The nJexl Programming Language

An Extensible and Embeddable JVM Language
for Business Development
&
Software Testing
Version 0.3+

nJexl

Nabarun Mondal

Contents

Contents i

Preface vii
- A brief History vii
- An Open Challenge From Java Land vii
- About The Language vii
- The Philosophy of the Logo viii
- Thanks viii

1 On Software and Business Development 1
- 1.1 Business Development 1
  - 1.1.1 Needs of the Business 1
  - 1.1.2 Imperative Implementation 1
- 1.2 Functional Approach 2
  - 1.2.1 Tenets of Functional Style 2
  - 1.2.2 Applying Functional Style 3
- 1.3 Testing Business Software Today 4
  - 1.3.1 Issues with QA Today 4
  - 1.3.2 Testing Sorting 5
  - 1.3.3 Formalism 5

2 About nJexl 9
- 2.1 nJexl Philosophy 9
- 2.2 Design Features 10
  - 2.2.1 nJexl is Embeddable 10
  - 2.2.2 Interpreted by JVM 10
  - 2.2.3 nJexl can Execute any Java Code 10
  - 2.2.4 nJexl can be used Functionally 10
  - 2.2.5 nJexl is Dynamically Typed 10
  - 2.2.6 nJexl Vs Java 10
- 2.3 Setting up Environment 11
  - 2.3.1 Installing Java Run Time 11
  - 2.3.2 Download nJexl one jar 11
  - 2.3.3 Add to Path 11
  - 2.3.4 Test Setup 12
  - 2.3.5 Maven Setup for Java Integration 12
  - 2.3.6 Setting up Editors 12
  - 2.3.7 The Proverbial “Hello, World” 14

3 nJexl Syntax in 3 Minutes 15
3.1 Building Blocks 15
3.1.1 Identifiers & Reserved Keywords 15
3.1.2 Assignments 15
3.1.3 Comments 16
3.1.4 Basic Types 16
3.1.5 Multiple Assignment 16
3.2 operators 16
3.2.1 Arithmetic 16
3.2.2 Logical 17
3.2.3 Comparison 17
3.2.4 Ternary 18
3.3 Conditions 18
3.3.1 If 18
3.3.2 Else 18
3.3.3 Else If 19
3.3.4 GoTo 19
3.4 Loops 19
3.4.1 While 19
3.4.2 For 19
3.5 Functions 20
3.5.1 Defining 20
3.5.2 Calling 20
3.5.3 Global Variables 20
3.6 Anonymous Function as a Function Parameter 21
3.6.1 Why it is needed? 21
3.6.2 Some Use Cases 22
3.7 Available Data Structures 23
3.7.1 Range 23
3.7.2 Array 23
3.7.3 List 23
3.7.4 Set 23
3.7.5 Dict 24
3.7.6 Mutability 24

4 Formal Constructs 25
4.1 Conditional Building Blocks 25
4.1.1 Formalism 25
4.1.2 Relation between “there exists” ∃ and “for all” ∀ 25
4.1.3 There Exist In Containers 26
4.1.4 Size of Containers : empty, size, cardinality 27
4.1.5 There Exist element with Condition : index, rindex 28
4.1.6 For All elements with Condition : select 29
4.1.7 Partition a collection on Condition : partition 29
4.1.8 Collecting value on Condition : where 29
4.1.9 Inversion of Logic 29
4.2 Operators from Set Algebra 30
4.2.1 Set Algebra 30
4.2.2 Set Operations on Collections 31
4.2.3 Collection Cross Product and Power 32
4.2.4 Collection Relation Comparisons 33

4.2.5 Mixing Collections 33
4.2.6 Collections as Tuples 34

5 Comprehensions on Collections 35
5.1 Using Anonymous Argument 35
5.1.1 Arrays and List 35
5.1.2 Set 35
5.1.3 Dict 36
5.2 Alternate Iteration Flow Constructs 36
5.2.1 Continue and Break 36
5.2.2 Continue 37
5.2.3 Break 37
5.2.4 Substitute for Select 38
5.2.5 Uses of Partial 39
5.3 Comprehensions using Multiple Collections : Join 40
5.3.1 Formalism 40
5.3.2 As Nested Loops 40
5.3.3 Join Function 40
5.3.4 Finding Permutations 41
5.3.5 Searching for a Tuple 41
5.3.6 Finding Combinations 41
5.3.7 Projection on Collections 41

6 Types and Conversions 43
6.1 Integer Family 43
6.1.1 Boolean 43
6.1.2 Short 44
6.1.3 Character 44
6.1.4 Integer 44
6.1.5 Long 45
6.1.6 BigInteger 45
6.2 Rational Numbers Family 45
6.2.1 Float 46
6.2.2 Double 46
6.2.3 BigDecimal 46
6.3 The Chrono Family 47
6.3.1 Date 47
6.3.2 Time 47
6.3.3 Instant 48
6.3.4 Comparison on Chronos 48
6.3.5 Arithmetic on Chronos 49
6.4 String : Using str 49
6.4.1 Null 49
6.4.2 Integer 50
6.4.3 Floating Point 50
6.4.4 Chrono 50
6.4.5 Collections 51
6.4.6 Generalised $toString()$ 51
6.5 Playing with Types : Reflection 51
6.5.1 type() function 51
6.5.2 The === Operator 52
6.5.3 The *isa* operator 52

6.5.4 inspect() function 53
6.5.5 Serialisation using dict() function 54
6.5.6 Field Access 55

7 Reusing Code 57
7.1 The Import Directive 57
7.1.1 Syntax 57
7.1.2 Examples 57
7.2 Using Existing Java Classes 58
7.2.1 Load Jar and Create Instance 58
7.2.2 Import Enum 58
7.2.3 Import Static Field 59
7.2.4 Import Inner Class or Enum 59
7.3 Using nJexl Scripts 59
7.3.1 Creating a Script 60
7.3.2 Relative Path 60
7.3.3 Calling Functions 60

8 Functional Style 63
8.1 Functions : In Depth 63
8.1.1 Function Types 63
8.1.2 Default Parameters 64
8.1.3 Named Arguments 64
8.1.4 Arbitrary Number of Arguments 64
8.1.5 Arguments Overwriting 65
8.1.6 Recursion 66
8.1.7 Closure 67
8.1.8 Partial Function 68
8.1.9 Functions as Parameters : Lambda 68
8.1.10 Composition of Functions 68
8.1.11 Operators for Composition 69
8.1.12 Eventing 70
8.2 Strings as Functions : Currying 71
8.2.1 Rationale 71
8.2.2 Minimum Description Length 72
8.2.3 Examples 72
8.2.4 Reflection 74
8.2.5 Referencing 74
8.3 Avoiding Conditions 75
8.3.1 Theory of the Equivalence Class 75
8.3.2 Dictionaries and Functions 76
8.3.3 An Application : FizzBuzz 76
8.4 Avoiding Iterations 77
8.4.1 Range Objects in Detail 77
8.4.2 Iterative Functions : On Objects 78
8.4.3 The Fold Functions 79

9 Input and Output 81
9.1 Reading 81
9.1.1 read() function 81
9.1.2 Reading from url : HTTP GET 81
9.1.3 Reading All Lines 82

9.2 Writing 82
9.2.1 write() function 82
9.2.2 Writing to url : HTTP POST 83
9.2.3 Sending to url : send() function 83
9.3 File Processing 84
9.3.1 lines() function 84
9.3.2 fopen() function 85
9.3.3 Reading 85
9.3.4 Writing 85
9.4 Working with JSON 85
9.4.1 What is JSON? 85
9.4.2 json() function 86
9.4.3 Accessing Fields 86
9.5 Working with XML 87
9.5.1 xml() function 87
9.5.2 Accessing Elements 89
9.5.3 XPATH Formulation 90
9.5.4 Xml to JSON 91
9.6 DataMatrix 91
9.6.1 matrix() function 91
9.6.2 Accessing Data 92
9.6.3 Tuple Formulation 92
9.6.4 Project and Select 92
9.6.5 The matrix() function 93
9.6.6 Keys 94
9.6.7 Aggregate 96
9.6.8 Diff Functions 97
9.7 Database Connectivity 98

10 Interacting with Environment 99
10.1 Process and Threads 99
10.1.1 system() function 99
10.1.2 thread() function 100
10.1.3 until() function 100
10.1.4 Atomic Operations 101
10.1.5 The clock Block 102
10.2 Handling of Errors 103
10.2.1 error() function 103
10.2.2 try() function 104
10.2.3 Multiple Assignment 104
10.2.4 Error Assignment 104
10.3 Order and Randomness 105
10.3.1 Lexical matching : tokens() 105
10.3.2 hash() function 106
10.3.3 Anonymous Comparators 106
10.3.4 Sort Functions 108
10.3.5 sqlmath() function 108
10.3.6 minmax() function 109
10.3.7 shuffle() function 109
10.3.8 The random() function 109

11 Java Connectivity 111

11.1 How to Embed 111
11.1.1 Dependencies 111
11.1.2 Programming Model 111
11.1.3 Example Embedding 112
11.1.4 Default Namespaces 112
11.1.5 Default Imports 113
11.2 Programming nJexl 113
11.2.1 Script Object Model 113
11.2.2 Function Model 113
11.3 Extending nJexl 114
11.3.1 Overloading Operators 114
11.3.2 Implementing Named Arguments 114
11.3.3 Using Anonymous Arguments 114

12 Examples with Interview Questions 117
12.1 Assorted Impractical Examples 117
12.1.1 A Game of Scramble 117
12.1.2 Find Anagrams of a String 118
12.1.3 Sublist Sum Problem 118
12.1.4 Sublist Predicate Problem 119
12.1.5 List Closeness Problem 119
12.2 Assorted Practical Examples 120
12.2.1 Generic Result Comparison 120
12.2.2 Verifying Filter Results 121

Bibliography 123

Index 124

Index 125

Preface

nJexl initially was a continuation of Apache Jexl project. That project was not active for 17 months, and I needed to use it, found so many dangling issues, so I was forced to fork and maintain it. The reasons for further modifications are to be found below.

A brief History

All I wanted is a language where I can write my test automation freely - i.e. using theories from testing. The standard book, and there is only one for formal software testing is that of Beizer.

There was no language available which lets me intermingle with Java POJOs and let me write my test automation (validation and verifications). Worse still - one can not write test automation freely using Java. As almost all of modern enterprise application are written using Java, it is impossible to avoid Java and write test automation : 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 idea of extending JEXL thus came into my mind : a language that has all the good stuffs from the vast Java libraries, but clearly not verbose enough.

After cloning JEXL - and modifying it real heavy - a public release in a public repository seemed a better approach. There, multiple people can look into it, rather than one lone ranger working from his den. And hence nJexl was born. The *n* stands for *Neo*,*New*, not Noga, which is, by the way, my nick name. The stylised form $nJexl$ has capital *J* because it is fed by JVM.

An Open Challenge From Java Land

This actually was an open challenge,
A *noted PhD* from Sun, read that essay, and had this to say:

"hmm, chuckle :) This guy has too much time on his hands ! he should be doing useful work, or inventing a new language to solve the problems. Its easy to throw stones - harder to actually roll up your sleeves and fix an issue or two, or write/create a whole new language, and then he should be prepared to take the same criticism from his peers the way he's dishing it out for others. Shame - I thought developers were constructive guys and girls looking to make the lives of future software guys and girls easier and more productive, not self enamouring pseudo-intellectual debaters, as an old manager of mine used to say in banking IT - 'do some work' !"

And I like to think that I just did. That too, while at home, in vacation time, and in night time (from 8 PM to 3 AM. - see the check-ins.) Alone. But the users are the best judge.

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 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.

It supports OOP. Albeit not recommended, as OOPs! tells you. clearly shows why. In case you want C++ i.e. multiple inheritance with full operator overloading, friend functions, etc. then this is for you anyways. Probably you would love it to the core.

Python is a brilliant language, and I shamelessly copied many, and many adages of Python here. The heavy use of __xxx__ literals, and the *me* directive, and *def* is out and out python.

The space and tab debate is very religious, and hence JEXL is " 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 *plug symbol* in the logo signifies what precisely it was meant to do, to be simply plugged-in to the already existing Java ecosystem. We are to enhance, not to replace. Not to enhance in a Spring way, but in more logical, simpler way. Anything spring can, nJexl can do.

The choice of colour is R, G, B showing the human way to unify colour sensation, to mean unification between prevailing Java land, and terse functional land. The idea is to do heavy computational stuff in Java itself, while the more complex business logic processing in external nJexl files. For software testing, tools can be written which uses nJexl core to simplify the need to be agile. The nJexl.testing project is a step in that direction. It also has it's own wiki page.

Thanks

And finally, all of these pages were typed using my trust worthy Mac Book Pro, at home 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, I love you. RIP. For Windows, the trusty TexWorks was used using MikeTex packages. Thanks to you too! In the end many a thanks to Gabriel Hjort Blindell - for the beautiful style file he created which can be found here. Gabriel, thanks a ton. Your comments on formatting this document were invaluable. The nJexl logo was created using freelogoservices. Keep up the amazing work.

