial[inx] += $.item // add the item up
5    $.partial // return
6  }
```

This demonstrates the succinct way in ZoomBA.

11.1.4 *Assertions*

Raise assertions, and raise it fast. ZoomBA is a dynamically typed, declarative language. It is of utmost important to check for errors as soon as it happens, and try not to manage pass it. That means :

```
1  def some_brilliant_function(x=42,y=false){
2    assert( x isa 'int' , 'Wrong type of x : needs int' )
3    assert( y isa 'bool' , 'Wrong type of y : needs bool' )
4    //... now proceed...
5  }
```

This should not be taken as The Type Debate but a way to mitigate what is expected. In the same way, this paves the way for :

```
1  def function_accepting_json( json_string ){
2      #( json_obj, err ) = json( json_string)
3      panic ( empty(json_obj) , 'Parse error in Json! ' + str(err) )
```

```
4        assert ( "field1" @ json_obj , "Why field1 is not passed?" )
5        // continue the rest...
6  }
```

11.2 ASSORTED THEORETICAL EXAMPLES

11.2.1 *A Game of Scramble*

We are sure you guys have know the game of jumbled up words. For example, someone gives you "Bonrw" and you need to say : "Brown" ! Suppose we ask you to write a program to do it. The simplest solution is this :

if we sort the all the words in a dictionary letter by letter and then use that sorted word as a key?
Then we can easily solve the problem by sorting on the letters of the word given and then checking if
that as key exist in the dictionary, then, find all possible matches !

```
1  word_dict = mset(file('words.txt')) as {
2               // generate char array
3               ca = $.o.toCharArray()
4               // generate a key by sorting and concatenating
5               key = str(sorta(ca),'')
6            }
7  // now a jumbled up word exists ?
8  ca = jumbled_word.toCharArray()
9  key = str( sorta( ca ) , '')
10 matches = word_dict[key] ?? [] // in case key is not found?
11 println(matches)
```

11.2.2 *Find Anagrams of a String*

The problem is, suppose someone gave you a string : "zoomba" say. The idea is to list all possible words with the letters in it. The letters may or may not have meaning, but that is ok. How to solve this problem?

The easy method is to generate permutation of all the letters, and check if it occurred before. Thus, given *word* is the string :

```
1  // create a set of words
2  words = set()
3  // get the sorted version in an array
4  letters = sorta ( word.toCharArray() )
5  // generate the args for permutations
6  args = list( [0 : #|letters|] ) as { letters }
7  join(@ARGS = args ) where {
8      continue ( $.o != letters )
9      v = str($.o,'')
10     // add the permutation to the words
11     words += v
12     // no need to store anything
13     false
14  }
15 // now words has all the anagrams
```

and this solves the problem. Notice that many advanced concepts are being used in the solution.

A related question is this :

Given a list of strings, return a list of lists of strings that groups all anagrams.

Ex. given [trees, bike, cars, steer, arcs]

return [[cars, arcs] , [bike], [trees, steer]]

To solve the problem, use the same approach :

```
1  strings = [ 'trees' , 'bike' , 'cars', 'steer', 'arcs' ]
2  words = dict()
3  lfold (strings) as {
4     key = str( sorta( $.o.toCharArray() ),'')
5     (words[key] += $.o) ?? (words[key] = list( $.o ))
6     }
7  println( words.values() )
```

11.2.3   Sublist Sum Problem

Given a list of integers, and a value sum, determine if there is a sublist of the given list with sum equal to given sum. There is a dynamic programming solution - which is left for the reader to figure out. Our solution would be minimalist. Observe that the solution is finding all possible combinations of the list, and check where the sum comes up. So, we notice that the previous anagram problem is the building block of this problem too! So, suppose the list of integers are stored in li :

```
1  // create a set for solutions
2  solutions = set()
3  // get the sorted version of the list
4  li = sorta ( li )
5  // generate the args for permutations
6  args = list( [0 : #|li|] ) as { li }
7  join(@ARGS = args ) where {
8     continue ( $.o != li )
9     // we need the sum
10    sm = sum($.o)
11    continue( sm != sum )
12    sa = sorta($.o) // we need combinations
13    v = str(sa)
14    continue( v @ solutions )
15    // add the combination to the solutions
16    solutions += v ; false // store nothing
17    }
18 // now solutions has all the solutions to the problem
```

11.2.4   Sublist Predicate Problem

Observe that, the most generic way to represent all of the above problems is by introducing a predicate $P(\$)$, and stating the problem as such :

Given a list , and a predicate $P(\$)$, determine if there is a sublist of the given list where $P(\$)$ is True.

Thus, the generic solution becomes :

```
1   // create a set for solutions
2   solutions = set()
3   // get the sorted version of the list
4   li = sorta ( li )
5   // generate the args for permutations
6   args = array{ li }( [0 : #|li|] )
7   perms = join( @ARGS = args )  where { continue ( $.o != li )
8              continue( !P( $.o ) )
9              l = sorta( list( $.o ) ) // we need combinations
10             v = str(l,'#') ; continue( v @ solutions )
11             // add the combination to the solutions
12             solutions += v ; false // store nothing
13        }
14   // now solutions has all the solutions to the problem
```

11.2.5   *List Closeness Problem*

Suppose there are points in a metric space, collected in a list. There are two such lists, and we need to check if these lists are sufficiently close to one another or not. Formally, given the distance function $d(,)$, and two collections (of same size N) A, B, they are close iff there exists permutations of A, B defined as $\pi_m(A)$ and $\pi_n(B)$ such that :

$$\forall i \in \{0, ..., N-1\} \,;\, d(\pi_m(A[i]), \pi_n(B[i])) < \epsilon$$

where m and n represents the Permutation Index, and ϵ is the closeness. In simple english, given two list of numbers of the same size, and a closeness number ϵ, this looks:

$$\forall i \in \{0, ..., N-1\} \,;\, |\pi_m(A[i]) - \pi_n(B[i])| < \epsilon$$

This is not a hard problem, but is formulated as such. To solve it, observe that given the distance metric, one can calculate the distance of all the points from a base point, in case of numbers, that would be o. That would let us create an order relation over the numbers. Thus, we can order the numbers in the ascending order, and then i'th element of A must be close to i'th element of B. That is, suppose A_s, B_s are the sorted versions of A, B, then :

$$\forall i \in \{0, ..., N-1\} \,;\, |A_s[i] - B_s[i]| < \epsilon$$

and thus, the ZoomBA solution is :

```
1   AS = sorta(A)
2   BS = sorta(B)
3   e = 0.01 // say?
4   s = #|AS| // the size
5   close = !exists( [0:s] ) where {#|AS[$.o] - BS[$.o]| > e}
```

11.2.6   *Shuffling Problem*

The problem statement is :

Given a string e.g. "ABCDAABCD". Shuffle the string so that no two similar letters together. E.g. AABC can be shuffled as ABAC.

So, how to solve this? As always, we formalise the problem. First we find that how many same alphabets are present, and make a group. Then, we go round robin over all the groups and exhaust the groups. When there would be no solutions? Clearly using the Pigeonhole Principle one can say, when one group would have more characters than the rest of the characters plus 1. For example, there can not be any solution for the string "AAA" : trivial, and "AAAB".

```
1  word = 'ABCDAABCD'
2  m = mset(word.toCharArray() )
3  #(min,Max) = minmax(m) where { size($.l.value) < size($.r.value) }
4  solvable = ( #|word| + 1 < 2 * #|Max.value| )
5  ! ( solvable ) || bye('sorry, no solution exists!')
6  keys = list(m.keySet())
7  shuffle(keys)
8  len = #|keys|
9  result = fold ([0:#|word|],'') as {
10        i = $.index
11       key = keys[i % len]
12      $.partial + m[key][0]
13 }
14 println(result)
```

Observe the use of the *bye()* function. When it is called, it simply returns from the calling function, with the string as return value. Wonder where it can be practically applied? We can think of one. Suppose we are going to go at a diner where girls and boys came. Now, there would be (if the soap operas are slightly true) boys and girls who were involved with one another. Thus a relationship existed is defined by $_xEX_y$ which means x, y were emotionally attached. Now, an extended relationship can be found, extending R, so that $_xEX_y$ and $_yEX_z$ implies $_xEX_z$ where N stands for do not talk. Clearly, N is a set generate by EX. Clearly none in the same N set wants to talk to one another, and thus do not want to seat side by side. That brings the problem, and now you see, how emotional stuff can generate a nice computer algorithm, that Microsoft asks in Interviews.

11.2.7 *Reverse Words in a Sentence*

Suppose a sentence reads : *"This is an utter waste of time"*, we should reverse the order of the words.

```
1  line = 'This is an utter waste of time'
2  words = tokens( line, '\\S+') as { $.o }
3  words = words ** -1
4  result = str(words, ' ')
```

We can reduce it further :

```
1  line = 'This is an utter waste of time'
2  $result = ''
3  tokens( line, '\\S+') as { $result += ' ' + $.o ; continue }
4  $result = $result[1:-1] // substring with index 1
```

11.2.8 *Recursive Range Sum*

Write a recursive function: *sum(x,max)* that calculates the sum of the numbers from x to max (inclusive). For example, *sum*(4, 7) would compute $4 + 5 + 6 + 7$ and return the value 22. The function must be recursive so you are not allowed to use any conventional loop constructs.