CHAPTER 1

On Software and Business Development

Software is needed to do most of the business, and it is software development, that becomes sometime the bottleneck of the business development. Here, we discuss the goal of nJexl, that is, what is the very problems that it seeks to eradicate.

1.1 Business Development

Does business need software? No and Yes. Clearly how brilliantly the code was written does not matter when one owns a portal like hello curry. Software augments business, there is no point making software the business. There are many firms doing software as business, and it is a very high niche game. In reality, most of the businesses are not about software, it is about taking user pain away, and always - making end users happy.

Marketing brings you customers, Tech-folks retain them. – Anon

1.1.1 *Needs of the Business*

Start with hello curry, whose game is to distribute food to the client. Hence it has a database full of food items, which it would showcase in the portal. User would click whatever she wants to eat, and checks out. The items gets delivered in time. Users can see their orders in a page. Occasionally, some specific food items gets added as the discounted price. Together, these requirements are what drives business for hello curry.

1.1.2 *Imperative Implementation*

Certain things in the implementations are invariant, one needs a database to store the users and their settings, and the food items. The code to showcase what to display as items are also fixed in nature. The real business problem is that of showing the *discounted* one. Under normal circumstances, one would write this code :

```
for ( item : items ){
   if ( discounted(item) ){
      display_item_discounted(item)
   }else{
     display_item(item)
   }
}
```

This sophomore level code is indeed correct when looked at in a very naive way. But the whole new complexity of the business starts with this, and is a one way ticket to the land of more and more complexity and unmanageable code. To showcase, observe now, how one would have added 2 different discounted category? Of course, one needs to add *necessary* code to the snippet :

```
for ( item : items ){
   if ( discounted_type_1(item) ){
      display_item_discounted_type_1(item)
   } else if ( discounted_type_2(item) ){
      display_item_discounted_type_2(item)
   } else{
     display_item(item)
   }
}
```

Thus, we are looking at an abyss of complexity.

1.2 FUNCTIONAL APPROACH

Proponent of functional programming are shamans with pedigree and degree from the land of mathematics and logic, and they look down at the miserly labor class imperative programmers. Thus, there is a great divide between them and us, the imperatives. Hence, not a single functional programming paradigm was ever applied in any practical (read who earns actual dollars and helps actual business) way (perhaps that is an exaggeration, but mostly true). But that can be changed. I intend to change that. While academia should learn that money comes from aiding businesses, our businesses must learn that under the snobbish attitude of the functional paradigm, there are some viable lessons. Thus we try to define what is functional programming, in total lay-mans terms.

1.2.1 *Tenets of Functional Style*

1. Thou shall have functions as first class citizens and are allowed to pass function as variables (including return values) and assign them to variables
2. Thou shall avoid loops as much as possible
3. Thou shall avoid branching (if, else) as much as possible
4. Thou shall replace these previous two constructs with higher order constructs passed onto thou by the language maker
5. Thou shall try not to change system state, objects should be immutable

1.2.2 *Applying Functional Style*

Now, lets see how these *tenets* of functional style helps us out here.We note the basic problem is that of list processing. However, we need to *apply* some *transform* (rather call a function) for each element of the list. Which function to call should be found by inspecting the item in question.

Suppose, there is a function that takes an item, and returns a function which is appropriate one to transform the item :

```
transform_function = find_appropriate_transform( item )
```

And then, we can apply this function to the item :

```
transform_function( item )
```

Well, now we have removed the conditionals in question :

```
for ( item : items ){
   transform_function = find_appropriate_transform( item )
   transform_function( item )
}
```

Purist will argue that how the "find_appropriate_transform" would be implemented? Oh, under functional paradigm, that becomes as simple as a hash or a dictionary :

```
def find_appropriate_transform(item){
   return __transformation_functions__[item.
      transformation_function_name]
}
```

And all is good. Now comes the more interesting part, where we use *hocus focus* called a *fold function* from functional magic to replace the for loop (see chapter 8) :

```
lfold{
   transform_function = find_appropriate_transform($)
   transform_function($)
}(items)
```

And we have practically nothing there to write. This sort of description of the problem induces what is known as Minimum Description Length. This is closely linked with Chaitin Solomonoff Kolmogorov complexity. But one thing is certain - no more volumes of coding with hundreds of developers. And thus if one really wants productivity, and sincere not to measure productivity by activity, but rather impact, this is for you and your business. Hard coding of code constructs itself wanes away in this style.

This brings a bigger question, are not we hard coding the number of functions which we can call inside that for loop? Yes we are, and even that can also be taken away. A generic series of computation can be achieved using something called multiplication of functions, under which function chaining and composition, which would be discussed later (see chapter 8).

1.3 TESTING BUSINESS SOFTWARE TODAY

Agile methodologies and the distant cousins of it stormed the industry away. Some people took it hard, one example of such is this. Many went along with it. Agile-like methodologies without reasonable testing infrastructure exists as of now because of 3 very important interconnected reasons which changed the way industry looks at QA :

1. Very few actually *sales a product* with binaries which needs to be distributed to clients anymore
2. Current *Products* are mostly web-portals and services (SAAS)
3. Thus, cost of shipping modified, fixed code in case of any error found is reduced to zero, practically

Hence, the age old norm about how software testing was thought won't work, and is not working. Traditional QA teams are getting thinner, in fact, in many organisations QA is being done with. But not all the organisations can do this. Anyone who is still selling a service, and wants to keep a nice brand name, there must be a QA. The amount of such activity varies never the less. Thus, only one word comes to the foray to describe the tenet of QA discipline - *economy*.

1.3.1 *Issues with QA Today*

Tenets of today is : Churn code out, faster, and with less issues, so that the pro activity makes end user happy. That is dangerously close to the ideas of TDD, however very few org actually does TDD. Old concepts like CI also came in foray due to these tenets, but the lack of Unit Testing (yes, any condition coverage below 80% is apathetic to the customer if she is paying money, ask yourself if you would like to book ticket using an airline booking system which is less than 80% tested or even use untested MS-Excel), makes most of the software built that way practically mine fields.

10 years back in Microsoft when we were doing CI (MS had CI back then even, we used to call it SNAP : Shiny New Automation Process), the whole integration test was part of the test suite that used to certify a build. Breaking a build due to a bug in a commit was a crime. Gone are those times.

Thus, the economy of testing in a fast paced release cycle focuses almost entirely to the automation front, manual testing simply can't keep up with the pace. Given there are not many people allocated for testing (some firms actually makes user do their testing) - there is an upper cut-off on what all you can accommodate economically. Effectively the tenet becomes : get more done, with less.

But what really makes the matter worse is, all of the so called automation testing frameworks are bloated form of what are known as *application drivers*- mechanism to drive the application UI, in some cases calling web services, along with simple logging mechanism that can produce nice and fancy reports of test passing and failing. Those things sale, but does not do justice to the actual work they were ought to do : "verify/validate that the system actually works out". Software testers and Engineering managers must think *validation first* and not *driving the UI* or simply buzzwords like *automating*. Anyone proclaiming that "We automated 100 test cases" should be told to go back and start *testing scenarios* instead of test cases, individual scenarios include many test cases, while anyone boasting about test scenarios should be asked : "what are you validating on those scenarios and how are you validating them?". Management, rather bad management takes number of test cases done after development a (satisfactory) measurement of test quality, and that is very naive, not to say anti-agile. In any case, any mundane example on testing would show, there are actually infinite possibilities adding more and more test cases, unless, of course one starts thinking a bit formally. Thus the foundational principle of statistics is :

"increasing sample size beyond a point does not increase accuracy"

and Occams Principle suggests that :

"Entities must not be multiplied beyond necessity"

1.3.2 *Testing Sorting*

A very simple example can be used to illustrate this discrepancy between rapid test automation and correct test validation:

Given a function that sorts ascending a list of numbers,
test that given an input, the function indeed sorts it as described

Thus, given a function called *sort* how does one verify that it indeed sorted a list *l* ?

Let's start with what the immediate response of people, when they get asked this. They often generate a simplistic and wrong solution : verify that the resulting list coming out of the *sort* method is indeed *sorted in ascending order*, that implementation would be indeed easy :

```
def is_sorted( l ){
   s = size(l)
   if ( s <= 1 ) return true
   for ( i = 1 ; i < s ; i += 1 ){
       if ( l[i-1] > l[i] ) return false
   }
   return true
}
```

But it would fail the test given the *sort* method used to generate *il* produces more or less number of elements than the original list *l*. What more, it would also fail when the *sort* method is generating same elements again and again.

This shows that we need to understand what the validation code should be, rather than describing the problem of sorting in plain English. Use of English as a language for contract is detrimental to the discipline of engineering, as we have seen just now. Thus, we ask : *What precisely sorting means?*

1.3.3 *Formalism*

To understand sorting, we need to take two steps. First of them is to define what order is, and the next is to define what is known as permutation. First one is pretty intuitive, second one is a bit troublesome. That is because permutation is defined over a set, not on a list, which are in fact multi-set if not taken as a tuple. Thus, it is easy to declare permutations of $(0,1)$ as : $(0,1)$ and $(1,0)$; but it became problematic when we have repetition : (a,a,b,c). This problem can be solved using permutation index.The current problem, however, can be succinctly summarised by :

Given a list l, the sorted permutation of it is defined as
a permutation of l such that the elements in the resulting list are in sorted order.

As we have noticed the solution presented before missed the first part completely, that is, it does not check that if the resultant list is indeed a permutation of the original or not.

To fix that we need to create a function called *is_permutation*, and then the solution of the problem would be :

```
sorting_works =  is_permutation(l,ol) and is_sorted(ol)
```

The trick is to implement is_permutation (l := list, ol:= output list):

```
def is_permutation(l,ol){
   d = dict()
   for ( i : l ) {
      // keep counter for each item
      if ( not (  i @ d ) ) d[i] = 0
      d[i] += 1
   }
   for ( i : ol ) {
      if ( not (  i @ d ) ) return false
      // decrement counter for each found item
      d[i] -= 1
   }
   for ( p : d ){
      // given some items count mismatched
      if ( p.value != 0 ) return false
   }
   // now, everything matched
   return true
}
```

The "a @ b " defines *if a is in b or not*. But that is too much code never the less, and clearly validating that code would be a problem. Hence, in any suitable declarative language optimised for testing such stuff should have come in by default, and tester ought to write validation code as this :

```
def is_sorted_permutation(l,ol){
    return ( l == ol and // '==' defines permutation for a list
    /*  _ defines the current item index
        $ is the current item
        index function returns the index of occurrence of
           condition
        which was specified in the braces -->{ }
    */
    index{ _ > 0 and  $[_-1] > $ }(l) < 0 )
}
```

And that is as declarative as one can get. Order in a collection matters not, all permutations of a collection is indeed the same collection, i.e. lists [1,2,3] is same as [2,1,3] which is same as [3,1,2]. For the *index*() function, we are letting the system know that anywhere I see previous item was larger than the current, return the index of this occurrence. Such occurrence would fail the test. Not finding such an occurrence would return -1, and the test would pass.

However, one may argue that in the solution we have hard coded the notion of *sorting order* . The above code basically suggests with " $$[_ -1] > $ " that to pass the test :

previous item on the list must be always less than or equal to the current

That is indeed hard coding the order *ascending*, but then, this can easily be fixed by passing the operator ">" along with the other parameters for validation function, using some notion called higher order functions discussed later. All these pseudo-code like samples are in fact - proper nJexl code, ready to be deployed. This also gives a head start on which direction we would be heading.

After 12 years of looking at the way software test automation works out, we figured out that imperative style is not suited at all for software validation. This observation was communicated to me 10 years ago by Sai Shankar:

"Software validation should be like SQL"

yes, he used to love talking in plain English. As we found, as the automation lead of the Office team, being imperative or OOP like would be to overtly complicate the test code. Thus, we, given a chance moved to declarative paradigm. This is a glimpse of how we write test automation and necessary validations. It is no wonder that we can produce formally correct automation and validation at a much faster rate, using much less workforce than the industry norm. There are really no fancy stuff here. Simple, grumpy, old Predicates. That is all the business world needs.

CHAPTER 2

About nJexl

A philosophy is needed to guide the design of any system. nJexl is not much different. Here we discuss the rationale behind the language, and showcase how it is distinct from its first cousin Java. While it is reasonably close 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, I believe there is no real general purpose language. All are special purpose, some special purposes are very generic in nature. *nJexl* however preaches a different philosophy than *scala*.

2.1 nJexl philosophy

To begin with, my 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, nJexl comes with it's tenets, which are :

1. Reduce the number of lines in the code;
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 (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.

That is,

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

Thus I made nJexl so that a language exists with it's full focus on *Business Process Automation and Software Testing & 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.

2.2 DESIGN FEATURES

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

2.2.1 *nJexl is Embeddable*

nJexl 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 nJexl scripts very easily, and all of nJexl 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 nJexl can be found in the test directory. Many scripts are there as samples in samples folder. In chapter 11 we would showcase how to embed nJexl in Java code.

2.2.2 *Interpreted by JVM*

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

2.2.3 *nJexl can Execute any Java Code*

nJexl enables you to use all the classes of the Java SDK's in nJexl, 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 nJexl script, and then loading a class as as trivial as importing the class.

2.2.4 *nJexl can be used Functionally*

nJexl 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.

nJexl 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.

2.2.5 *nJexl is Dynamically Typed*

nJexl, unlike other 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.

2.2.6 *nJexl Vs Java*

Most of the type one can treat nJexl as a very tiny shorthand for incredibly fast programming using JVM. However, nJexl 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 :

```
def parent_function( a ) {
    def child_function (b) {
        // child can use parents args: happily.
      a + b
    }
    if ( a != null ) {
      return child_function
   }
}
```

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

```
fp = parent_function(10)
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.

2.3 SETTING UP ENVIRONMENT

2.3.1 *Installing Java Run Time*

You need to install Java runtime 1.8 (64 bit). To test whether or not you have successfully installed it try this in your command prompt :

```
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

2.3.2 *Download nJexl one jar*

Download the latest one-jar.jar file from here.

2.3.3 *Add to Path*

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

```
alias njexl = "java -jar njexl.lang-0.3-<time-stamp>-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 njexl.bat
java -jar njexl.lang-0.3-<time-stamp>-onejar.jar  %1 %2 %3 %4 %5 %6
```

and then add path to this njexl.bat file in your path variable, see here.

2.3.4 *Test Setup*

Open a command prompt, and type :

```
$njexl
(njexl)
```

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

2.3.5 *Maven Setup for Java Integration*

In the dependency section (latest release is 0.2) :

```
<dependency>
  <groupId>com.github.nmondal</groupId>
  <artifactId>njexl.lang</artifactId>
  <version>0.3-ALPHA-1</version> <!-- or 0.3-SNAPSHOT -->
</dependency>
```

That should immediately make your project a nJexl supported one.

2.3.6 *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 jexl and a specially made theme for jexl 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 jexl script file looks like this :

To include for vim :

Create these two files :

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

For most nix systems it would be same as :

```
mkdir -p ~/.vim/ftdetect/
touch ~/.vim/ftdetect/jxl.vim
touch ~/.vim/syntax/jxl.vim
```

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

```
autocmd BufRead,BufNewFile *.jxl,*.jexl,*.njxl,*.njexl  set filetype=jxl
```

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/jxl.vim file as is.

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

```
/**
   Potential One liners
   which shows you the expressive power
   of nJexl
*/
import 'java.lang.System.out' as out

/* generate and multiply a list by n */

list{ $ * n }( [a:b].list())

/* find unique items in a list */

set ( list{ $ / n }( [a:b].list()) )


/*
Finds http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
Compare it to SCALA to see the difference on your own
*/
def soe( n ){
    select {
        x = $ // set the current iterate variable
        // _$_ is the partial result as of now , thus I am using partial result of select!
        where ( index{ x % $ == 0 }( _$_ + 2 ) < 0 ){ $ = x }
    }( [3:n+1].list() ) + 2  // adding 2 in the end list of primes
}

out:println( soe(31) )

```

Figure 2.1 – Using Sublime Text.

```
dummy.jexl
import 'java.lang.System.out' as out

def some_func(s){
    out:println(s)
    my:void_func()
    return true
}

def void_func(){
    return false
}
if ( #def __args__ ){
    // should print 42 here...
    write( str( __args__ ,'' ) )
}
some_func("Hello, World!")
~
~
~
~
<odes/java/njexl/src/lang/samples/dummy.jexl   CWD: /Codes/java/njexl/src/lang/samples   L
```

Figure 2.2 – Using Vim (MacVim).

2.3.7 *The Proverbial "Hello, World"*

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

```
write('Hello, World!')
```

and go to the command prompt, and run :

```
$ njexl hello.jxl
Hello, World
$
```

and that would be the proverbial starting code.

CHAPTER 3

nJexl Syntax in 3 Minutes

With some understanding on C,C++, Java, then it will be very easy for you to learn nJexl. The biggest syntactic difference between nJexl and other languages is that the ';' statement end character is optional. When we consider a nJexl program it can be defined as a collection of objects that communicate via invoking each others methods.

3.1 Building Blocks

3.1.1 *Identifiers & Reserved Keywords*

nJexl is case-sensitive, which means identifier Hello and hello would have different meaning. All nJexl 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, lt, eq, le, ge, gt, and, or, not, xor.

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

   atomic, xml, thread, system, lines, list, dict, array , type , write, read, send, random, hash , until , me, my , int, float, double, INT, DEC, NUM, char, bool, byte, str.

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

   @,$, anything with '__'.

3.1.2 *Assignments*

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

```
a = 1 //  assigns local variable a to Integer 1
b = 1.0 //assigns local variable a to b float 1.0
c = 'Hello, nJexl' // assigns local variable c to String 'Hello
   , nJexl'
d = "Hello, nJexl" ## same, strings are either single or double
    quoted
/*
assigns the *then* value of a to e,
subsequent change in a wont reflect in e
*/
e = a
```

3.1.3 *Comments*

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

3.1.4 *Basic Types*

Basic types are :

```
a = 1 // Integer
b = 1.0 //  float
c = 'Hello, nJexl' // String
d = 1.0d ## Double
I = 1h // BigInteger
D = 1.0b // BigDecimal
tt = true // boolean
tf = false // boolean
null_literal = null // special null type
```

3.1.5 *Multiple Assignment*

nJexl supports multiple assignment. It has various usage:

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

3.2 OPERATORS

3.2.1 *Arithmetic*

```
a = 1 + 1 // addition : a <- 2
z = 1 - 1 // subtraction : z <- 0
m = 2 * 3 // multiply : m <- 6
d = 3.0 / 2.0 // divide d <- 1.5
x = 2 ** 10 // Exponentiation x <- 1024
y = -x // negation, y <- -1024
r = 3 % 2 // modulo, r <- 1
r = 3 mod 2 // modulo, r <- 1
a += 1 // increment and assign
z -= 1 // decrement and assign
```

3.2.2 *Logical*

```
o = true or true // true , or operator
o = true || true // same
a = true and false // false , and operator
a = true && false // false, and operator
defined_a = #def a // true if a is defined, false otherwise
o = (10 != 20 ) // not, true
o = not ( 10 = 20 ) // true
o = 10 eq 20 // false
```

3.2.3 *Comparison*

```
t = 10 < 20 // true, less than
t = 10 lt  20 // true , less than
f = 10 > 20 // false, greater then
f = 10 gt 20 // false, greater then
t = 10 <= 10 // true, less than or equal to
t = 10 le  10 // true , less than or equal to
t = 10 >= 10 // true, greater then or equal to
t = 10 ge 10 // true, greater then or equal to
t = ( 10 == 10 ) // true, equal to
t = ( 10 eq 10 ) // true, equal to
f = ( 10 != 10 ) // false, not equal to
f = ( 10 ne 10 ) // false, not equal to
```

3.2.4 *Ternary*

```
// basic ternary
min = a < b ? a : b // general form (expression)?option1:option2
// try fixing null with it
non_null = a == null? b : a
// or use the null coalescing operator
non_null = a ?? b
// same can be used as definition coalescing
defined = #def(a) ? a : b
//same as above
defined = a ?? b
```

3.3 CONDITIONS

People coming from any other language would find them trivial.

3.3.1 *If*

```
x = 10
if ( x < 100 ){
   x = x**2
}
write(x) // writes back x to standard output : 100
```

3.3.2 *Else*

```
x = 1000
if ( x < 100 ){
   x = x**2
}else{
  x = x/10
}
write(x) // writes back x to standard output : 100
```

3.3.3 *Else If*

```
x = 100
if ( x < 10 ){
   x = x**2
} else if( x > 80  ){
  x = x/10
} else {
   x = x/100
}
write(x) // writes back x to standard output : 10
```

3.3.4 *GoTo*

```
/*
 In case anyone like to use GOTO
*/
i = 0
goto #label false // wont go there
goto #label (i == 0) // will go there
i = 2
write("This should be skipped")

#label
write("This should be printed")
goto #unconditional
#unconditional
return i
```

3.4 LOOPS

3.4.1 *While*

```
i = 0
while ( i < 42 ){
  write(i)
  i += 1
}
```

3.4.2 *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 :

```
for ( i : [0:42] ){ // [a:b] is a range type
  write(i)
}
```

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

```
for ( i = 0 ; i < 42 ; i+= 1  ){ // [a:b] is a range type
  write(i)
}
```

3.5 FUNCTIONS

3.5.1 *Defining*

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

```
def count_to_num(num){
   for ( i : [0:num] ){
      write(i)
   }
}
// just assign it
fp = count_to_num
```

One can obviously return a value from function :

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

3.5.2 *Calling*

Calling a function is trivial :

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

3.5.3 *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 write to it, use *var* keyword.

```
var a = 0
x = 0
def use_var(){
   a = 42
   write(a) // prints 42
   write(x) // prints 0, can access global
   x = 42
   write(x) // prints 42
}
// call the method
use_var()
write(a) // global a, prints 42
write(x) // local x, prints 0 still
```

The result would be :

```
42
0
42
42
0
```

3.6 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.

3.6.1 *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:

```
l = list()
  for ( x : [0:n] ){
      l.add ( x * x )
  }
return l
```

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

```
// return is not really required, last executed line is return
def map(x){ x * x }
l = list()
for ( x : [0:n] ){
      l.add ( map(x) )
}
return l
```

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

```
def map(x){ x * x }
def list_from_list(fp, old_list)
    l = list()
    for ( x : old_list ){
    // use the function *reference* which was passed
      l.add( fp(x) )
      }
      return l
}
list_from_list(map,[0:n]) // same as previous 2 implementations
```

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

```
list{ $*$ }([0:n])
```

The curious block construct after the list function 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. The explanation is as follows. For an anonymous function parameter, there are 3 implicit guaranteed arguments :

1. $ –> Signifies the item of the collection , we call it the *ITEM*
2. $$ –> The context, or the collection itself , we call it the *CONTEXT*
3. _ –> The index of the item in the collection, we call it the *ID* of iteration
4. Another case to case parameter is : _$_ –> Signifies the partial result of the processing , we call it *PARTIAL*

3.6.2 *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 :

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

But now, I 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 :

```
#(min,max) = minmax{
                size($.0) < size($.1)
                }( "" , "aa" , "abc" , "aa", "bbbbb" )
write(min) // prints empty string
write(max) // prints bbbbb
```

3.7 AVAILABLE DATA STRUCTURES

3.7.1 *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:

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

3.7.2 *Array*

A very simple way to generate inline array is this:

```
a1 = [0 , 1, 2, 3 ] // an integer array
a2 = [1 , 2.0 , 3, 4 ] // a number array
ao = [ 0 , 1, 'hi', 34.5 ] // an object array
AO = array ( 0,1,2,3 ) // an object array
```

Arrays are not modifiable, you can not add or remove items in an array. But, you can replace them :

```
a1[0] = 42 // now a1 : [ 42, 1, 2, 3 ]
```

3.7.3 *List*

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

```
l = list ( 0,1,2,3 ) // a list
l += 10 // now the list is : 0,1,2,3,10
l -= 0 // now the list is : 1,2,3,10
x = l[0] // x is 1 now
l[1] = 100 // now the list is : 1,100,3,10
```

3.7.4 *Set*

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

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

3.7.5 *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:

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

3.7.6 *Mutability*

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

```
x = [1,2,3]
x + 10 // add some
write(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 :

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

CHAPTER 4

Formal Constructs

Formalism is the last thing that stays in a software developers mind nowadays. This is a result of mismanaging expectation - software may be art, but is not science at all. Most of the cases, a bit of formal thinking solves a lot of the problems that can come. This section would be specifically to understand what sort of formalism we can use in practice.

4.1 Conditional building blocks

4.1.1 *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.

4.1.2 *Relation between "there exists" $\exists$ and "for all" $\forall$*

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 :

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

4.1.3 *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 @.

```
l = [1,2,3,4] // l is an array
in = 1 @ l // in is true
in = 10 @ l // in is false
d = { 'a' : 10 , 'b' : 20  }
in = 'a' @ d // in is true
in = 'c' @ d // in is false
in = 10 @ d // in is false
in = 10 @ d.values() // in is true
s = "hello"
in = "o" @ s // in is true
/* This works for linear collections even */
m = [2,3]
in = m @ l // true
in = l @ l // true
```

This was not simply put in the place simply because we dislike $x.contains(y)$, in fact we do. We dislike the form of object orientation where there is no guarantee that x would be null or not. Formally, then :

```
// equivalent function
def in_function(x,y){
   if ( empty(x) ) return false
   return x.contains(y)
}
// or one can use simply
y @ x // same result
```

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

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

Thus, the operator "=~" is the *match* operator, while "!~" is the *not match* operator.

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

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

```
n = null
e = empty(n) // true
n = []
e = empty(n) // true
n = list()
e = empty(n) // true
d = { : }
e = empty(d) // true
nn = [ null ]
e = empty(nn) // false
```

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

```
n = null
e = size(n) // -1
n = []
e = size(n) // 0
n = list()
e = size(n) // 0
d = { : }
e = size(d) // 0
nn = [ null ]
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 :

```
n = null
e = #|n| // 0
n = []
e = #|n| // 0
n = list()
e = #|n| // 0
d = { : }
e = #|d| // 0
nn = [ null ]
e = #|nn| // 1
```

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

4.1.5 *There Exist element with Condition : index, rindex*

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 :

```
l = [ 1, 2, 3, 4, 5, 6 ]
// search an element such that double of it is 6
i = index{ $ * 2 == 6 }(l) // i : 2
// search an element such that it is between 3 and 5
i = index{ $ < 5 and $ > 3 }(l) // i : 3
// search an element such that it is greater than 42
i = index{  $ > 42 }(l) // 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 none of the elements matches the true value, it returns -1. Index function runs from left to right, and there is a variation *rindex*() which runs from right to left.

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

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

4.1.6 *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.