```
1  def sum(x,max){
2     dif = max - x
3     x*(diff + 1) + diff*(diff+1)/2
4  }
```

This function uses integer arithmetic, and hence is recursive. Add and subtract are actually recursion. Lets solve it in another way :

```
1  def sum_func(x,max){
2     if ( max == x ) return x
3     return ( max + sum_func(x, max - 1 ) )
4  }
5  size( @ARGS ) > 1 || bye('I need two arguments!')
6  #( x , max )  = [ int(@ARGS[0]) , int( @ARGS[1] ) ]
7  println (sum_func(x,max))
```

11.2.9 *String from Signed Integer*

With input as a integer, write an algorithm to convert that to string without using any built in functions. It may be a signed number.

```
1  def to_str(i){
2     if ( i == 0 ) return "0"
3     sign_val = (i < 0 )?'-':'' // same here, not using standard
4     i = ( i> 0 )?i:-i // not using any standard function at all
5     s = ''
6     for ( ; i > 0 ; i /= 10 ){
7        s =  str( i % 10) + s
8     }
9     sign_val + s
10 }
11 println ( to_str( int(@ARGS[0] ) ))
```

11.2.10 *Permutation Graph*

Given an integer "n" and pair of "I" swapping indices, generate the largest number. Swapping indices can be reused any number times.

```
n : 1243
Indices:
(0,3)
(2,3)
Answer:
3421, 3214, 4213, 4231,...
The result is 4231 which is the largest.
```

The problem is that of Permutation Graph. Observe, then, the rules lets you generate nodes from the starting node. Also observe that the size of the graph generated by the rules is finite, the graph generated is strictly a subgraph of the overall permutation graph, which is finite by definition. A subset of a finite set is finite.

So, how to generate the graph? Obviously we need to apply the rules again and again. But there is a chance of a cycle.
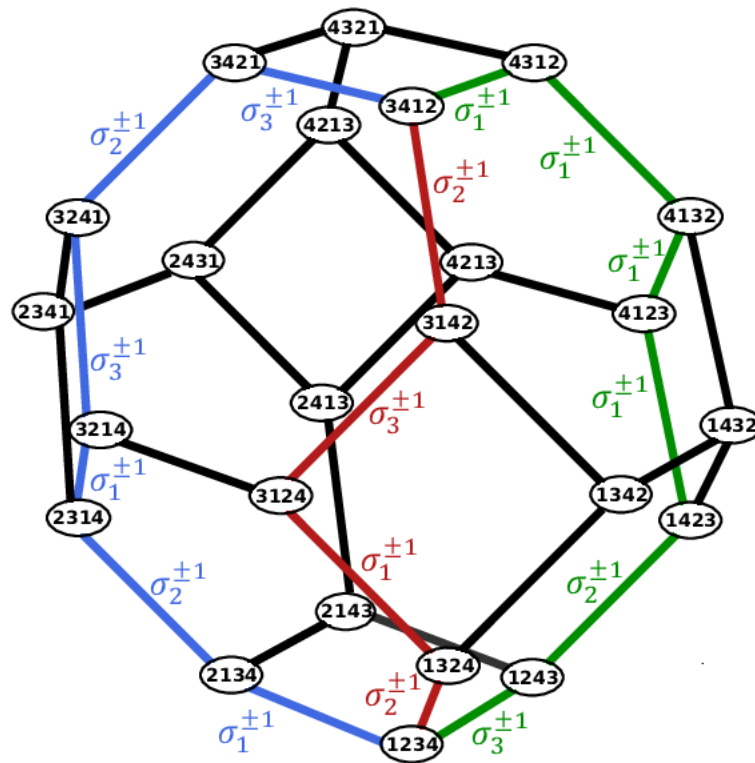
FIGURE 11.1 – Permutation Graph.

The problem here, is a very well known problem: Graph Exploration. Given this problem came from Facebook is quite apt. So, how do we solve it?

1. Keep a dictionary of traversed nodes, or say integers : nodes := { node : traversed ? false }
2. Initialise nodes to starting seed (1243).
3. Pick one item from the node set, which is not traversed, and apply both the rules.
4. If the results generates new nodes, not already in nodes, then add them to nodes
5. If traversal yield old nodes, mark the node as traversed : true
6. check if there is any nodes left to pick and traverse
7. The exploration is done, now go over all the nodes and check which one is the largest.

And here is the ZoomBA code :

```
1   nodes = { 1243 : false }
2   rules = [ [0,3] ,[2,3] ]
3   /* generate the permutation */
4   def get_node(cur,rule){
5       c_array = str(cur).toCharArray()
6       t = c_array[rule.0]
7       c_array[rule.0] = c_array[rule.1]
8       c_array[rule.1] = t
9       int ( str(c_array,''))
10  }
11  /* Explore the Graph */
12  while (true){
13      /* observe the division over a dictionary
14          This generates a set of keys where
15          value is the right operand */
16      not_traversed = nodes / false
17      break(empty(not_traversed))
18      // mark as true
19      nodes[ not_traversed[0] ] = true
20      // get the current node
21      cur = not_traversed[0]
22      lfold (rules) as {
23              // generate the next node
24              nn = get_node( cur, $.o )
25              // in case it exists, continue
26              continue(nn @ nodes )
27              // add to the dictionary for nodes
28              nodes[nn] = false
29      }
30  }
31  // generate the min, max
32  #(m,M) = minmax(nodes) as { $.o.key  }
33  println(M)
```

11.2.11   *Find Perfect Squares*

Find the list of perfect squares between two given numbers. So, how do we solve it? We go back to formalism : the perfect squares are:

$$s = \{x | \exists n \in \mathbb{N} \ s.t. \ x = n^2 \ and \ b \le x \le e\}$$

Thus, to solve it :

```
1   def find_all_perfect_squares( begin, end ){
2       b = floor ( begin ** 0.5 )
3       e = ceil ( end ** 0.5 )
4       select ( [b:e+1] ) where {
5           sq = $.o ** 2
6           sq <= end && sq >= begin
7       } as { $.o ** 2 }
8   }
9   println( find_all_perfect_squares(10,100) )
```

11.2.12   *Max Substring with no Duplicate*

Given "s" a string, find max size of a sub-string, in which no duplicate chars present. To solve the problem, observe that a substring is defined by the start index : i and end index : j, given the string. Thus, all possible substrings are nothing but all possible combinations of (i, j). Thus,

a very simple solution would be to find all possible substrings and see which one is the highest size. The base case is finding if the given string has no duplicates. Hence the code :

```
1   s = 'abacdefghabk'
2   def vals: { large_size : 1 , substring : 'a' } // create a container
3   indices = [0:#|s|].list()
4   result = join ( indices, indices )  where {
5         continue($.0 >= $.1)
6         ss = s[$.0 :  $.1]
7         l = size(ss)
8         continue(large_size > l )
9         set_size = #|set(ss.toCharArray())|
10        continue( set_size != l )
11        vals.substring = ss
12        vals.large_size = l
13        false
14      }
15  println('%s with size %d\n', vals.substring, vals.large_size )
```

11.2.13 *Maximum Product of Ascending Subsequence*

Given a sequence of non-negative integers find a subsequence of length 3 having maximum product with the numbers of the subsequence being in ascending order. As an example: with input $6, 7, 8, 1, 2, 3, 9, 10$ the result would be $8, 9, 10$. Thus, here is the solution :

```
1   l = [ 6,7, 8, 1, 2, 3, 9, 10 ]
2   i = [0:#|l|].list()
3   ans = l[0] * l[1] * l[2]
4   result = [l[0],l[1],l[2] ]
5   join(i,i,i) where {
6      // we need subsequences so...
7      continue( $.0 >= $.1 or $.1 >= $.2 )
8      // we need ascending, so
9      continue( l[$.0] > l[$.1] or l[$.1] > l[$.2] )
10     r = l[$.0] * l[$.1] * l[$.2]
11     continue(r <= ans)
12     result = [l[$.0], l[$.1], l[$.2] ]
13     ans = r ; false
14  }
15  printf('Values %s with product %d %n', str(result) , ans )
```

In this form, we can generalise it, to any number of items instead of 3. Suppose the same problem was asked and the size of the tuple was fixed at m. The function, instead of making it a multiply, one can generalise to any $f(\$)$. The reader should code that general problem, accordingly.

11.2.14 *Minimal Sum of Integers in Digit Array*

Given the array of digits (0 is also allowed), what is the minimal sum of two integers that are made of the digits contained in the array. For example, array: $1, 2, 7, 8, 9$. The min sum $(129 + 78)$ should be 207. To solve this, note that a partition of digits would split the digits into 2 halves. Sort the halves so that the integers generated by the halves are smallest. Now, check the result, and continue. Observe that a partition is nothing but a binary string of same size as the array, 0's defining left partition, while 1's defining right partition. Thus :

```
1   d = [1, 2, 7, 8, 9]
2   l = '1278'
3   r = '9'
4   min = int(l) + int(r)
5   // observe that a partitions are done using bits
6   n = #|d|
7   upto = 2 ** (n+1)
8   lfold [1:upto]  as {
9     selection = $.o
10    left = list() ; right = list()
11    bitmask = 1
12    lfold( [ 0 : n ] , bitmask ) as {
13       bitmask = $.o
14      // bitwise and operation
15      b = selection & bitmask
16      if ( b == 0 ){
17       left += d[ $.index ]
18      } else {
19       right += d[ $.index ]
20      }
21      bitmask = bitmask * 2
22    }
23    // obvious
24    continue ( empty(left) || empty(right) )
25    // if any starts with 0
26    continue ( left #^ 0 || right ^# 0 )
27    // due stuff
28    left = str(sorta(left),'') ; right = str(sorta(right),'')
29    v = Z(left) + Z(right)
30    continue ( v >= min )
31    min = v ; l = left ; r = right
32  }
33  printf('%s + %s = %d\n' , l, r, min )
```

11.2.15 *Ramanujan Partitions*

Given a number N, write a program that returns all possible combinations of numbers that add up to N, as lists. (Exclude the N+o=N). For example, if $N = 4$ return $[[1,1,1,1],[1,1,2],[2,2],[1,3]]$ See more about the problem here. One of Indian greats, rather worlds finest who worked beyond this problem.

To systematically generate the partitions, observe that one needs to select groups of "1"s from the list of N "1" :

$$[[1],[1],[1],[1]] := [[1],[1,1,1]] := [[1,1],[1,1]] := [[1],[1],[1,1]]$$

Thus, the problem is selecting the gaps between these "1"s, and there are $N-1$ of them. Thus we must systematically select between $[1:N]$ gaps, numbered between 0 and $N-1$. Sum the resulting groups, and keep them sorted as the key. To select a number between 0 and $N-1$ use the binary encoding of numbers between 1 to 2^{N-1}.

```
1  def ramanujan_partition(N){
2    // imagine N-1 + symbols
3    // 1+1+...+1
4    // select + symbols
5    max = 2 ** ( N - 1 )
6    partitions = set()
7    for ( n : [1: max] ) { // avoid trivial partition
8      s_rep = str(n,2)
9      s_rep = '0' ** ( N -1 - #|s_rep| ) + s_rep
10     // in s_rep, '1' means select partition from that '+' symbol
11     // for N = 5,    1 + 1 + 1 + 1 + 1
12     // 0001  means     0   0   0   1   => 4 + 1
13     current_partitions = list()
14     cur_val = 1
15     for ( c : s_rep ){
16       continue ( c == _'1' ){
17         current_partitions += cur_val
18         cur_val = 1
19       }
20       cur_val += 1
21     }
22     current_partitions += cur_val
23     sorta( current_partitions )
24     partitions += str(current_partitions )
25   }
26   partitions // return
27 }
28 println( ramanujan_partition(5))
```

11.2.16 *Consecutive Elements in Subset*

Given a set of numbers, find the longest subset with consecutive numbers be it any order. Input: $S = [5, 1, 9, 3, 8, 20, 4, 10, 2, 11, 3]$, we have 2 consecutive sets : $s_1 = [1, 2, 3, 4, 5]$ and $s_2 = [8, 9, 10, 11]$. Ans is s_1.

```
1  S = [5, 1, 9, 3, 8, 20, 4, 10, 2, 11, 3]
2  S = set(S)
3  S = sorta(S)
4  write(S)
5  max_size = 1
6  indices = list(0,1)
7  r = [0:#|S|]
8  join( r,r ) where {
9    i = $.l
10   j = $.r
11   continue( i >= j )
12   consecutive = ( index{ S[$-1] + 1 != S[$] }([i+1:j]) < 0 )
13   continue( not consecutive )
14   continue( max_size > j - i )
15   max_size = j - i
16   indices[0] = i ; indices[1] = j
17   false
18 }
19 write( S[[indices.0 : indices.1]]  )
```

11.2.17 *Competitive Array*

Given an array of elements, return an array of values pertaining to how many elements are greater than that element remaining in the array. Ex. for $[3, 4, 5, 9, 2, 1, 3]$ Return $[3, 2, 1, 0, 1, 1, 0]$.

First element is 3 because $3 < 4, 5, 9$. Second element is 2 because $4 < 5, 9$.

```
1   items = [3,4,5,9,2,1, 3]
2   d = mset(items)
3   lfold ( d ) as { d[$.key] = [ size($.value) , 0 ] }
4   keys = list(d.keySet())
5   keys = sorta(keys)
6   higher_count = 0
7   rfold(keys) as {
8       t = d[$.o]
9       t[1] = higher_count
10      higher_count += t[0]
11  }
12  result = list(items) as {  (d[$.o])[1] }
13  println(result)
```

This is so trivial that no explanation is given for the code.

11.2.18 Sum of Permutations

Find the sum of all 4 digit numbers formed from $1, 2, 3, 4$ without repetition. The easy way is to do the permutations and then summing them up.

```
1   S = sum( perm([1,2,3,4]) ) as { int( str($.o, '')) }
2   println(S)
```

The result can be found algebraically. Observe that the summation would be over the digits $1, 2, 3, 4$ and the integers created would have the power places as $10^3 + 10^2 + 10 + 1 = 1111$, so, each for every permutation starting with digit d would sum up to $1111 \times (1 + 2 + 3 + 4) = 11110$. Now, how many permutations stating with a digit d ? That would be found by taking that digit out of the permutation, and permutating the rest of the digits : $(4 - 1)! = 3! = 6$, hence the total would be 66660.

11.2.19 Print a String Multiple times

Print a character 1000 times without using loop and recursion. Sometimes, I also, get astonished by the amount of IM that an interview can produce. The one who asked this question probably never heard of recursively enumerable languages. In any case, I just could not avoid this, because, in ZoomBA, you actually can. I mean seriously, you can.

```
1   s = 'Moron asked me this' // stands for what it is, precisely
2   println( s** 1000 ) // prints s a 1000 times
```

11.2.20 Next Higher Permutation

How do we generate next permutation? By swapping two indices, and we have $^nC_2$ options. So, one solution would be simply to iterate over and find the minimum of all next permutations which is higher than the current one:

```
1   string_val = @ARGS[0]
2   nu = int( string_val )
3   num_arr = string_val .toCharArray()
4   r = [0 :  #| string_val  | ]
5   // pick the highest one
6   nhp = int ( str( sortd(num_arr) ,'') )
7   nhp != nu || bye('Dude, got to be kidding me!')
8   join (r,r) where {
9        i = $.0 ; j = $.1
10       continue( i >= j )
11       num_arr = string_val.toCharArray()
12       t = num_arr[i]
13       num_arr[i] = num_arr[j]
14       num_arr[j] = t
15       nn = int ( str(num_arr,''))
16       continue( nn < nu )
17       if ( nn < nhp ){ nhp = nn }
18       false
19   }
20  println(nhp)
```

Now, this works, so, question is, can we make it any faster? Turns out, we can. Given next higher permutation is possible, there has to be one discrepancy where $D_i < D_j$ with $i < j$. The objective would be to find minimum of such a span $|i - j|$ is minimum, with maximum value of i. So, we can start from the right, with $i = n - 2$ and $j = n - 1$, and then searching left for a match.

```
1   def next_higher_perm( n ){
2      digits = str(n).toCharArray()
3      len = #|digits|
4      j = len - 1
5      while ( j > 0 ){
6         i = j - 1
7         while ( i >= 0 ){
8         if ( digits[i] < digits[j] ){
9            // swap and we are done
10           t = digits[i]
11           digits[i] = digits[j]
12           digits[j] = t
13           return int( str(digits,'') )
14        }
15        i -= 1
16        }
17        j -= 1
18     }
19     n // nothing can be done so...
20  }
21  println ( next_higher_perm( @ARGS[0] ) )
```

11.2.21 *Maximal Longest Substring Problem*

Given a string S , print the longest substring P such that lexicographically : $P > S$. One may assume that such substring exists.

```
1  S = "hello , world"
2  def max_large_sub(s){
3     len = #|s|
4     j = index([1:len]) where { s[0] < s[$.o] }
5     j > 0 ? s[0:j+1] : ''
6  }
7  println(max_large_sub(S))
```

11.2.22 *Partitions which are Palindromes*

Given a string, print all possible palindromic partitions.

```
1   s = @ARGS[0]
2   r = [0 : #|s|]
3   partitions = join( r, r ) where  {
4      i = $.0 ; j = $.1
5      // do not consider trivial one chars
6      continue( i >= j )
7      ss = s[ i : j ]
8      // replace the join with reassignment of item
9      ( (ss ** -1) == ss )
10  } as { s[$.0:$.1] }
11  println(partitions)
```

11.2.23 *Triple Sum*

Count triplets in a collection with sum smaller than a given value. One must read the theory behind here. Without much ado, one can solve this problem by using the joins :

```
1   items = tokens( @ARGS[0] , '\d+' ) as { int( $.o ) }
2   items = sorta(items)
3   num = int( @ARGS[1], 0 )
4   r = [ 0 : #|items| ]
5   results = join(r,r,r) where {
6      // preserve order
7      continue( $.0 >= $.1 || $.1 >= $.2 )
8      sum = items[$.0] + items[$.1] + items[$.2]
9      sum < num
10  } as { [ items[$.0] , items[$.1] , items[$.2] ]  }
11  println(results)
```

11.2.24 *Pythagorean Triplet*

Find if there exists Pythagorean Triplets in an array. That is, given an array of integers, write a function that returns true if there is a triplet (a, b, c) that satisfies $a^2 + b^2 = c^2$. In another word, find all of them. This is what we find here.