```
l = [ 1, 2, 3, 4, 5, 6 ]
// select all even elements
evens = select{ $ % 2 == 0 }(l)
// select all odd elements
odds = select{ $ % 2 == 1 }(l)
```

4.1.7 *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 .

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

4.1.8 *Collecting value on Condition : where*

Given a *select*() or *partition*(), we are collecting the values already in the collection. What about we want to change the values? This is done by the conditional known as *where*() . The way where works is, we put the condition inside the where. The result would be the result of the condition, but the where clause body would be get executed. Thus, we can change the value we want to collect by replacing the $ITEM$ variable, that is \$.

```
l = [ 1, 2, 3, 4, 5, 6 ]
#(evens,odds) = partition{ where($ % 2 == 0){ $ = $**2  }  }(l)
write(evens) // prints 4, 16, 36
write(odds) // prints 1, 3, 5
```

This is also called *item rewriting* .

4.1.9 *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:

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

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

4.2 OPERATORS FROM SET ALGEBRA

4.2.1 *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 :

$$A \subset B \text{ when } A \subseteq B \text{ and } \exists x \in B \text{ s.t. } x \notin A$$

d. Superset Equals:

$$A \supseteq B \text{ when } B \subseteq A$$

e. Superset :

$$A \supset B \text{ when } B \subset A$$

4.2.2 *Set Operations on Collections*

For the sets, the operations are trivial.

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

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

```
d1 = {'a' : 10, 'b' : 20 , 'c' : 30 }
d2 = {'c' : 20 , 'd' : 40  }
// union is or : u = { 'a' : 10, 'b' : 20, 'c' : [ 30,20] , 'd'
    : 40  }
u = d1 | d2
i = d1 & d2 // intersection is and : i = { : }
m12 = d1 - d2 // m12 = d1
m21 = d2 - d1 // m12 = d2
delta = d1 ^ d2 // delta = {'a' : 10, 'b' : 20, 'd': 40}
```

4.2.3 *Collection Cross Product and Power*

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

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

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

and thus :

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

For general collection power can not be negative. Here are some examples now:

```
b = [0,1]
gate_2 = b*b // [ [0,0],[0,1],[1,0],[1,1] ]
another_gate_2 = b ** 2 // same as b*b
gate_3 = b ** 3 // well, all truth values for 3 input gate
gate_4 = b ** 4 // all truth values for 4 input gate
```

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.

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

4.2.4 *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:

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

So, we go again with lists:

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

And finally with dictionaries:

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

4.2.5 *Mixing Collections*

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

```
s = set(1,2,3)
l = list(1,3,3,2)
sub = s < l  // true
sup = l > s // true
u = l | s // u = [1,2,3,3 ]
```

4.2.6 *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 :

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

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

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

```
// read m starts_with l
sw = m #^ l // true
// read m ends with l
ew = m #$ l // true
```

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

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

CHAPTER 5

Comprehensions on Collections

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.

5.1 Using Anonymous Argument

5.1.1 *Arrays and List*

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

```
def map(item) {/* does some magic */}
def comprehension(function, items){
   c = collection()
   for ( item : items ){
     c_m = function(c)
     c.add( c_m )
   }
   return c
}
//or this way :
c = collection{ function($) }(items)
```

Obviously *list*() generates *list* and *array*() generates *array*. Hence, these are valid :

```
an = array{ $ + 2 }([1,2,3,4]) // [ 3,4,5,6 ]
ln = list{ $ ** 2 }([1,2,3,4]) // [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.

5.1.2 *Set*

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

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

5.1.3 *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:

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

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

5.2 ALTERNATE ITERATION FLOW CONSTRUCTS

5.2.1 *Continue and Break*

In normal iterative languages, there are continue and break.

The idea of continue would be as follows :

```
for ( i : items ){
    if ( condition(i) ){
      // execute some
      continue
    }
    // do something else
}
```

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 :

```
for ( i : items ){
    if ( condition(i) ){
    /* execute some code */
    break
    }
    // do something else here
}
```

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.

5.2.2 *Continue*

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

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

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 :

```
for ( n : integers ){
    continue( n % 15 == 0 ){ write('FizzBuzz') }
    continue( n % 3 == 0 ){ write('Fizz') }
    continue( n % 5 == 0 ){ write('Buzz') }
    write(n)
}
```

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

5.2.3 *Break*

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

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

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 :

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

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

Obviously, the block of the break is optional.

5.2.4 *Substitute for Select*

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

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

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

```
ls = select{ <condition> }(collection)
lc = list{ continue( not ( <condition> ) ) ; $ }(collection)
```

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

```
ls = select{ where ( <condition> ){ <body> } }(collection)
lc = list{ continue( not (<condition>) ) ; <body> }(collection)
```

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

```
l_e1 = select{ $ % 2 == 0 and $ < 5 }([0:10])
l_e2 = list{ break( $ > 5 ) ;
       continue($ % 2 != 0) ; $ }([0:10])
```

Note that Break body is inclusive:

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

So, with a body, the boundary value is included into *break*.

5.2.5 *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:

```
l = list{ continue( $ @ _$_ ) ; $ }( [1,1,2,3,1,3,4] )
/* 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:

```
def is_prime( n ){
    primes = set(2,3,5)
    for ( x : [6:n+1] ){
        x_is_prime = true
        for ( p : primes ) {
            break( x % p == 0  ){ x_is_prime = false }
        }
        if ( x_is_prime ){
          if ( x == n ) return true
          primes += x
        }
    }
    return false
}
```

Now, a declarative style coding :

```
def is_prime( n ){
    primes = set{
    // store the current number
        me = $
        // check : *me* is not prime - using partial set of
           primes
        not_prime_me = ( index{  me % $ == 0 }( _$_ ) >= 0 )
        // if not a prime, continue
        continue( not_prime_me )
        // collect me, if I am prime
        $
    }( [2:n+1] )
    // simply check if n belongs to primes
    return ( n @ primes )
}
```

Observe that, if we remove the comments, it is a one liner. It really is. Hence, declarative style is indeed succinct, and very powerful.

5.3 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.

5.3.1 *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 :

$$\{e \in S_1 \times S_2 \times ... \times S_n \mid P(e)\}$$

5.3.2 *As Nested Loops*

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

```
for ( a : A ){
   for ( b : B ) {
     write ( '(%s,%s)', a,b )
   }
}
```

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 essentially is :

```
collect = collection()
for ( a : A ){
   for ( b : B ) {
      tuple = [a,b]
      if ( condition ( tuple ) ) { collect += tuple  }
   }
}
```

5.3.3 *Join Function*

Observe that the free join, is really a full cross product, and is available using the power operator :

```
A = ['a','b']
B = [ 0, 1 ]
j1 = A * B // [ [a,0] , [a,1] ,[b,0], [b,1] ]
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 : ${}^3P_2$:

```
A = ['a','b', 'c' ]
// generate 2 permutations
p2 = join{  $.0 != $.1 } ( A , A )
```

5.3.4 *Finding 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 $\$.0! = \$.1$. It has to move beyond. So, for ${}^3P_3$ we have :

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

which is the declarative form for permutation, given all elements are unique.

5.3.5 *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:

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

which is the declarative form of the problem.

5.3.6 *Finding Combinations*

We did find permutation. What about combinations? What about we want to find combinations from 3 elements, taken 2 at a time? We observe that every combination is still a permutation. So, once it qualified as a permutation, we need to check that if the pair is in strictly increasing order or not (sorted). Thus :

```
A = [ 'a' ,'b', 'c' ]
c = join{ // check permutation , then
       #|set($)| == #|$| and
       // check sorted - same trick we used earlier too!
       index{ _ > 0 and  $$[_-1] > $ }($) < 0
    }(A,A)
```

This is how we can solve the problem with the aid of a declarative form.

5.3.7 *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). The idea is simple :

```
a = [0,1,2,3,4]
sub(a,2) // [2, 3, 4]
sub(a,2,3) // [2, 3]
sub(a,-1) // [0, 1, 2, 3]
sub(a,0,-1) // [0, 1, 2, 3]
sub(a,1,-1) // [1, 2, 3]
```

If you do not like the name $sub()$, the same function is aliased under $project$.

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

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

Chapter 6

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 :

```
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.

6.1 Integer Family

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

1. bool : Boolean
2. short : Short
3. char : Character
4. int : Integer
5. long : Long
6. INT : BigInteger

There are no primitive types. That was an absolutely disaster of a design that was made in Java land, which I fixed it here. Everything is, under the hood, an object here.

6.1.1 *Boolean*

The syntax is :

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

Observe both in action :

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

6.1.2 *Short*

This is almost never required, and is there for backward compatibility with Java data types. The syntax is :

```
val = short(value, optional_default_value = null )
```

Usage is :

```
val = short("hello") // val is null
val = short("hello",0) // val is 0
val = short('42') // val is 42
val = short(42) // val is 42
```

6.1.3 *Character*

For almost all practical purposes, character is noting but the short value, interpreted by a code page.

The syntax is :

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

Usage is :

```
val = char("hello") // val is "h"
val = char(121231231,0) // val is 0
val = char('4') // val is '4'
// ascii stuff?
val = char(65) // val is 'A'
```

Generally none needs to get into this, because generally $String.charAt(index)$ is a good substitute for finding a character.

6.1.4 *Integer*

This is very useful and the syntax is :

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

Usage is :

```
val = int("hello") // val is null
val = int("hello",0) // val is 0
val = int('42') // val is 42
val = int(42) // val is 42
val = int ( 10.1 ) // val is 10
val = int ( 10.9 ) // val is 10
```

6.1.5 *Long*

This is rarely required, and the syntax is :

```
val = long(value, optional_default_value = null )
```

Usage is :

```
val = long("hello") // val is null
val = long("hello",0) // val is 0
val = long('42') // val is 42
val = long(42) // val is 42
val = long( 10.1 ) // val is 10
val = long( 10.9 ) // val is 10
```

6.1.6 *BigInteger*

This is sometimes required, and the syntax is :

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

Usage is :

```
val = INT("hello") // val is null
val = INT('hi',10,42 )// val is 42
val = INT('42') // val is 42
val = INT(54,13 ) // val is 42
```

6.2 RATIONAL NUMBERS FAMILY

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

1. float : Float
2. double : Double
3. BigDecimal : BigDecimal

6.2.1 *Float*

This is not very useful and the syntax is :

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

Usage is :

```
val = float("hello") // val is null
val = float("hello",0) // val is 0.0
val = float('42') // val is 42.0
val = float(42) // val is 42.0
val = float ( 10.1 ) // val is 10.1
val = float ( 10.9 ) // val is 10.9
```

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

```
val = 0.01 // val is a float type, automatic
```

6.2.2 *Double*

This is generally required, and the syntax is :

```
val = double(value, optional_default_value = null )
```

Usage is :

```
val = double("hello") // val is null
val = double("hello",0) // val is 0.0
val = double('42') // val is 42.0
val = double(42) // val is 42.0
val = double( 10.1 ) // val is 10.1
val = double( 10.9 ) // val is 10.9
```

6.2.3 *BigDecimal*

This is sometimes required, and the syntax is :

```
val = DEC(value,default_value = null )
```

Usage is :

```
val = DEC("hello") // val is null
val = DEC('hi',10,42 )// val is 42.0
val = DEC('42') // val is 42.0
val = DEC(42.00001 ) // val is 42.00001
```

6.3 THE CHRONO FAMILY

Handling date and time has been a problem, that too with timezones. nJexl simplifies the stuff. We have three basic types to handle date/time:

1. date : java.util.Date : because of Java compatibility with SDKs.
2. time : org.joda.time.DateTime : included because it is the best chrono library out there.
3. instant : java.time.Instant : for newer systems who wants to experiment.

6.3.1 *Date*

This is how you create a date:

```
val = date([ value, date_format ] )
```

With no arguments, it gives the current date time:

```
today = date()
```

The default date format is $yyyyMMdd$, so :

```
dt = date('20160218') // Thu Feb 18 00:00:00 IST 2016
```

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

```
dt = date('2016/02/18', 'yyyy/MM/dd' )
// dt := Thu Feb 18 00:00:00 IST 2016
dt = date('2016-02-18', 'yyyy-MM-dd' )
// dt := Thu Feb 18 00:00:00 IST 2016
```

6.3.2 *Time*

This is how you create a joda DateTime:

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

With no arguments, it gives the current date time:

```
today = time()
```

The default date format is $yyyyMMdd$, so :

```
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 :

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

If you want to convert one time to another timezone, you need to give the time, and the timezone:

```
dt = time()
dt_honolulu = time(dt , 'Pacific/Honolulu' )
// dt_honolulu := 2016-02-17T17:23:02.754-10:00
dt_ny = time( dt, 'America/New_York' )
// dt_ny := 2016-02-17T22:23:02.754-05:00
```

6.3.3 *Instant*

With no arguments, it gives the current instant time:

```
today = instant()
```

It is freely mixable with other chrono types.