```
1  items = tokens( @ARGS[0] , '\d+' ) as { int( $.o ) }
2  items_set = set(items)
3  r = [ 0 : #|items| ]
4  results = join (r,r) where {
5     // preserve order
6     continue( $.0 >= $.1 )
7     tot= items[$.0] ** 2 + items[$.1] **2
8     // note the use of num() function
9     t = tot ** 0.5 ; t = num(t)
10    t @ items_set
11 } as { [ items[$.0] , items[$.1] , t ] }
12 println(results)
```

11.2.25 *Substring as Permutation*

Given a string, find whether it has any permutation of another string. For example, given
"abcdefg" and "ba", it should return true, because "abcdefg" has substring "ab", which is a
permutation of "ba".

```
1  def perm_exists(s1,s2){
2     small = s1 ; large = s2
3     if ( #|s1| > #|s2| ){   small = s2 ; large = s1 }
4     small_arr =  small.toCharArray()
5     small_set =  set(small_arr)
6     large_len = #|large| ; small_len = #|small|
7     i = 0
8     while( i <  large_len ){
9        if ( i + small_len > large_len ) { return false }
10       continue(  large[i] !@ small_set  ){ i += 1 }
11       ss = large[ i : i + small_len - 1]
12       if ( ss.toCharArray() == small_arr ){ return true }
13       i += 1
14    }
15    return false
16 }
```

Note the usage of *A == B* for the checking of *collection A is a permutation of collection B*

11.2.26 *Pivot Partition*

Given an unsorted array of integers, you need to return maximum possible n such that the
array consists at least n values greater than or equals to n. Array can contain duplicate values.
Sample input : $[1, 2, 3, 4]$, output : 2. Sample input : $[900, 2, 901, 3, 1000]$, output: 3.

```
1  def pivot_partition(a){
2     ma = mset(a)
3     ma = dict(ma) as { [ $.key , size($.value)]  }
4     keys = list( ma.keySet() )
5     keys = sortd(keys)
6     greater_eq = 0
7     i = index(keys) where {
8        greater_eq += ma[$.o]
9        greater_eq >= $.o
10       }
11    ( i > 0 ) ? keys[i]  :  'None'
12 }
```

Note the use of body variable. . Observe this behaviour:

```
1  x = 0
2  fold ( [0:10] ) as {
3    x+= 1
4    println(x) // goes from 0,1,2,... 9
5  }
6  println("But x is... " + x ) // prints 0
```

One can see that the variable 'x' was not eventually modified. But repeated iteration of the fold will preserve the state of the variable.

11.2.27 *Find all subsequences where Predicate*

One problem statement is, given a list l, find all possible sublists such that the sum of elements in the sublists is a number N. Formally, find the list l_N such that:

$$l_N = \{x \mid x \subset l \text{ and } \sum_{y \in x} y = N\}$$

that is, $l_N \subset 2^l$ where 2^l is the power list(set) of list l. We use the same idea as Ramanujan Partition, but this time use the index set as the bitmap . We also generalise the notion of sum using a generic predicate:

```
1  // define the predicate
2  P = def(a, N ){ sum(a) == N }
3  def find_sublists(l,P){
4     L = size(l)
5     // total no of subsets ( cardinality of power set)
6     n = 2 ** L
7     fold( [1:n],set() ) as {
8       // right bitmap
9       bit_map = str($.o,2)
10      items = rfold(bit_map.toCharArray() , list() ) as {
11        // that is why read from right
12        continue($.o == '0')
13        /* put items whose indices are to be selected
14         -ve indices read from right! */
15        $.p.add(0, l[- $.index - 1] ) ; $.p
16      }
17      continue( ! P(items, N ) )
18      // add them up
19      $.p.add( items ) ; $.p
20    }
21  }
```

11.3 ASSORTED PRACTICAL EXAMPLES

This section contains examples which would probably never be asked in any interviews, if anyone asks the in an interview, please join that firm immediately.

11.3.1 *Generic Result Comparison*

Most of the time, it is of importance that we compare results coming from an older and a newer system. Both are to be somehow equivalent. Obviously these results are list of complex objects, which differs with the versions. Thus, the objects which comprise of the old list *ol* would be slightly different than that of the new list *nl*.

Thus, the only way to handle these sort of thing would be using the projection operator. That is, isolate the set of fields which are equivalent , that would be a list of tuple : $C_i = (_O C_i, _N C_i)$ where $_O C_i$ is the old field, while $_N C_i$ is the new field. That is really not the most generic way, but let's assume it is. Now, it boils down to doing the projection on these :

$$t_o = \pi_{_O C}(O_o)$$

and

$$t_n = \pi_{_N C}(O_n)$$

and now, there are lists of it. Thus, now, these two list should be equal. Hence, in ZoomBA, here is the code one would write :

```
1  // old tuple list , generated by projecting old columns
2  otl = list (ol) as { o = $.o ; str( OC , '' ) as {  o[$.o] ?? '' } }
3  // new tuple list , generated by projecting new columns
4  ntl = list (nl) as { o = $.o ; str( NC , '' ) as {  o[$.o] ?? '' } }
5  // now compare 2 list of strings!
6  matches = ( otl == ntl )
```

11.3.2 *Verifying Filter Results*

Sometimes people tend to write imperative code to apply filter on search results. That is, given the result is a list of objects, all the objects would be yielding true while applying predicate $P()$. Thus, formally, a filtered collection F over a collection C is :

$$\forall x \in F \, ; \, P(x) = True \text{ and } \nexists y \in C \text{ s.t. } P(y) \text{ and } y \notin F$$

and this $F \subseteq C$. The question is how to test for filtering? Given we already have a predicate $P()$ defined :

```
1  // the dev code:
2  Fd = select(C) where { P($.o)  }
3  // thus, this is legal but bad
4  filtered = (F == Fd)
5  // instead, we do it the negated way
6  filtered = ( exists(F) where { ! P($.o) }  ) &&
7            ( exists(C - F) where { P($.o) }  ) && F <= C
```

But this has a problem. The solution loops over both F and C. Will there be a way to reduce the looping? Given C, F are sets, it is easy :

```
1  filtered = ( exists(F) where { !P($.o) && $.o !@ C } ) &&
2            ( exists(C - F) where { P($.o) }  )
```

11.3.3 *Storing Positions of Elements in a String*

Given a string where there are numbers and some numbers are repeated, e.g. 13413124..., design a data structure for it and the data structure should store positions of each number. Such a problem requires simple data structure, a map with a list as value would do just fine :

```
1  s = "13413124"
2  d = lfold (s.toCharArray(),dict()) as {
3      k = int($.o)
4      if (  k !@ $.p  ){ $.p[k] = list() }
5      $.p[k] += $.i // store the index
6      }
7  println(d)
```

11.3.4 *Find Largest N : heap()*

You are given a large set of integers, which are not sorted. Figure out a method to retrieve the largest N elements, in $\Theta(n)$ run time. For this, we use *heap()* .

```
1  N = 10
2  l = list([0:100]) as { random(1000) }
3  // create a max-heap
4  h = heap(N, true)
5  h ++= l // add all
6  println(h)
```

To create a min-heap, just use the argument as *false* to *heap()* function. The first argument is the heap size.

11.3.5 *Rational Number Representation*

Write a function which, given two integers (a numerator and a denominator), prints the decimal representation of the rational number "numerator/denominator". Since all rational numbers end with a repeating section, print the repeating section of digits inside parentheses; the decimal printout will be/must be. Examples: $(1,3) = 0.(3)$, $(2,4) = 0.5(0)$; $(22,7) = 3.(142857)$.

```
1   def rational(n,d ){
2      // these are stored in list
3      digits = list()
4      // in case there is a loop/recurrence
5      visited = dict()
6      // find the integer part
7      int_part = str(n/d )
8      // rest is the new numerator
9      n %= d
10     // index is there to find the spot for each : n
11     // where "(" is needed to be inserted
12     i = 0
13     while(n != 0 ){
14        n = n * 10
15        while ( n < d ){
16           // append 0, only when
17           digits += '0'
18           n *= 10
19           i+= 1
20        }
21        break ( n @ visited ){
22           // found a recurrence
23           digits.add( visited[n] , '(')
24           digits += ')'
25        }
26        // mark the visit of n
27        visited[n] = i
28        // add to the digits
29        digits += (n/d)
30        n %= d
31        i+= 1
32     }
33     str ('%s.%s', int_part ,  str(digits,'') )
34  }
35  print ( rational(1,2) )
```

11.3.6 *Maximum Span of Stock Prices*

Given a stock price for some consecutive days, find the maximum span of each day's stock price.
Span is the amount of days before the given day where the stock price is less than that of given
day. Hence, given the input : $[2, 4, 6, 9, 5, 1]$ the output would be : $[-1, 1, 2, 3, 4, -1]$ The solution
is trivial :