6.3.4 *Comparison on Chronos*

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

```
d = date() // wait for some time
t = time() // wait for some time
i = instant()
// now compare
c = ( d < t and t < i ) // true
c = ( i > t and t > d ) // 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 :

```
d = date('19470815') // Indian Day of Independence
t = time('19470815') // Indian Day of Independence
// now compare
c = ( d == t ) // true
```

6.3.5 *Arithmetic on Chronos*

Dates, obviously can not be multiplied or divided. That would be a sin. However, dates can be added with other reasoanable 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 DateTime.

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

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

Same can easily be achived by the *plusDays*() function :

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

And times can be subtracted :

```
d = time('19470815') // Indian Day of Independence
nd = d.plusDays(1) // 1947-08-16T00:00:00.000+05:30
diff = nd - d
//diff := 86400000 ## Long (millisec gap between two dates)
```

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

6.4 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. 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 nJexl by the function *str*().

6.4.1 *Null*

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

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

6.4.2 Integer

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

```
bi = INT(42)
s = str(bi) // '42'
s = str(bi,2) // base 2 representation : 101010
```

6.4.3 Floating Point

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

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

To illustrate the point:

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

6.4.4 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:

1. Date : SimpleDateFormat
2. Time : DateTimeFormat

The syntax is :

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

Now, some examples :

```
d = date()
t = time()
i = instant()
str( d ) // default is 'yyyyMMdd' : 20160218
str( t , 'dd - MM - yyyy' ) // 18 - 02 - 2016
str( i , 'dd-MMM-yyyy' ) // 18-Feb-2016
```

6.4.5 *Collections*

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

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

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

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

6.4.6 *Generalised toString*()

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

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

Another way to handle the linearisation is that of dictionary like property buckets:

```
d = { 'a' : 10 , 'b' : 20  }
s_form = str{  [ $.a , $.b ]  }(d,'@' )
// lineraize, generates : 10@20
```

6.5 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.

6.5.1 *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:

```
d = { : }
c1 = type(d) // c1 := java.util.HashMap
i = 42
c2 = type(i) // c2 := java.lang.Integer
c3 = type(20) // c3 := java.lang.Integer
c3 == c2 // true
c1 == c2 // false
```

6.5.2 *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 nJexl which reads : *"find a type at runtime, kill the type"* , we are pretty open into the type business :

```
a = [1,2,3]
ca = type(a) // Integer Array
b = list(1,3,2)
cb = type(b) // XList type
a == b // true
ca == cb // false
// type equals would be :
ca == cb  and 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 :

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

6.5.3 *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.

```
null isa null // true
a = [1,2,3]
a isa [] // false : a is a type of object array?
a isa [0] // true : a is a type of Integer array?
b = list(1,3,2)
b isa list() // true : b is a type of list ?
```

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

There is another distinctly better way of handling *isa*, by using *alias* strings, strings that starts with @ and has a type information, e.g. *@map* :

```
a = [1,2,3]
a isa "@arr" // true : a is a type that is array?
a isa "@array" // same, true, arr is same as array
b = list(1,3,2)
b isa "@list" // true : b is a type of list ?
{:} isa "@dict" // true
{:} isa "@map" // true
0 isa "@Z" // Z : the natural numbers
0 isa "@num" // num : the numbers
4.2 isa "@Q" // Q : the rational numbers
s = set(1,2,3)
s isa "@set" // true
t = time()
t is a "@chrono" // true : for date/time/instant
e = try{ 0/0 }()
e isa '@error' // true
```

The strings used are case insensitive. One can pass arbitrary class full name matcher too :

```
'abc' isa '@String' // true
'abc' isa '@java\.lang\.String' // true
'abc' isa '@str' // true
'abc' isa '@foo' // false
'abc' isa '@s' // false, must take 3 or more letters
```

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

6.5.4 *inspect() function*

A proper reflection requires to *inspect* a particular object type. The *inspect*() function comes to foray. The objective of this function is to return an UnmodifiableMap representing the structure of the object. In short, the map returned has these key ingredients :

1. The type property : designated as "*t*". Holds the type of the object or class.
2. The static fields property : designated as "*F*". Holds the static fields of the object or class, with their type. It is a list.
3. The instance fields property : designated as "*f*". Holds the instance fields of the object or class, with their type. It is a list.
4. The instance methods property : designated as "*m*". Holds the methods of the object or class, with their name. It is a list.

This suitably demonstrates the usage:

```
d = {'a' : 42 }
id = inspect(d)
/*
{t=java.util.HashMap,
F=[(UNTREEIFY_THRESHOLD,int), (TREEIFY_THRESHOLD,int),
(DEFAULT_LOAD_FACTOR,float),
(DEFAULT_INITIAL_CAPACITY,int), (serialVersionUID,long),
(MAXIMUM_CAPACITY,int), (MIN_TREEIFY_CAPACITY,int)],

f=[(entrySet,interface java.util.Set), (threshold,int),
(modCount,int), (size,int), (loadFactor,float),
(table,class [Ljava.util.HashMap$Node;)],

m=[getClass, getOrDefault, newTreeNode, replace, putMapEntries,
   put,
containsValue, compute, merge, entrySet, writeObject,
   containsKey,
access$000, eq, comparableClassFor, readObject, afterNodeAccess,
size, loadFactor, newNode, replacementTreeNode, hash,
   reinitialize,
internalWriteEntries, wait, values, computeIfAbsent, notifyAll,
 registerNatives, replaceAll, remove, notify, capacity,
    replacementNode,
  hashCode, get, putAll, putVal, keySet, removeNode, forEach,
     treeifyBin,
  clear, isEmpty, tableSizeFor, afterNodeRemoval,
     computeIfPresent,
  compareComparables, equals, clone, resize, toString,
  finalize, getNode, putIfAbsent, afterNodeInsertion]}

*/
```

6.5.5 *Serialisation using dict() function*

Sometimes it is of importance to inspect an instance as what the instance is supposed to be, as a property bucket. That is where the *dict*() function comes in.

```
d = dict()
pd = dict(d,null,null)
/*
pd := {modCount=1, @t=java.util.HashMap, size=1,
 loadFactor=0.75, entrySet=[a=42], threshold=12,
 table=[Ljava.util.HashMap$Node;@7e32c033}
*/
```

Note the quirky way to inspect what is there for the object when the object is dictionary. For objects which are not a dictionary, inspecting them is easier :

```
d = date()
pd = dict(d)
/*
pd :=  { cdate=2016-02-29T20:10:06.627+0530,
         @t=java.util.Date,
         fastTime=1456756806627 }
*/
```

6.5.6 *Field Access*

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

```
d = date()
d.fastTime  // some value
```

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

```
d = date()
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.

CHAPTER 7

Reusing Code

The tenet of nJexl 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.

7.1 The import directive

7.1.1 *Syntax*

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

```
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.

7.1.2 *Examples*

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

```
import 'java.lang.Integer' as Int
```

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

```
Int:parseInt('20') // int : 20
Int:valueOf('20') // int : 20
```

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

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

7.2 USING EXISTING JAVA CLASSES

7.2.1 *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.

```
// put all dependencies of xmlbeans.jar in there
success = load('path/to/xmlbeans_jar_folder') // true/false
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:

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

Calling methods now is easy:

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

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

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

7.2.2 *Import Enum*

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

```
// creates a wrapper for the enum
e = enum('com.noga.njexl.lang.extension.SetOperations')
// the enum value access using name
value = e.OVERLAP
// using integer index
value = e[4]
```

The same thing can be achieved by :

```
// gets the value for the enum
v = enum('com.noga.njexl.lang.extension.SetOperations',
         'OVERLAP' )
v = enum('com.noga.njexl.lang.extension.SetOperations', 4 )
```

7.2.3 *Import Static Field*

Import lets you import static fields too. For example :

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

However, another way of achieving the same is :

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

7.2.4 *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:

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

7.3 USING NJEXL SCRIPTS

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

7.3.1 *Creating a Script*

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

```
/* hello.jxl */
def say_hello(arg){
   write('Hello, ' + str(arg) )
}
s = "Some one"
my:say_hello(s)
/* end of script */
```

This script is ready to be re-used. Observe that the function, when the script calls it, comes with the namespace *my* :, which says that, use current scripts *say_hello*() function.

7.3.2 *Relative Path*

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

```
import './hello.jxl' 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.

```
import '_/hello.jxl' as Hello
Hello:say_hello('Waseem!" )
```

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

7.3.3 *Calling Functions*

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

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

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

```
import 'some/path/file' as NS
NS:__me__(args,... )
```

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

```
import '_/hello.jxl' as Hello
Hello:__me__()
```

CHAPTER 8

Functional Style

Functional style is a misnomer, in essence it boils down to some tenets:

1. Functions as first class citizens, they can be used as variables.
2. Avoid modifying objects by calling methods on them. Only method calls returning create objects.
3. Avoid conditional statements, replace them with alternatives.
4. Avoid explicit iteration, replace them with recursion or other higher order functions.

8.1 functions : in depth

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

8.1.1 *Function Types*

nJexl 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 :

```
def my_function( a, b ){ a + b }
```

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 :

```
l = list { $** 2}([0:10] )
```

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

```
import 'foo' as FOO
FOO:__me__()
```

8.1.2 *Default Parameters*

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

```
// default values passed
def my_function( a = 40 , b = 2 ){ a + b }
// call with no parameters :
write ( my_function() ) // prints 42
write ( my_function(10) ) // prints 12
write ( 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 :

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

But it is not :

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

8.1.3 *Named Arguments*

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

```
def my_function( a , b ){ write('(a,b) = (%s ,%s)\n' ,a , b ) }
my_function(1,2)  // prints (a,b) = (1,2)
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 :

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

8.1.4 *Arbitrary Number of Arguments*

Every nJexl function can take arbitrary no of arguments. To access the arguments, one must use the __*args*__ construct, as shown :

```
// this can take any no. of arguments
def my_function(){
   // access the arguments, they are collection
   s = str (  __args__ , '#' )
   write(s)
}
my_function(1,2,3,4)
/* 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*.

```
// this can take any no. of arguments, but named are 3
def my_function(a,b,c){
   // access the arguments, they are collection
   s = str (  __args__ , '#' )
   write(s)
}
my_function(1,2)
/* 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.

Given we do not know in advance how many parameters will be passed to a script, the scripts, when used as functions must use this construct as follows :

```
/* I am a script  */
def main(){
 //  process args ...
}
/*
 only when this is defined, I am a script
 but being called as a function
*/
if ( #def __args__ ) {
   x = __args__ // relocate
   main( __args__ = x )
}
```

8.1.5 *Arguments Overwriting*

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

```
l = ['a','b','c' ]
perm_3_from_3 = join{ #|set($)| == #|$| }(l,l,l)
l = ['a','b','c' ,'d' ]
perm_4_from_4 = join{ #|set($)| == #|$| }(l,l,l,l)
perm_2_from_4 = join{ #|set($)| == #|$| }(l,l)
```

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.

```
l = ['a','b','c' ,'d' ]
// this call overwrites the args :
perm_2_from_4 = join{ #|set($)| == #|$| }(__args__ = [l,l] )
```

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

```
perm_r = join{ #|set($)| == #|$| }(__args__ = array{ l }([0:r]))
```

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

8.1.6 *Recursion*

It is customary to introduce recursing 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:

```
/* successor function */
def s( n ){
    if ( n == 0 ) return 1
    return  ( s(n-1) + 1 )
}
/* addition function */
def add(a,b){
  if ( b == 0 ) return a
  return s ( add(a,b-1) )
}
```

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 addition function does not even have any addition symbol anywhere!

Now we test the functions:

```
write(s(0)) // prints 1
write(s(1)) // prints 2
write(s(5)) // prints 6
write(add(1,0)) // prints 1
write(add(1,5)) // prints 6
write(s(5,1)) // prints 6
```

Now, obviously we can do factorial :

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

8.1.7 *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:

```
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:

```
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:

```
factor = 3
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:

```
write ( multiplier(1) ) // prints 3
write ( 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.

8.1.8 *Partial Function*

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

```
// this shows the nested function
def func(a){
    // here it is : note the name-less-ness of the function
    r = def(b){ // which gets assigned to the variable "r"
       write("%s + %s ==> %s\n", a,b,a+b)
    }
    return r // returning a function
}
// get the partial function returned
x = func(4)
// now, call the partial function
x(2)
//finally, the answer to life, universe and everything :
x = func("4")
x("2")
```

This shows nested functions, as well as partial function.