```
1   l = [2,4,6,9,5,1]
2   n = size(l)
3   result = rfold (l,list()) as {
4      cur_index = n - $.i - 1
5      continue(cur_index == 0 )
6      v = $.o
7      #(m,M) = minmax([0:cur_index]) as { $.o }
8      if ( l[m] < v ){
9         $.p.add(0, cur_index - m )
10     }else{
11        $.p.add(0,-1)
12     }
13     $.p
14  }
15  // for the first element
16  result.add(0,-1)
17  println(result)
```

11.3.7 *Recognising String from Languages*

Suppose there is a nice formal grammar from which we need to match strings. Such a formal grammar (a context free one), see here for more disucssion, (you should, because this was one of Google's question) can be easily made by using bitstream alphabet :

$$\Sigma = \{0, 1\}$$

the definition of the language is :

$$L = \{w \mid w = 0^n 1^n \; ; \; n \in \mathbb{N}\}$$

Now, we need to write a function f such that, it accepts a string from that language L, and rejects any other string. How do we envision such a thing? We know there must be abstraction of a stack to solve it, because it is context free. The abstraction is easy :

```
1   // stack based implementation
2   def in_lang(s){
3      c_1 = _'1' ; c_0 = _'0'
4      len = #|s| ; i = 0
5      while ( i < len && s[i] == c_1  ){ i +=1 }
6      if ( i == len ){  return (len == 0 || false) }
7      count = i
8      while ( i < len && s[i] == c_0  ){ i += 1 }
9      if ( i != len ) { return false }
10     ( count* 2 == len )  // return
11  }
12  println( in_lang( @ARGS[0] ) )
```

11.3.8 *Globally Unique ID*

Design a system to return an unique ID for each request. For most of requests, the ID value should increase as time goes, the system should handle 1000 requests per second at least. Timestamps alone is not valid since there might be multiple requests with same timestamps. The solution is not that hard, this works perfectly fine :

```
1   def GUID(){
2      s = str( [0:3], '' )as{ str( INT(time()))}
3   }
```

11.3.9 *Inversions In A Pair of Collections*

You are given two integer arrays A and B. $0 \le i < len(A)$ so i is iterator of array A. $0 \le j < len(B)$ so j is iterator of array B. Find all the pairs (i, j) such that : $i < j$ and $A[i] > B[j]$.

```
1   i_a = [0:#|A|]
2   i_b = [0:#|B|]
3   inversions = join (i_a,i_b) where { $.0 < $.1 && A[$.0] > B[$.1] }
```

Actually, no explanation needed!

11.3.10   *Summation of Binary Integers*

Given two binary numbers each represented as a string write a method that sums up the binary numbers and returns a result in the form of binary number represented as a string. One may assume that input fits in the memory and the input strings are, in general, of different length. Optimize your solution, do not use unnecessary 'if' branching. As an example: *sumBinary('0111101', '1101')* returns 1001010.

In ZoomBA, the solution would be cheating:

```
1  def sumBinary(a,b){
2     i = int(a,2,null) // specify base 2, and default
3     j = int(b,2,null)
4     x = i + j
5     str(x,2)
6  }
```

But we decided to un-cheat, and give a slightly less cheat :

```
1  bits = { 0 : '0' , 1 : '1' }
2  def to_int(a){
3     lfold (a.toCharArray(), 0 ) as {
4        $.p = 2 * $.p + int($.o)
5        }
6  }
7  def to_bin_str(a){
8     r = a % 2
9     n = a / 2
10    bin_rep = bits[r]
11    while ( n > 0 ) {
12       r = n % 2
13       n = n / 2
14       bin_rep = bits[r] + bin_rep
15    }
16    bin_rep
17 }
18 def sum_binary(a,b){
19    x = to_int(a)
20    y = to_int(b)
21    to_bin_str( x + y)
22 }
23 s = sum_binary(@ARGS = @ARGS )
24 println(s)
```

Which works out. However, purists would still argue that we cheated, and did not actually sum it up bit by bit. So, to appease them, as Amazon ordered :

```
1   //                     ABC  :  RC
2   bin_add_logic = { '000' : '00' ,
3                     '010' : '10' ,
4                     '100' : '10' ,
5                     '110' : '01' ,
6                     '001' : '10' ,
7                     '011' : '01' ,
8                     '101' : '01' ,
9                     '111' : '11' }
10  // when carry is 0 append empty...else...1
11  carry_string = { '0' :'' , '1' : '1' }
12  def bin_add(a,b){
13      size_a = #|a|
14      size_b = #|b|
15      #(m,M) = minmax(size_a,size_b)
16      // pad both of them up to max : M with 0
17      a = '0' ** (M - size_a) + a
18      b = '0' ** (M - size_b) + b
19      carry = '0'
20      tot = fold ( [M-1:-1] , '' ) as {
21          key = a[$.o] + b[$.o] + carry
22          add_result = bin_add_logic[key]
23          carry = str(add_result[1])
24          $.p = add_result[0] + $.p
25      }
26      tot = carry_string[carry] + tot
27  }
28  #(a,b) =  @ARGS
29  println ( bin_add(a,b) )
```

11.3.11 Generating System Response Curve

Most of the performance testing done are totally wrong, because most of these rely on a single metric : average, and if someone is lucky, then standard deviation. But that is wrong way to look at it. The right way is using system response curve, which is a pdf. Thus, the algorithm to generate such a curve from log files is this :

1. Start with a bucket size of of a time slice ΔT.
2. A response time t_r belongs to bucket B_k iff $k\Delta T \leq t_r < (k+1)\Delta T$.
3. Count all the response times within each bucket B_k, and divide it by total no of response, store it to each bucket
4. Print all the buckets B_k with the fraction frequency of each bucket.

This, when plotted, with a sufficient thin ΔT produces the nice distribution diagram of the systems response. The code for this is minimal in ZoomBA :

```
1   data = list(file('log_file.txt')) as { float( $.o ) }
2   #(m,M) = minmax(data)
3   printf('Min : %f , Max : %f\n', m, M )
4   bucket_size = 0.001
5   stat = mset(col=data,sorted=true) as { int (($.o - m )/bucket_size) }
6   T = float(size(data))
7   print('Bucket\tTiming')
8   fold(stat) as {
9       printf('%f\t%f%n' , m +  ($.o + 1) * bucket_size ,
10      size(stat[$.o]) * 100.0 / T )
11  }
```

11.3.12   *Shuffling in a Media Player*

If I am designing a media player and I want to store songs and play them in random order,
how will select the next song to play in a way which prevents the same song being played in
consecutive turn?

```
1   // you get the idea
2   songs = { 1 : 'Track1' , 2 : 'Track2' , 3 : 'Track3'  }
3   // shuffles and plays songs
4   def play_songs_after_shuffle( songs ){
5      keys = list( songs.keySet())
6      // while at least one swap did not happen
7      shuffle(keys)
8      lfold(keys) as { println ( songs[$.o] ) }
9   }
10  play_songs_after_shuffle( songs )
```

11.3.13   *Ordering Thread Execution*

Print series 010203040506... using multi-threading. 1st thread will print only 0 2nd thread will
print only even numbers and 3rd thread print only odd numbers. Here is how we do it:

```
1   // use global to lock over #atomic
2   $state = 0
3   // do state management
4   def function(exec_str, cur_state , next_state ){
5      n = 1
6      while(true){
7         // wait for the state to be mine
8         while( state != cur_state );
9         #atomic{
10            // crown jewel of ZoomBA is currying
11            println( #'#{exec_str}' ) ; $state = next_state ; n+= 1
12         }
13      }
14  }
15  // spawn these children
16  t0 = thread() as { function('0' , 0 , 1) }
17  t1 = thread() as { function('2*n - 1' , 1 , 2) }
18  t2 = thread() as { function('2*n' , 2 , 0) }
19  tc = thread()
20  // a bit of wait to display results
21  tc.sleep(300)
22  // do not care...
```

11.3.14   *Asynchronous Computation*

Suppose, we are trying to call a web api, which returns large chunks of data, and we can not call
it synchronously. Therefore, we need to ensure that after the method was successfully executed,
there is a callback method that should get called. This problem is trivial in ZoomBA and here is
how to get it done :

```
1  def call_back(res){
2     println('I am done reading')
3     println(res) // same as @ARGS[0]
4  }
5  ta = async(call_back) as { read( 'https://www.reddit.com' ) }
6  ta.join() // done
```

Note that we are using the eventing mechanism.

11.3.15 Computation of Query

Suppose, the problem is that of generation of queries. As an example, take this query :
Find all the customers who spent >2 minutes on Page "XYZ" & purchased >2 items of coffee_X & gave a review of >3.
The POJOs given are :

```
class PageView {
private String URL;
private String customerID;
private Integer timeSpent;}

class Purchase {
private String productID;
private String customerID;
private Integer itemsPurchased;}

class Review {
private String productID;
private String customerID;
private Integer reviewPoints;}
```

How does one code it? Here is the query:

```
1   page_url = pages['XYZ']
2   customer_ids_pv = set (page_views) as {
3         continue ( ! ( $.URL == page_url &&
4             $.timeSpent > 2 ))
5             $.customerID }
6   product_id = products['coffee_X']
7   customer_ids_pur = set(purchases) as {
8         continue( ! ( $.customerID @ customer_ids_pv &&
9             $.productID == product_id &&
10            $.itemsPurchased > 2 ))
11            $.customerID }
12  customers = set (reviews) {
13        continue( ! ( $.customerID @ customer_ids_pur &&
14            $.productID == product_id &&
15            $.reviewPoints > 3 ))
16            $.customerID  }
17  lfold(customers) as { println( $.o ) }
```

11.3.16 Generating Strings

Given a string (for example: "a?bc?def?g"), write a program to generate all the possible strings by replacing '?' with '0' and '1'.

Example: Input : *a?b?c?*

Output: a0b0c0, a0b0c1, a0b1c0, a0b1c1, a1b0c0, a1b0c1, a1b1c0, a1b1c1.

Observe that, given there are are n question marks (?), the problem is that of generating binary integers from 0 to $2^n - 1$, all should be padded up with '0's to make it n digits. Then, look at the occurrence of the bits in position for the question mark, and replacing them. Thus, a solution is :

```
 1  def generate_strings(template){
 2      ta = template.toCharArray()
 3      n = sum (ta , 0) as {  ( $.o == '?' ) ? 1 : 0  }
 4      p = 2**n
 5      lfold ([0:p])  as {
 6         s = str($.o, 2)
 7         s = '0' ** ( n - #|s| ) + s
 8         count = 0
 9         r = lfold( ta, '') as {
10            if ( $.o == '?' ){
11               $.p += s[count]
12               count += 1
13            }else{
14               $.p += $.o
15            }
16            $.p
17         }
18         println(r)
19      }
20  }
```

The use of such a program is obvious for anyone who wants to make a finite decision game.

11.3.17 *First Unique URL*

Given a very long list of URLs, find the first URL which is unique (occurred exactly once).