8.1.9 *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 :

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

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

```
def  apply ( param , a_function ){
     a_function(param)
}
```

So, suppose we want to now apply arbitrary function :

```
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.

8.1.10 *Composition of Functions*

To demonstrate a composition, observe :

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

Now the application :

```
 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 :

```
def compose (){
    p = __args__  ; n = size( p )
    n >= 2 or bye('min 2 args are required!')
    i = p[0] ; p = p[[1:n]]
    lfold{  $(_$_) }( p, i)
 }
```

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

```
// note the nameless-ness :)
r = compose(6, def (a){ a** 2} , def (b){ b + 6 } )
write(r)
```

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

8.1.11 *Operators for Composition*

Functions support two operators :

1. The '*' operator for function composition :

$$(f * g)(x) := f(g(x))$$

2. The '**' operator for exponentiation, fixed point iteration :

$$(f ** n)(x) := f^n(x)$$

Let's have a demonstration :

```
// predecessor function
def p(){ int( __args__[0] ) - 1 }
// successor function
def s(){  int( __args__[0] ) + 1 }
// now use them
write( list( s(0) , s(1) , s(2) ) )
write( list( p(3) , p(2) , p(1) ) )
I = s * p // identity function !
write( list( I(0) , I(1) , I(2) ) )
// power?
add_ten = s**10
write( add_ten(0))
write( add_ten(10))
```

The result is :

```
[1, 2, 3]
[2, 1, 0]
[0, 1, 2]
10
20
```

With this composition defined as another function, there is a very nice way to do recursion. Suppose we want to define factorial :

```
def f(){
   n = __args__[0] + 1 ;  r = n * int ( __args__[1] )
   [ n, r ]
}
n = 5 // factorial 5 ?
factorial_n = f**n
r = factorial_n(0,1)
write ( r[1] ) // prints 120
```

Now, we can do Fibonacci :

```
def f( ) { p = __args__ ; n = p[0] + p[1] ; [ p[1] , n ] }
n = 5
fibonacci_n = f**n
r = fibonacci_n(0,1)
write ( r[1] ) // prints 8
```

8.1.12 *Eventing*

All nJexl 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 nJexl functions. See the object Event.

Here is a sample demonstration of the eventing:

```
def my_func(){
   write('My function got called with args: [ %s ] \n',
        str(__args__))
}
def event_hook(event){
    // eventing takes only one parameter
    method_name = event.method.name
    event_type = event.pattern
    method_args = event.args
    write( '%s happened with method "%s" \n' ,
              event_type , method_name )
    write( 'Arguments passed are [ %s ]\n' ,
             str(method_args, '#') )
}
// add the before hook
my_func.before.add( event_hook )
// add the after hook
my_func.after.add( event_hook )
// now call the function
my_func('Hello, world!')
```

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

```
__before__ happened with method "my_func"
Arguments passed are [ Hello, world! ]
My function got called with args: [ Hello, world! ]
__after__ happened with method "my_func"
Arguments passed are [ Hello, world! ]
```

8.2 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.

8.2.1 *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". nJexl 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 :

```
(njexl)write('Hello, World!')
```

This code writes 'Hello, World!' to the console. From the interpreter perspective, it identifies that it needs to call a function called write, 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.

8.2.2 *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 write code that reduces this complexity.

As an example, take a look about this string :

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

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

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

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

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

8.2.3 *Examples*

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

```
(njexl)x=1.01125
=>1.01125
(njexl)y=1.0113
=>1.0113
(njexl)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 :

```
(njexl)str:format("%.4f",x)
=>1.0113
(njexl)str:format("%.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 :

```
(njexl)c_string = "%%.%df"  ## This is the original one
=>%%.%df
## apply precision, makes the string into a function
(njexl)p_string = str:format(c_string,4)
=>%.4f
## apply a number, makes the function evaluate into a proper value
(njexl)str:format(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 :

```
(njexl)str:format(str:format(c_string,4),x)
=>1.0113
(njexl)str:format(str:format(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 write the corresponding test code? The test code would be, invariably messy :

```
if ( operation == '+' ) {
    do_plus_check();
} else if ( operation == '-' ) {
    do_minus_check();
}
// some more code ...
```

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

```
switch ( operation ){
    case '+' :
       do_plus_check(); break;
    case '-' :
       do_minus_check(); break;
    ...
 }
```

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:

$$operand_1 \quad < operator > \quad operand_2$$

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) :

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

8.2.4 *Reflection*

Calling methods can be accomplished using currying.

```
import 'java.lang.System.out' as out
def func_taking_other_function( func ){
    `#{func}( 'hello!' )`
}
my: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.

8.2.5 *Referencing*

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

```
(njexl)x = [1,2]
=>@[1, 2]
(njexl)y = { 'x' : x }
=>{x=[I@5b37e0d2}
(njexl)x = [1,3,4]
=>@[1, 3, 4]
(njexl)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 :

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

So, in effect I am using a dictionary to hold name of a variable, instead of having a hard variable reference, thus, when I am dereferencing it, I would get back the value if such a value exists!

8.3 AVOIDING CONDITIONS

As we can see the tenets of functional programming says 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.

8.3.1 *Theory of the Equivalence Class*

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

```
is_valid_date(d_str){
   num = int(d_str)
   days = num/1000000
   rest = num % 1000000
   mon = rest /10000
   year = rest % 10000
   max_days = get_days(mon,year)
   return (  0 < days and days <= max_days  and
             0 < mon and mon < 13 and
             year > 0 )
}
get_days(month,year){
   if ( month == 2  and leap_year(year){
       return 29
   }
   return days_in_month[month]
}
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 :

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

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

8.3.2 *Dictionaries and Functions*

Suppose we are given a 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 :

```
sort_a = index{  _ > 0 and $$[_ - 1 ] > $ }(collection) < 0
sort_d = index{  _ > 0 and $$[_ - 1 ] < $ }(collection) < 0
```

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 :

```
op   = { 'ascending' : '>' , 'descending' : '<'   }
sorted = index{
    _ > 0 and `$$[_ - 1 ] #{op} $` }(collection) < 0
```

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

```
sort_a = def(collection) {
          index{  _ > 0 and $$[_ - 1 ] > $ }(collection) < 0 }
sort_d = def(collection) {
          index{  _ > 0 and $$[_ - 1 ] < $ }(collection) < 0 }
verifiers  = { 'ascending' : sort_a , 'descending' : sort_d  }
verify_func = verifiers[sort_type]
verified = verify_func(collection)
```

8.3.3 *An Application : FizzBuzz*

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

```
/* Pure If/Else */
def fbI(range){
  for ( i : range ){
      if ( i % 3 == 0 ){
          if ( i % 5 == 0 ){
              write('FizzBuzz')
          } else {
              write('Fizz')
          }
      }else{
          if ( i % 5 == 0 ){
             write('Buzz')
            }else{
               write(i)
            }
      }
  }
}
```

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

```
def fbD(range){
  d = { 0 : 'FizzBuzz' ,
        3 : 'Fizz' ,
        5 : 'Buzz' ,
        6 : 'Fizz',
       10 : 'Buzz' ,
       12 : 'Fizz' }
  for ( x : range ){
       r = x % 15
       continue ( r @ d ){ write( d[r] ) }
       write(x)
  }
}
```

8.4 AVOIDING ITERATIONS

The last tenet is avoiding 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.

8.4.1 *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 :

```
d_start = time()
d_end = d_start.plusDays(10)
// a range from current to future
d_range = [d_start : d_end ]
// another range with 2 days spacing
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 : $PyYmMwWdDThHmMsS$. Curiously, date/time range also has many interesting fields :

```
r.years
r.months
r.weeks
r.days
r.hours
r.minutes
r.seconds
## tells you number of working days
## ( excludes sat/sun ) between start and end!
r.weekDays
```

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

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

All the range types have some specific functions.

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

8.4.2 *Iterative Functions : On Objects*

Upto this point we have discussed about the *select* , *index* functions which are not associated with any objects. But every collection object have those in build. For example, observe :

```
l = list(1,2,3,4,1,2,3)
s = l.set() // [1,2,3,4]
l1 = l.select{ $ >= 2 }() //[2, 3, 4, 2, 3]
exists = l.indexOf{ $ > 2  }() // 1
exists = l.lastIndexOf{ $ > 2  }() // 6
```

8.4.3 *The Fold Functions*

The rationale of the fold function is as follows :

```
def do_something(item){ /* some code here */ }
for ( i : items ){
   do_something(i)
}
// or use fold
lfold{ do_something($) }(items)
```

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 :

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

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

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

This prints :

```
4 3 2 1 0
```

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

```
fact_n = lfold{ _$_ * $  }( [2:n+1] , 1 )
```

We move onto find Fibonacci :

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

We now sum the elements up:

```
sum_n = rfold{ _$_ + $  }( [2:n+1] , 0 )
```

Generate a set from a list :

```
my_set = lfold{ _$_ += $ }( [1,2,1,2,3,4,5] , set() )
```

Finding min/max of a collection :

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

Finding match index of a collection :

```
inx = lfold{ break ( $  > 3 ){  _$_ = _ }  }( [1,2,1,2,3,4,5]  )
```

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:

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

To do an *index*() function :

```
// first find the element greater than 3
selected = lfold{
                break ( $  > 3 ) { _ }
           }( [1,2,1,2,3,4,5] )
```

And to do an *rindex*() function :

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

CHAPTER 9

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. nJexl let's you do both, and those are the topic of this chapter.

9.1 reading

9.1.1 *read() function*

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

```
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.

```
value = read('my_file.txt')
/* value contains all the text in my_file.txt */
```

9.1.2 *Reading from url : HTTP GET*

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

```
value = read('http://www.google.co.in')
/* value contains all the data in the google home page */
```

To handle the timeouts, it comes with overloads:

```
// url, connection timeout, read timeout, all in millisec
value = read('http://www.google.co.in', 10000, 10000 )
/* value contains all the data in the google home page */
```

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

```
_url_ = 'https://httpbin.org/get'
params = { 'foo' : 'bar' , 'complexity' : 'sucks'  }
// str: is the namespace alias for 'java.lang.String'
data = str{
          str:format( "%s=%s" , $.key, $.value )
      }( params.entrySet() , '&' )
response = read( str:format("%s?%s", _url_ , data ) )
```

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

9.1.3 *Reading All Lines*

The function *lines*() 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.

```
for ( line : lines('my_file.txt') ){
   write(line) // proverbial cat program
}
```

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

```
ll = lines ( 'my_file.txt' , true )
```

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

9.2 WRITING

9.2.1 *write() function*

We are already familiar with the write function. With a single argument, it writes the data back to the standard output:

```
write('Hello,World!')
```

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

```
// creates my_file.txt, writes the string to it
write('my_file.txt', 'Hello,World!')
```

Given there is formatting arguments with "%" symbols, works as if it is a *printf*() call :

```
//  writes the string to it
write('%s\n', 'Hello,World!')
```

The formatting guide can be found here.

9.2.2 *Writing to url : HTTP POST*

Write can be used to send POST call to urls, observe the following:

```
_url_ = 'https://httpbin.org/post'
params = { 'foo' : 'bar' , 'complexity' : 'sucks'   }
response = write(  _url_ , params )
```

9.2.3 *Sending to url : send() function*

To write back to url, or rather for sending anything to a server, there is a specialised send() function.

```
_url_ = 'https://httpbin.org/post'
params = { 'foo' : 'bar' , 'complexity' : 'sucks'   }
// url,  protocol     ,  body, headers, connection timeout, read
    timeout
response = send(  _url_ , "GET"   , params , {:} , 1000, 1000 )
```

The SOAP XML stuff can be easily done with *send*. See the example soap body we will send to :

```
<?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:

```
template_pay_load = read('samples/soap_body.xml')
pay_load = str:format ( template_pay_load , 'Hyderabad', 'India'
    )
headers = { "SOAPAction" : "http://www.webserviceX.NET/
   GetWeather" ,
             "Content-Type" : "text/xml; charset=utf-8" }
__url__ = "http://www.webservicex.net/globalweather.asmx"
resp = send( __url__ , "POST" , pay_load , headers )
x = xml(resp)
sr = x.element("//GetWeatherResult")
write(sr.text)
```

9.3 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.

9.3.1 *lines() function*

lines() function generates an iterator, which can be used to read the file. It caches the previous reads, so after you ended reading, resetting the iterator you can read it again. Observe we are replicating the Unix program wc:

```
fi = lines('my_file.txt' )
// count no of lines
lc = lfold{ _$_ += 1 }(fi,0)
// resets the iterator
fi.reset()
// character count
cc = lfold{ _$_ += size($) + 1  }(fi,0)
// resets the iterator
fi.reset()
// word count?
wc = lfold{ _$_ += size($.split(" \t") ) }(fi,0)
// reset again
fi.reset()
// or in a single go, using a tuple value
#(l,w,c) = lfold{
               // do a tuple assignment
               #(lc,cc,wc) = _$_
               lc+=1 ; cc += size($) + 1
               wc += size($.split(" \t") )
              _$_ = [ lc, cc, wc ]
      }(fi,[0,0,0] )
```

9.3.2 *fopen() function*

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

```
fp = fopen( path [, mode_string ] )
```

The mode strings are defined as such: There are 3 modes :

1. Read : the default mode : specified by "r"
2. Write : the write mode : specified by "w"
3. Append : the append mode : specified by "a"

9.3.3 *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 :

```
// open in read mode : get a reader
fp = fopen( path ,"r" )
while ( true ){
   // call the standard java functions
   line = fp.readLine()
   break( null == line )
   write(line)
}
// close the reader
fp.close()
```

9.3.4 *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 :

```
// open in read mode : get a reader
fp = fopen( "hi.txt" ,"w" )
// call the standard java functions
fp.println("Hi")
// close the Stream
fp.close()
```

9.4 WORKING WITH JSON

9.4.1 *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 nJexl they are always casted back to a dictionary.