```
 1  existing = dict()
 2  uniqs = oset()
 3  for ( u : file('urls.txt')){
 4      if ( u @ existing ){
 5          existing[u] += 1
 6          uniqs -= u
 7      } else {
 8          existing[u] = 0
 9          uniqs += u
10      }
11  }
12  first_uniq = uniqs.iterator.next ?? null
```

Now this is surely some way of doing this. But we may wonder, is there a less verbose way? Yes there is.

```
 1  ms = mset( col=file('urls.txt'), order=true)
 2  mc = find( ms.entries ) where { $.value == 1 }
 3  mc.value // does it
```

In this formulation, there are double iteration. Is there a better way to solve it, in a manner that is more declarative? There is one using *proxy()* function:

```
1  uniqs = oset()
2  d = proxy( dict(), true) as {
3     if ( $.method.name != 'put' ) return
4      if ( $.args[0] @ $.src ){
5          uniqs -= $.args[0]
6      } else {
7          uniqs += $.args[0]
8      }
9  }
10 ms = mset( col=file('urls.txt'), acc=d)
11 first_uniq = uniqs.iterator.next ?? null
```

Notice that use the function *proxy()* to create an interceptor, which gets called whenever any method call on the proxy object happens.

11.3.18 *Conditional Assignment : Casing*

You are given four integers 'a', 'b', 'y' and 'x', where 'x' can only be either zero or one. Your task is as follows: If 'x' is zero assign value 'a' to the variable 'y', if 'x' is one assign value 'b' to the variable 'y'. You are not allowed to use any conditional operator (including ternary operator). Follow up: Solve the problem without utilising arithmetic operators '+ - * /'.

This is one of the founding stone of the declarative paradigm, therefore, it qualifies as a practical problem. The solution, of course can be done using what we explained in chapter 1:

```
1  conditionals = { 0 : a , 1 : b }
2  y = conditionals[x]
```

But, observe that hash is complex structure, we can simplify the problem by :

```
1  conditionals = [a, b ]
2  y = conditionals[x]
```

And that works perfectly well.

11.3.19 *Parking Rearrangement Problem*

Suppose there is a parking with slots named as $1, 2, ..., n$ with a free slot designated as 0. How can one rearrange the cars from a source configuration to a target configuration? The problem is that every swapping of the cars must be via the empty slot. For example, if src is $[1, 3, 0, 2, 4]$, then to move a car 1 to the position of 3, one must go through the 0'th slot :

$$[1, 3, 0, 2, 4] \rightarrow [1, 0, 3, 2, 4] \rightarrow [0, 1, 3, 2, 4]$$

The code is as follows:

```
1   def park(i,src,tgt, map_src,map_tgt){
2      pos_fut = map_tgt[i]
3      j = src[pos_fut]
4      if ( j == 0 ) return 0
5      // move j out to 0 position
6      pos_0 = map_src[0]
7      src[pos_fut] = 0
8      src[pos_0] = j
9      // store new address
10     map_src[j] = pos_0
11     println(str(src))
12     // now move i to that position
13     pos_i = map_src[i]
14     src[pos_i] = 0
15     src[pos_fut] = i
16     map_src[i] = pos_fut
17     map_src[0] = pos_i
18     println(str(src))
19     j
20  }
21  def transform(src,tgt){
22     println(str(src))
23     map_src = dict(src) as { [$.o , $.i ] }
24     map_tgt = dict(tgt) as { [$.o , $.i ] }
25     visited = set()
26     i = 1
27     while( i !@ visited  ){
28        visited += i
29        i = park(i,src,tgt,map_src,map_tgt )
30     }
31     // out of here, swap the 0 and the last no now
32     pos_0 = map_src[0]
33     next = tgt[pos_0]
34     next_pos = map_src[next]
35     src[next_pos] = 0
36     src[pos_0] = next
37     println(str(src))
38     println(str(tgt))
39  }
40  src = [ 1, 3, 0, 2, 4 ]
41  tgt = [ 3, 2, 4, 1, 0]
42  transform(src,tgt)
```

11.3.20 *Interleaving of Strings*

Given two strings, print all the inter leavings of the two strings. Interleaving means that the if B comes after A , it should also come after A in the interleaved string. As an example, with *AB* and *CD* we have : [*ABCD, ACBD, ACDB, CABD, CADB, CDAB*]. This has application : DNA cross and sequencing. The solution is clearly non optimal - and must use grammar and language generators to generate the specific strings in question, but the solution displays the awesomeness of ZoomBA, so it is presented here.

```
1   def in_order(s,arr){
2       len = #|s|
3       j = lfold(arr, 0 ){
4           break( $.p == len )
5           continue( s[$.p] != $.c[ $.i ] )
6           $.p +=1
7       }
8       (j == len)
9   }
10  s1 = "AB" ; s2 = "CD"
11  tot = s1 + s2
12  arr = tot.toCharArray()
13  arg = list{ arr }( [0 : #|arr| ] )
14  results = join ( @ARGS = arg ) where {
15      ( in_order(s1,$.o) && in_order(s2,$.o)
16  } as { str($.o,'')}
17  println(results)
```

A better strategy would be thinking about what interleaving would mean, and that brings to the newer problem of finding all possible *partitions of a string*. What is a partition of a string? Given a string S, a partition of size k is a k-tuple of substrings $< s_1, s_2, ..., s_k >$ such that : $S = s_1 s_2 ... s_k$, and none of the s_i is empty string. Thus, given a string *aba*, possible partitions would be : $[a, b, a], [ab, a], [a, ba]$. This is very close the *Ramanujan Partition Problem*, and the code to solve it shows why :

```
1   def partition_string(s,bi){
2       len = #|s|
3       def seed : { start : 0, l : list() }
4       seed = lfold ([1:len] , seed ) as {
5           if ( bi[$.o -1]  == '1' ){
6               $.p.l += s[start : $.o-1]
7               $.p.start = $.o
8           }
9           $.p
10      }
11      l = seed.l
12      l += s[start:len-1]
13  }
14  def generate_partitions(s){
15      N = #|s|
16      n = 2 ** (N - 1)
17      p = lfold ([1:n] , list() ) as {
18          bi = str($.o, 2)
19          bi = '0' ** ( N - #|bi|  - 1 ) + bi
20          $.p.add( partition_string(s,bi) )
21          $.p
22      }
23  }
```

The function *generate_partitions* generates partitions for a string. Now, how this problem is related to interleaving problem? Start with a partition p_1 from a string S_1. This partition suppose has n_1 substrings. Thus, there are n_1 gaps available to *interleave* partitions from the second string S_2.

Start with an example : given string *abc* and partition $[a, b, c]$, and any other string to interleave, we can choose to exploit the inner positions shown as $*$ as follows:

$$*a * b * c*$$

Thus, the interleaving partitions required can be of sizes $n_1, n_1 + 1, n_1 - 1$. Thus, interleaving between partition p_1 of size n_1 of string S_1, and p_2 of size n_2 of string S_2 is only possible if the partition sizes are not differing by more than 1, hence yielding the condition :

$$|n_1 - n_2| < 2$$

and this, is the join condition! After this, what we need to do? We know, if the lengths differ, then we need to pick the smaller, and fit that into the larger partition. We will call this function *insert(large,small)*. Now, if the sizes are the same, there are two different partitions from strings "abc" and "ABC" :

$$AaBbCc \text{ and } aAbBcC$$

and we would call it a function *place(left,right)*. Thus, how these functions look?

```
1  def insert(large, small){
2     lfold([1:#|large|], large[0]) as {
3       $.p += small[ $.o - 1 ] + large[$.o]
4     }
5  }
6  def place(left, right){
7     lfold ([1:#|right|] , left[0] ) as {
8         $.p += right[ $.o - 1 ] + left[$.o]
9     } + right[-1]
10 }
```

They should not really be functions, but then it introduces the awesome notion called readability. That is not entirely fad, there are meaning to the madness. Observe now, thanks to all these modularisation of the source code, the final, interleaving code is now a piece of cake, or rather very close to what is one:

```
1  def interleave(s1,s2){
2     partition_1 = generate_partitions(s1)
3     partition_2 = generate_partitions(s2)
4     join (partition_1,partition_2) where {
5        #(p1,p2) = $.o // assign
6        len1 = #|p1| ; len2 = #|p2|
7        continue( #|len1 - len2 | > 1 )
8        if ( len1 < len2 ){
9            println ( insert(len2, len1 ) )
10       } else if ( len2 < len1 ){
11          println ( insert(len1, len2 ) )
12       }else {
13          println ( place(p1,p2 ) )
14          println ( place(p2,p1 ) )
15       }
16       false  // do not add them at all!
17    }
18 }
```

and thus, we end this section.

JAVA CONNECTIVITY

JVM is the powering force behind ZoomBA. This chapter illustrates various usage scenarios where ZoomBA can be embedded inside a plain Java Program.

12.1 HOW TO EMBED

12.1.1 *Dependencies*

In the dependency section (latest release is 0.2) :

```xml
<dependency>
  <groupId>org.zoomba-lang</groupId>
  <artifactId>zoomba.lang.core</artifactId>
    <version>0.3-SNAPSHOT</version>
</dependency>
```

That should immediately make your project a ZoomBA supported one.

12.1.2 *Programming Model*

ZoomBA has a random access memory model. The script interpreter is generally single threaded, but each thread is given its own engine to avoid the interpreter problem of GIL.

The memory comes in two varieties, default registers, which are used for global variables. There is also this purely abstract ZContext type, which abstracts how local storage gets used. This design makes pretty neat to integrate with any other JVM language, specifically Java.

Here is an example embedding.

12.1.3 *Example Embedding*

One sample embedding is furnished here :

```java
/* How to use ZoomBA Natively, i.e. not using JSR-232 */
import zoomba.lang.core.interpreter.ZContext;
import zoomba.lang.core.interpreter.ZContext.*;
import zoomba.lang.core.interpreter.ZInterpret;
import zoomba.lang.core.interpreter.ZScript;
import zoomba.lang.core.operations.Function;
import zoomba.lang.core.types.ZException;
import zoomba.lang.core.types.ZTypes;
//... can be more...
public static void dance(String[] args) {
   String file = args[0];
   ZScript zs = null;
   try {
        zs = new ZScript(file, null);
   }catch (Throwable t){
       if ( t.getCause() instanceof ZException.Parsing ){
           ZException.Parsing p = (ZException.Parsing)t.getCause();
           System.err.printf("Parse Error: %s\n", p.errorMessage );
           System.err.printf("Options were : %s\n", ZTypes.string(p.
               correctionOptions) );
       }else{
           System.err.printf("Error in parsing due to : %s\n", t.getCause() );
       }
       System.exit(2);
   }
   Object[] scripArgs = ZTypes.shiftArgsLeft(args);
   Function.MonadicContainer mc = zs.execute(scripArgs);
   if (mc.isNil()) {
     // handle when return is ... Nil
   } else {
     if (mc.value() instanceof Throwable) {
         Throwable underlying = (Throwable) mc.value();
       if ( underlying instanceof ZException){
           if ( underlying instanceof ZException.ZTerminateException ){
               System.exit(3);
           }
           System.err.println(underlying);
           zs.consumeTrace(System.err::println);
           System.exit(4);
       }
       if ( underlying instanceof StackOverflowError ){
           zs.consumeTrace(System.err::println);
           System.err.println("---> Resulted in 'Stack OverFlow' Error!");
           System.exit(4);
       }
       underlying.printStackTrace();
       System.exit(1);
     }
   }
   System.exit(0);
}
```

12.2   PROGRAMMING ZOOMBA

Because it is JSR-223 compliant you can do this :

```
public void javaxScripting() throws Exception {
    ScriptEngineManager scriptEngineManager = new ScriptEngineManager();
    ScriptEngine engine = scriptEngineManager.getEngineByName("zoomba");
    assertNotNull(engine);
    Object result = engine.eval("2+2");
    assertEquals(4,result);
    Bindings bindings = engine.createBindings();
    bindings.put("x", 2 );
    bindings.put("y", 2 );
    result = engine.eval("x+y", bindings);
    assertEquals(4,result);
    ScriptContext scriptContext = engine.getContext();
    scriptContext.setAttribute("x", 2, ScriptContext.ENGINE_SCOPE );
    scriptContext.setAttribute("y", 2, ScriptContext.GLOBAL_SCOPE );
    result = engine.eval("x+y");
    assertEquals(4,result);
    CompiledScript compiledScript = ((Compilable)engine).compile("x+y");
    assertEquals( engine, compiledScript.getEngine());
    result = compiledScript.eval(bindings);
    assertEquals(4,result);
    // function invoke ?
    engine.eval("def add(a,b){ a + b }",scriptContext);
    result = ((Invocable)engine).invokeFunction("add", 2, 2);
    assertEquals(4,result);
    // method invoke ?
    Object zo = engine.eval(new StringReader("def ZO{ def add(a,b){ a + b } } ;
        new(ZO) ") ,scriptContext);
    assertNotNull(zo);
    assertTrue(zo instanceof ZObject);
    result = ((Invocable)engine).invokeMethod(zo,"add", 2, 2);
    assertEquals(4,result);
    // now attributes stuff
    assertEquals( 2, scriptContext.getScopes().size());
    assertNotNull(scriptContext.getAttribute("x"));
    final int scope = scriptContext.getAttributesScope("x");
    assertEquals(ScriptContext.ENGINE_SCOPE , scope);
    assertNotNull(scriptContext.getAttribute("y", ScriptContext.GLOBAL_SCOPE));
    scriptContext.removeAttribute("y", ScriptContext.GLOBAL_SCOPE);
    assertNull(scriptContext.getAttribute("y", ScriptContext.GLOBAL_SCOPE));
    assertNull(scriptContext.getAttribute("y"));
    assertEquals(-1, scriptContext.getAttributesScope("y"));
    // now let's eval a script... from external file
    zo = engine.eval("@samples/test/date.zm",scriptContext);
    assertNotNull(zo);
    compiledScript = ((Compilable)engine).compile("@samples/test/date.zm");
    assertNotNull(compiledScript);
    engine.put("y",10);
    assertEquals(10,engine.get("y"));
}
```

12.3 EXTENDING ZOOMBA

There are two extension points for Java developers who are going to make their classes and functions ZoomBA ready.

12.3.1 *Implementing Named Arguments*

If one wants to implement a method which would mix properly with ZoomBA named arguments, then one must accept a Java Map, whose keys are the name of the parameters while values are the values passed.

Thus, once we know that the argument is Map, we can extract the value, and do the due diligence. Here is example from language itself:

```
/* Say the call is : mset(col=l,sorted=true) */
static Map accumulatorMap(NamedArgs na){
    if ( Boolean.TRUE.equals( na.getOrDefault("sort", false ) ) ){
        return new TreeMap<>();
    }
    if ( Boolean.TRUE.equals( na.getOrDefault("order", false ) ) ){
        return  new LinkedHashMap<>();
    }
    Object o = na.get("acc");
    if ( o != null ) {
        if (o instanceof Map<?, ?>) {
            return (Map) o;
        }
        throw new IllegalArgumentException("'acc' is not a map!" );
    }
    return new HashMap<>();
}
```

12.3.2 *JVM Functional Types*

If one wants to implement the anonymous blocks like almost all the ZoomBA functions do, one needs to expect the last few argument to be *FunctionalInterface*:

```
1  import java.util.ArrayList as AL
2  al = new ( AL, [ 1,2,3,4 ] )
3  al.stream.forEach( as { println($.o) }  )
```

As we can see this is a perfect way to incorporate anonymous args. However we can do better:

```
1  import java.util.ArrayList as AL
2  al = new ( AL, [ 1,2,3,4 ] )
3  al.stream.forEach() as { println($.o)  }
```

As we can see, the system automatically recognizes the last few parameters to be functional interfaces and automatically adjusts. However to grant reflective access to the *stream()* we need to run it with the command line switch:

```
--add-opens java.base/jdk.internal.loader=ALL-UNNAMED  \
--add-opens java.base/java.util.stream=ALL-UNNAMED
```

12.3.3 *ZoomBA Plugins*

12.3.3.1 Basics of Syntax

A ZoomBA plugin is characterized by # followed by a function style call, but all parameters passed are named. As you would now recall there are some default plugins already built in that you have encountered:

```
1  #(t,o)=#clock{/* execution time*/} // time taken, output
2  #atomic{  /* atomic block with automatic revert mechanism */  }
```

There are some other plugins which we would discuss here. But before that, we would discuss about how said plugins can be written.

12.3.3.2 Plugin Interface

A plugin must conform to a particular interface DSLPlugin.