9.4.2 *json() function*

Suppose the json string is there in a file:

```
/* sample.json file  */
{
  "noga": {
    "dbName": "noga",
    "url": "jdbc:postgresql://localhost:5432/noga",
    "driverClass" : "org.postgresql.Driver",
    "user": "noga",
    "pass": ""
  },
  "some2": {
    "dbName": "dummy",
    "url": "dummy",
    "driverClass" : "class",
    "user": "u",
    "pass": "p"
  },
}
```

To read the file (or the text which has json) :

```
jo = json('sample.json')
```

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.

9.4.3 *Accessing Fields*

Now, to access any field:

```
jo.noga.dbName // "noga"
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 nJexl, 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:

```
prop_value = 'noga.dbName'
value = `jo.#{prop_value}` // same thing, "noga"
```

In this way, one can remove the hard coding of accessing JSON objects.

9.5 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 be 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 I could not avoid XML.

9.5.1 *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 XmlMap object :

```
x = xml('sample.xml', 'UTF-8' ) // x is an XmlMap object
```

The encoding string (UTF-8) is optional. Default is UTF-8.

The function xml() can also convert a proper Java Object into string, in other words, it does Xml Serialisation. Suppose we have the JSON object :

```
/* demo.json file  */
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup
                       Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
          "para": "A meta-markup language, used to create markup
              languages such as DocBook.",
          "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

This is how we can generate an XML out of it :

```
// load json
jo = json('db.json')
// convert to xml string
x = xml(jo)
```

After formatting the xml string is :

```
<root>
  <glossary>
    <title>example glossary</title>
    <GlossDiv>
      <GlossList>
        <GlossEntry>
          <GlossTerm>Standard Generalized Markup Language</GlossTerm>
          <GlossSee>markup</GlossSee>
          <SortAs>SGML</SortAs>
          <GlossDef>
                 <para>A meta-markup language, used to
                 create markup languages such as DocBook.
                 </para>
            <GlossSeeAlso>
              <i>GML</i>
              <i>XML</i>
            </GlossSeeAlso>
          </GlossDef>
          <ID>SGML</ID>
          <Acronym>SGML</Acronym>
          <Abbrev>ISO 8879:1986</Abbrev>
        </GlossEntry>
      </GlossList>
      <title>S</title>
    </GlossDiv>
  </glossary>
</root>
```

9.5.2 *Accessing Elements*

Given I have the XmlMap object, I 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 :

```
// XmlMap of the x
y = xml(x)
y.root.children[0].name //glossary
y.root.children[0].children[0].text // example glossary
```

For the slideshow xml:

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
x.root.name //slideshow
// access attributes
x.root.children[0].attr.type // "all"
x.root.children[0].attr['type'] // "all"
```

The other way of accessing elements, is to use the *element*() function, which uses XPATH :

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
// get one element
e = x.element("//slide/title")// first title element
e.text // "Wake up to WonderWidgets!"
e = x.element("//slide[2]/title" ) // second element
e.text // "Overview"
```

In the same way, for selecting multiple elements, *elements*() function is used :

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
// get all the elements called titles
es = x.elements("//slide/title" ) // a list of elements
// print all titles ?
lfold{  write( $.text ) }(es)
```

9.5.3 *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 :

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
// text of the title element
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.

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
// text of the title element, exists?
b = x.exists("//slide/title/text()" ) // bool : true
b = x.exists("//slide/title" ) // bool : true
b = x.exists("//foobar" ) // bool : false
```

Obviously, one can mix and match, that is, give a default to the *xpath*() function :

```
// string : text comes
t = x.xpath("//slide/title/text()" , 'NONE_EXISTS' )
// string : 'NONE_EXISTS'
t = x.xpath("//foobar/text()",  'NONE_EXISTS' )
```

9.5.4 *Xml to JSON*

The xml can be converted to a suitable JSON equivalent, by invoking the *json*() function. This function is available on the XmlMap object, also the standard *json*() function would convert any XmlMap object.

```
// XmlMap of the x
x = xml('sample.xml') // slideshow xml
// json form of the xml
jo = x.json()
jo = json(x)   // same as above
```

9.6 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, $r_i[C_j] \to r_{ij}$, 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.

9.6.1 *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 :

```
m = matrix("test.tsv", '\t', true ) // loads the matrix
m.coloumns // the set of columns
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.

9.6.2 *Accessing Data*

A data matrix can be accessed by rows or by columns. For example :

```
x = m.rows[0][1] // x is "Eve"
x = m.rows[1][2] // x is "Doe"
```

Now, using column function $c()$

```
y = (m.c(1))[1] // y is "Eve"
y = (m.c(2))[1] // y is "Doe"
```

9.6.3 *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:

```
t = m.tuple(1) // 2nd row as a Tuple
t.0 // 2
t["First Name"] // John
t.Points // 80
```

This tuple object is the one which gets returned as implicit row, when select operations on matrices are performed.

9.6.4 *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 :

```
m.c(0,1) // selects first column, and a cell of 3rd row
m.c(0, [0:2]) // selects first column, and two cells : 1st and 2
   nd 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 :

```
x = m.select(0,2)
// x  := [[1, Jackson], [2, Doe], [3, Johnson], [4, Smith]]
```

Same thing can be accomplished by :

```
x = m.select('Number' , 'Last Name' )
// x  := [[1, Jackson], [2, Doe], [3, Johnson], [4, Smith]]
```

Changing the order of the columns results in a different Tuple, as it should :

```
x = m.select(2 , 0 )
// x :=  [[Jackson, 1], [Doe, 2], [Johnson, 3], [Smith, 4]]
```

As always, *select*() works with condition, so :

```
x = m.select{  $.Number > 2  }(2 , 0 )
// 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 :

```
x = m.select{
        where ( $.Number > 2 ){
            $[2] = $[2].toLowerCase()
          }
       }(2 , 0 ) // x :=  [[johnson, 3], [smith, 4]]
```

9.6.5 *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 :

```
m2 = m.matrix{
        where ( $.Number > 2 ){
            $[2] = $[2].toLowerCase()
          }
       }(2 , 0 )
```

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.

9.6.6 *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 :

```
def match_all_rows(m1,m2){
   count = 0
   for ( r1 : m1.rows ){
     // linearise the left tuple
     st1 = str(r1,'#')
     for ( r2 : m2.rows ){
         // linearise the right tuple
         st2 = str(r2,'#')
         // they match
         if ( r1 == r2 ){ count += 1 }
     }
   }
   return ( count == size(r1.rows) and  count == size(r2.rows) )
}
```

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:

```
def match_all_rows(m1,m2){
   // exactly m * n
   l1 = list{ str($,'#') }(m1.rows )
   // exactly m * n
   l2 = list{ str($,'#') }(m2.rows )
   // exactly n
   diff = l1 ^ l2
   return empty(diff)
}
```

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 :

```
// a set of attribute indices
key_attrs = [ index_1 , index_2 , ...  ]
def is_key( m, key_attrs){
   s = set{
         r = $ // store the row
         // generate a list by projecting the attributes
         l = lfold{ r[$] }(key_attrs)
         // linearise the tuple
         str(l,'#')
      }(m.rows)
   return  size(s) == size(m.rows)
}
```

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 :

```
// a set of attribute indices
key_attrs = [ index_1 , index_2 , ...  ]
def key( m, key_attrs, sep){
   keys = lfold{
         r = $ // store the row
         // generate a list by projecting the attributes
         l = lfold{ r[$] }(key_attrs)
         // linearise the tuple
         k = str(l,sep)
         if ( k @ _$_ ){ // the key already exists
            _$_[k] += _ // append the current row index to the
               rows
          }else{
            // key does not exist, create one and then append
            _$_[k] = list(_)
           }
         _$_
      }(m.rows, dict() )
   return keys
}
```

This idea is already in-built in the DataMatrix, and is known as the *keys*() function :

```
m.keys{  /* how to generate one */ }( args... )
m.keys( columns_for_keys )
```

So, to simply put, *keys*() generate a key dict for the matrix, as shown below, using the matrix in (9.6.1) :

```
// the key is the column Number
m.keys( 'Number' )
// see what the keys are ?
m.keys /* {4?=[3], 3?=[2], 2?=[1], 1?=[0]}  */
```

We can generate a non unique key too, just to show the practical application :

```
// the key is the column Number modulo 2: [0,1]
m.keys{  $.Number % 2  }( )
// see what the keys are ?
m.keys /* {0=[1, 3], 1=[0, 2]}  */
```

9.6.7 *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.

```
def aggregate(m,columns , aggregate_function ){
  agg_rows = list()
  for ( key : m.keys ){
      row = list()
      rows_list = m.keys[key]
      for ( c : columns ){
          // aggregate the rows on a single column
          col_data = m.c( c, rows_list )
          agg_data = aggregate_function( col_data )
          row += agg_data // add a replacement for all the rows
      }
      // add to the newer row
      agg_rows += row
  }
  // now we have the new matrix data rows
}
```

The syntax for the function is :

```
ma = m.aggregate( col_name_or_index_list )
ma = m.aggregate( col_name_or_index1, col_name_or_index2, ... )
ma = m.aggregate{
                 /*  pass the function to aggregate */
                }( col_name_or_index1, col_name_or_index2, ...
                   )
```

As an example, say we take the key example:

```
ma = m.aggregate{ str($,'&') } ( 'First Name' , 'Last Name'  )
ma.rows
/*
[
  [John&Jill, Doe&Smith],
  [Eve&Adam, Jackson&Johnson]
]
*/
```

9.6.8 *Diff Functions*

Sometimes it is necessary to difference out two matrices.

Suppose we have two matrices :

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

Table 1

Number	First Name	Last Name	Points	Extra
1	eve	jackson	92	
2	john	doe	83	x
3	adam	johnson	68	
4	jill	smith	49	xx

Table 2

Now, we want to diff these two. There is a very simple way to diff :

```
m1 = matrix('table1.tsv')
m2 = matrix('table2.tsv')
// do is the diff object storing : m1-m2, m1 & m2, m2-m1
do = m1.diff(m2)
/* do does not contain any common stuff at all */
```

The diff object has the following fields :

1. Left - Right : *lr* : keys not matching
2. Common : *id* : keys match, but the values do not
3. Right - Left : *rl* : keys not matching

It also has a method $diff()$ telling that if they differ or not. To demonstrate :

```
m1 = matrix('table1.tsv')
m2 = matrix('table2.tsv')
m1.keys('Number')
m2.keys('Number')
// do is the diff object storing : only m1 & m2 : keys match!
do = m1.diff(m2)
```

This poses a problem. We need to specify, how to differentiate two tuples, when the key matches. In effect, we want to define equality over two tuples. That can be done with :

```
// do is the diff object storing : everything matches
do = m1.diff{
 $[0]['First Name'].toLowerCase() == $[1]['First Name'].
    toLowerCase() and
 $[0]['Last Name'].toLowerCase() == $[1]['Last Name'].
    toLowerCase() and
 #| $[0]['Points'] - $[1]['Points'] | < 10
}(m2)
```

and now, the two matrices will match, with no diff.

9.7 DATABASE CONNECTIVITY

CHAPTER 10

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.

10.1 Process and Threads

10.1.1 *system() function*

To run any operating system command from nJexl, 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 :

```
status = system(command_to_os)
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 :

```
status = system("whoami") // system prints "noga"
/* status code is 0 */
```

In case of unknown command, it would throw an error:

```
status = system("unknown command")
/* error will be thrown */
```

In case of command is found, but command execution reported an error, it status would be non zero:

```
status = system("ls -l thisDoesNotExist")
```

```
/* os reprots
ls: thisDoesNotExist: No such file or directory
status := 1
*/
```

In the same way, multiple words can be put into system, which is of some usefulness when one needs to pass quoted arguments :

```
status = system("ls" , "-al" , "." ) // system prints result of
    "ls -al ."
/* status code is 0 */
```

There is no way to create a process, and have a process handle as of now, one can use the Process class to create process if required.