```
/**
     * Process the plugin
     * @param interpret underlying interpreter
     * @param config configuration of the plugin
     * @param block actual plugin code block
     * @param data in case of any data needs to be passed
     * @return result of plugin execution
     * @throws Exception in case of any error
     */
    Object process(ZInterpret interpret,
                 Map<String, Object> config,
                 ASTBlock block,
                 Object data) throws Exception;


    /**
     * Plugin Registry
     * Simple idea, add name and Plugin
     */
    Map<String, DSLPlugin> REGISTRY = new LinkedHashMap<>();
```

We need to do 2 things.
- implement this interface.
- Register the class by adding an entry in *REGISTRY*.

In fact here is the snippet from the same interface *registering* some default *Plugins*:

```
/**
  * Bunch of Default plugins to register
  */
 static void defaultRegistration() {
       REGISTRY.put("atomic", ATOMIC); // Atomic operations
       REGISTRY.put("clock", CLOCK); // perf tuning
       REGISTRY.put("def", OBSERVABLE); // reactive programming with zmb
       REGISTRY.put("retry", RETRY); // retrying a block of code
       REGISTRY.put("timeout", TIMEOUT); // timing out a block of code
    }
```

Now how to actually write some plugin code? Here is the *Retry* plugin along with the associated Retry code:

```
/**
  * The retry construct
  */
DSLPlugin RETRY = (interpret, config, block, data) -> {
       Object handler = config.get(Retry.HANDLER);
       // special casing, in case it is defined and required
       if ( handler instanceof ZScriptMethod ){
           ZScriptMethod.ZScriptMethodInstance instance =
               ((ZScriptMethod) handler).instance(interpret);
           config.put(Retry.HANDLER, instance);
       }
       Retry retry = Retry.fromConfig(config);
       Callable<Object> callable = retry.callable( () -> block.jjtAccept(
           interpret, data) );
       return callable.call();
    };
```

Next we are going to describe various default plugins.

12.3.4 Atomic

This plugin creates a local sandbox within which you can revert back any changes done on local variables which are not references. This way you can simulate the *synchronized* block, along with a primitive rollback mechanism.

Observe the issue without any synchronized mechanism:

```
1  def modify(){ $var += 1 }
2  $var = 0
3  tl = list ( [0:10 ] ) as { thread() as modify }
4  fold ( tl ) as { $.o.join() }
5  println( $var ) // max is 10 can be of any value
```

But the moment we do this:

```
1  def modify(){ #atomic{ $var += 1 } }
2  $var = 0
3  tl = list ( [0:10 ] ) as { thread() as modify }
4  fold ( tl ) as { $.o.join() }
5  println( $var ) // always 10
```

12.3.5 Clock

A typical use case is as coded in performance test folder:

```
1  r = #clock{
2    for ( [0:1000000] ){
3      i = $
4    }
5  }
6  return r.0
```

And this explains the usage of *#clock* block.

12.3.6 Reactive

This starts with lazy evaluation. To evaluate any expression lazily, again and again we use the construct:

```
1  lz = #def{ x + y } // creates an Observable
2  x = 12
3  y = 30
4  lz.value //  42
5  lz + 10 //  52
6  lz - 10 // 32
7  10 + lz // error : can not do it
8  println(lz) // it is 42
```

Now an *Observable* is actually a *Supplier* of values. But in our specific case, it is a supplier of new values if the expression value changed. It has the same effect as the closure:

```
1  lz = def(){ x + y } // creates a non closed supplier
2  x = 12
3  y = 30
4  println(lz()) // it is 42
```

That brings the question of what is real difference between these two then? From a theory standpoint, not much. What can be done with an Observable, can be done with a non closed function. But in practice it gives us convenience.

Any time the value gets updated, there can be an event handler handling it for us:

```
1  def update_handler( previous,current, object ){
2    printf("[method] Observable updated --> %s --> %s %n", previous, current )
3  }
4  x = 20
5  y = 42
6  lz = #def{ x + y }
7  lz.update( update_handler )
8  lz.update( ) as {
9    printf("[lambda] Observable updated --> %s --> %s %n", $.p,  $.o )
10   }
11 println(lz) // event handlers get called because lz gets evaluated
```

12.3.6.1 Lazy Evaluation of Assertion Arguments

Observe this panic assert:

```
1  panic( x != null ,  'x is not null and has length : ' + x.length )
```

What happens? There are two cases.

- x is not null. Then panic fires and the message string evaluates right.
- x is null. Now panic should not fire , but what happens to the message? Message needs to still evaluate or not? For regular programming languages, it would. And that would crash the panic because x is null! Now for ZoomBA it does not.

This is the lazy evaluation for assertion class of functions. Unless the parameter needs to be evaluated, the parameters does not get evaluated, because for assertion class of functions, every argument is wrapped around inside an Observable by design. In fact this is the primary reason ZoomBA has observable.

12.3.6.2 As Foundation of Reactive Paradigm

Observe this chain:

```
1  db_data = #def{ /* get data from db */ }
2  // this is reactive paradigm
3  user_change.update( ) as { user.data = db_data }
4  user_change = #def{ /* do something about user changes */ }
5  def login(user_name){
6    /* new user came in */
7    user = user_change
8  }
9  // finally
10 login("zoomba")
```

What happens is a chain of evaluation.

- login triggers *user_change*
- On update of the Observable, *db_data* gets assigned to user data
- That triggers evaluation of *db_data* itself

And this chain can potentially go to any length. This is the crux of reactive paradigm.

12.3.7 *Retry*

One of the crucial component of resiliency is able to retry. Consider connection to a website to download an image. And it might fail. So we need to try again. This is actually a pattern for the design pattern seekers.

The question is, is there a clean way to solve this problem so that code is minimal. Turns out, we can. Here is the code snippet to do so:

```
1  RETRY_CONFIG = { "strategy" : "exp",
2                   "max" : 1,
3                   "interval" : 750 + random(1000)
4                 }
5  #(o ? e ) = #retry (RETRY_CONFIG) {  download_image( image_url) }
6  !empty(e) && eprintln("Failed : " + image_url )
```

This shows the key ingredient of *retry* plugin. First there is a configuration that we pass to configure the behavior. Then it is just a matter of simple method call on the block itself like clock or atomic.

12.3.7.1 Strategies

There are 3 strategies in place:

- Counter : *counter-* just till count reaches *max* value with fixed *interval* in ms.
- Random : *random* - counter, but the interval is randomized. The *interval* passed is the average.
- Exponential Back off : *exp* - counter, but the interval increases exponentially. *interval* is the first item

Counter is the default strategy.

12.3.7.2 As a Decorator Pattern

As told, under the hood it is a decorator pattern. This code here decorates any java.util.function.Function to create another *Function* but with retry enabled.

In fact this is how one can use the retry directly from Java itself:

```
Function<Integer,Integer> errFunc = (x) -> { throw new RuntimeException(); } ;
Map config = Map.of( "debug", true , "interval", 10000 , "max", 2 );
Retry retry = Retry.fromConfig( config );
Function<Integer,Integer> withRetry = retry.function(errFunc);
```

The same is actually applicable for *Timeout*.

12.3.8 *Timeout*

For ensuring users do get into a response rather than being *hung,stuck* in a computation loop, *timeout* of any operation is needed. The trouble is, of course - how to timeout an operation? There are two different ways to get this thing.

First of them is to use a separate thread to run the computation and then join the current thread so there is literally a block until the computation thread either times out or produces output, either in terms of result or an error.

The other one is a bit more deranged and is required collaborative multitasking - and implementing it right is real hard.

In ZoomBA we support both. Here is the use of Timeout decorator:

```
Timeout to = Timeout.fromConfig( Map.of( "interval", 10L, "inline", false ) );
 // regular usage
 final Function<Integer,Integer> t_id = to.function( id );
```

12.3.8.1   Preemptive Timeout : Different Thread

The configuration *inline* defines the timeout strategy, *false* implies we would be using a different thread:

```
1  def long_func(n){ /* some really long function based on n */ }
2  #(r ? e) = #timeout(interval=1000){ long_func(10) }
```

As one can see, the default is *inline : false*. The *interval* is the amount of time in millisecond to wait before the operation gets timed out. We must understand that the plugin block gets executed in a different thread, so being a Pure Function helps. We can use timeout like this with regular methods, but we should be careful about being the function pure.

12.3.8.2   Collaborative Timeout : Same Thread

This obviously does the inline or timeout in the same thread, however, there is a catch.

```
1  def long_func(n){ /* some really long function based on n */ }
2  #(r ? e) = #timeout(interval=1000, inline=true){ long_func(10) }
```

The catch is, how does one interrupts the main thread? This pose a problem in any CLR language.

The solution is a bit tedious in case of ZoomBA - we need to write methods which are in some sense *interruptible* :

```
1   def long_func(times, inline=false){
2     for( [0:times] ){
3       for ( [0:times] ){
4         x = $
5         // note this -- crucial
6         panic( inline && thread().interrupted(), "must be co-operative!" )
7       }
8     }
9     42 // return
10  }
```

What we are doing is forcing the current thread to panic on interrupt to relinquish control. This is the key to how to do Collaborative Scheduling - or Co-Routines.

Hence the crux is to essentially *catch interrupt* whenever we think it should be good enough.

Bibliography

[1] Donald E. Knuth, *The Art of Computer Programming* . Addison-Wesley, 2011.

[2] Alfred V. Aho, Monica S. Lam, Ravi Shethi , Jeffery D. Ullman, *Compilers: Principles, Techniques, and Tools* . Prentice Hall, 2006.

[3] Kanglin Li, Mengqi Wu, Sybex, *Effective Software Test Automation: Developing an Automated Software Testing Tool* . Sybex , 2004.

[4] Guido van Rossum, Fred L. Drake, *Python Language Reference Manual* PythonLabs 2003.

[5] Martin Odersky, *The Scala Language Specification Version 2.9* PROGRAMMING METHODS LABORATORY EPFL, SWITZERLAND, 2014.

Index