10.1.2 *thread() function*

thread() function, creates a thread. The syntax is simple :

```
t = thread{  /* thread function body  */  }( args_to_be_passed )
```

Thus, simply, use *thread*() to create a thread:

```
t = thread{
   write('My Id is %d\n', _ )
   write('My args are: %s\n', str($$,'\n' ) )
   write('I am %s\n', $ )
  }( "Hello", "world" )
/* java isAlive() maps to alive */
while ( t.alive ){
   cur_thread = thread() // returns current thread
   cur_thread.sleep(1000)
}
```

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]
```

10.1.3 *until() 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 *until*().

The syntax is :

```
until [ { condition-body-of-function } ]
      ( [ timeout-in-ms = 3000, [ polling-interval-in-ms = 100]
         ])
```

So these are explanations :

```
until() // this is ok : simple default wait
until (4000) // ok too !
until (4000, 300 ) // not cool : 300 is rejected , no condition!
```

A very mundane example would be :

```
i = 3
until { i-= 1 ; i == 0 }( 1000, 200 ) // induce a bit of wait
// this returns true : the operation did not timeout
i = 100
until { i-= 1 ; i == 0 }( 1000, 200 ) // induce a bit of wait
// this returns false : timeout happened
```

and, finally, we replace the while statement of the thread example using *until*() :

```
t = thread{
   /* do anything one needs */
  }(  )
/* java isAlive() maps to alive */
until{ ! t.alive }( 10000, 300 )
```

10.1.4 *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 :

```
count = 0
// create a list of threads to do something?
l = list{ thread{ count+= 1 }() }( [0:10] ) // increase count...
until{ running = select{ $.isAlive }(l) ; empty(running)  }(
   10000, 50 )
write(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{}) :

```
count = 0
// create a list of threads to do something?
l = list{ thread{ #atomic{ count+= 1 } }() }( [0:10] )
// above tries to increase count...
until{ running = select{ $.isAlive }(l) ; empty(running)  }(
   10000, 50 )
write(count) // the value is 10
```

and now, the count is always 10.
Atomic can be used to revert back any changes in the variables :

```
x = 0
#atomic{
   x = 2 + 2
}
#atomic{
   x = 10
   write(x) // prints 10
/* error, no variable as "p', x should revert back to 4 */
   t = p
}
write(x) // x is 4
```

There is this function *atomic*() which takes primitive data types, and converts them into their atomic equivalent. See the Concurrency package. Thus,

```
ai = atomic(o) // atomic integer : valued at 0  : int
al = atomic(ol) // atomic long : valued at 01 : long
ai += 42 // ai sets to 42, atomic operation
al += 42 // al sets to 42, atomic operation
```

10.1.5 *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*() function. The syntax is simple :

```
// times the body
#(time_taken_in_nano_sec, result ) = #clock{ /* body  */}
```

Thus this is how you tune a long running code:

```
m1 = 0
#(t,r) = #clock{
   // load a large matrix, 0.2 million rows
   m1 = matrix('test.csv', ',' , false)
   0 // return 0
}
write(t) // writes the timing in nano seconds
write(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.

```
#(t,r) = #clock{  x = foobar_not_defined_var }
// r would be Jexl 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.

10.2 HANDLING OF ERRORS

I toyed with the idea of try-catch, and then found that : Why Exception Handling is Bad? ; GO does not have exception handling. And I believe it is OK not to have exceptions. Code for everything, never ever eat up error like situations. 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.

10.2.1 *error() function*

But we need errors to be raised. This is done by the *error*() function.

```
error[ { /* body which when evaluates to true raise error */} ](
    args... )
```

The function *error*() can have multiple modes, for example this is how one raise an error with a message :

```
error( "error message")
```

This is how one raise an error on condition :

```
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:

```
error{ /* error condition */ }("error message")
```

10.2.2 *try() function*

In case the system can raise an error, that needs to be caught, *try*() function is used. The syntax is :

```
try{ /* guard block */ }( return_value_if_error_occured = null )
```

Thus, in case of no error, *try*() returns the value of the block, but in case of error it returns the value passed in the argument, default for that is *null*. Observe :

```
import 'java.lang.Integer' as INT
n = try{ INT:parseInt('42') }()   // n := 42 int
// error will be raised and caught
n = try{ INT:parseInt('xx42') }()   // n := null
// error and 0 value would be assigned
n = try{ INT:parseInt('xx42') }(0)   // n := 0
```

10.2.3 *Multiple Assignment*

GoLang supports multiple assignments. This feature is picked from GoLang. Observe the following :

```
#(a,b) = [ 0 , 1 , 2 ] // a = 0, b = 1
#(: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.

10.2.4 *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.

```
import 'java.lang.Integer' as INT
#(n,:e)  =  INT:parseInt('42') // n := 42 int, e := null
// error will be raised and caught
#(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:

```
import 'java.lang.Integer' as INT
#(n,:e)  =  INT:parseInt(str_val)
empty(e) or 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.

10.3 ORDER AND RANDOMNESS

10.3.1 *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.

```
s = '11,12,31,34,78,90' // CSV integers
l = select{ where ( ($ = int($) ) !=null ){ $ } }(s.split(','))
// 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 :

```
s = '11,12,31,34,78,90'
tmp = ''
l = list()
for ( c : s ){
    if ( c == ',' ){
        l += int(tmp) ; tmp = '' ; continue
    }
    tmp += c
}
l += int(tmp)
write(l)
```

However, this is a problem, because we are using too much coding. Fear not, we can still reduce it :

```
tmp = ''
l = select{
    if ( $ == ',' ){  $ = int(tmp) ; tmp = '' ; return true }
    tmp += $ ; false  }(s.toCharArray() )
l += int(tmp)
```

Ok, goodish, but still too much code. 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 :

```
tokens( <string> , <regex> ,
match-case = [true|false] )
// returns a matcher object
tokens{ anon-block }( <string> , <regex> , match-case = [true|
    false] )
// calls the anon-block for every  matching group
```

Thus, using this, we can have:

```
// what to do : string : regex
l = tokens{ int($) }(s, '\d+')
```

and we are done. That is precisely how code is meant to be written.

10.3.2 *hash() function*

Sometimes it is important to generate hash from a string. To do so :

```
h = hash('abc')
// h   := 900150983cd24fb0d6963f7d28e17f72
```

It defaults to "MD5", so :

```
hash( 'MD5' ,'abc')
// h := 900150983cd24fb0d6963f7d28e17f72
```

They are the same. One can obviously change the algorithm used :

```
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 :

```
h = hash('e64', 'hello, world')
//h := aGVsbG8sIHdvcmxk
//And to decode it back :
s = hash('d64', 'aGVsbG8sIHdvcmxk' )
// s:= "hello, world"
```

10.3.3 *Anonymous Comparators*

Ordering requires to compare two elements. The standard style of Java comparator is to code something like this :

```
def compare( a, b ) {
    if ( a < b ) return -1
    if ( a > b ) return  1
    return 0
}
```

Obviously, one needs to define the relation operator accordingly. For example, suppose we have student objects :

```
student1 = { 'name' : 'Noga' , 'id' : 42  }
student2 = { 'name' : 'Waseem' , 'id' : 24  }
```

And we need to compare these two. We can choose a particular field, say "id" to compare these :

```
def compare( a, b ) {
    if ( a.id < b.id ) return -1
    if ( a.id > b.id ) return  1
    return 0
}
```

Or, there is another way to represent the same thing :

```
def compare( a, b ) {
    ( a.id < b.id )
}
```

In nJexl 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 :

```
cmp = comparator(a,b)
if ( cmp ) {
  write( 'a<b' )
}else{
  write( 'a >= b' )
}
```

In the subsequent section anonymous comparators would be used heavily. The basic syntax of these functions are :

```
order_function{ /* anonymous comparator block */ }(args... )
```

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 in order.

10.3.4 *Sort Functions*

Sorting is trivial:

```
cards = ['B','C','A','D' ]
sa = sorta(cards) // ascending
// sa := [A, B, C, D]
sd = sortd(cards) //descending
// 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:

```
students = [ {'roll' : 1, 'name' : 'X' } ,
             {'roll' : 3, 'name' : 'C' } ,
             {'roll' : 2, 'name' : 'Z' } ]
sa = sorta{ $[0].name < $[1].name }(students)
/* sa := [{roll=3, name=C}, {roll=1, name=X},
        {roll=2, name=Z}] */
```

Obviously we can do it using the roll too:

```
sa = sorta{ $[0].roll < $[1].roll }(students)
// sa := [{roll=1, name=X}, {roll=2, name=Z}, {roll=3, name=C}]
```

10.3.5 *sqlmath() function*

Sometimes it is of importance to generate $\#(min, max, sum)$ of a collection. That is precisely what $sqlmath()$ achieves :

```
l = [0,-1,-3,3,4,10 ]
#(m,M,s) = sqlmath(l)
/*  m := -3 ; M := 10 ;sum := 9  */
```

Obviously, the function takes anonymous function as argument, thus, given we have :

```
x = [ 2,3,1,0,1,4,9,13]
#(m,M,s) = sqlmath{$*$}(x)
/* [0, 169, 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:

```
#(m,M,s) = sqlmath{ $.value }( { 1:2, 3:4, 5:6 })
// [2, 6, 12]
```

10.3.6 *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 :

```
x = [ 2,3,1,0,1,4,9,13]
#(m,M) = minmax(x)
// [0, 13]
```

This also supports anonymous functions,thus :

```
students = [ {'roll' : 1, 'name' : 'X' } ,
             {'roll' : 3, 'name' : 'C' } ,
             {'roll' : 2, 'name' : 'Z' } ]
#(m,M) = minmax{ $[0].name < $[1].name }(students)
// [{roll=3, name=C}, {roll=2, name=Z}]
```

10.3.7 *shuffle() function*

In general, testing requires shuffling of data values. Thus, the function comes handy:

```
cards = [ 'A', 'B' , 'C' , 'D' ]
// inline shuffling
happened = shuffle(cards)
// happened := true , returns true/false stating if shuffled
cards
// order changed : [D, A, C, B]
```

10.3.8 *The random() function*

The *random*() function is multi-utility function. With no arguments it returns a SecureRandom :

```
sr = random() // A SecureRandom
```

random() function can be used in selecting a single value at random from a collection :

```
l = [0,1,2,3,4]
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 :

```
a = [0,1,2,3,4]
rs = random(a,3) // pick 3 items at random without replacement
// works on strings too....
rs = random("abcdefghijklmpon",10)
```

Given a seed data type, it generates the next random value from the same data type. Behold the behaviour:

```
// passing 0 generates a gaussian
gaussian = random(0)
// pass a long value to get next long value
r_long = random(1l)
// pass a double value to get next double value
r_double = random(1.0d)
// pass a float value to get next float value
r_float = random(0.1f)
// pass a big decimal value to get next big decimal value (-1,1)
r_bd = random(0.1b)
// pass a boolean value to get a random boolean value
r_b = random(true)
/*  Passing a BigInteger value in random generates
a next random BigInteger
at least 4 times the decimal digit size
of the input big integer in binary */
r_bi = random(2h)
```

The last behaviour can be used to generate random strings :

```
// generate a large random big integer
s = random(10h)
// convert it into string with a suitable base
rs = str(s,36)
```

CHAPTER 11

Java Connectivity

JVM is the powering force behind nJexl. This chapter illustrates various usage scenarios where nJexl can be embedded inside a plain Java Program.

11.1 HOW TO EMBED

11.1.1 *Dependencies*

In the dependency section (latest release is 0.2) :

```
<dependency>
  <groupId>com.github.nmondal</groupId>
  <artifactId>njexl.lang</artifactId>
  <version>0.3-ALPHA-1</version> <!-- or 0.3-SNAPSHOT -->
</dependency>
```

That should immediately make your project a nJexl supported one.

11.1.2 *Programming Model*

nJexl has a random access memory model. The script iterpreter 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 JexlContext type, which abstracts how local storage gets used. This design makes nJexl remarkably memory heavy, but at the same time pretty neat to integrate with any other JVM language, specifically Java. Any time a new scope is created, the *copy*() mechanism ensures that the parent context is suitably passed along. This is what really makes nJexl memory heavy, but fast in terms of access time of variables.

The nJexl JexlEngine creates an Expression or a Script from a string or a file.

An interpreter is created when one either *evaluate*() the Expression or *execute*() the script. To use variables, one must pass a *JexlContext* in the evaluate or execute function.

The result of these calls will always be an object. In case somehow the call does not return any value, it would return a *null* value.

11.1.3 *Example Embedding*

One sample embedding is furnished here :

```
import com.noga.njexl.lang.JexlContext;
import com.noga.njexl.lang.JexlEngine;
import com.noga.njexl.lang.JexlException;
import com.noga.njexl.lang.Script;
import java.util.*;

public class Main {
   public static void main(String[] args){
         // get an already nicely prepared JexlContext
         JexlContext jc = com.noga.njexl.lang.Main.getContext();
         // get a list of functions which would become
         // already imported namespaces
         Map<String, Object> functions
             = com.noga.njexl.lang.Main.getFunction(jc);
         // create an Engine
         JexlEngine JEXL = com.noga.njexl.lang.Main.getJexl(jc);
         // set the functions
         JEXL.setFunctions(functions);
         // whatever the arguments are, add to the variables
         jc.set(Script.ARGS, args);
         try {
             // import the script -- always use this new one
             Script sc = JEXL.importScript(args[0]);
             // execute the script
             Object o = sc.execute(jc);
             int e = 0;
             if (o instanceof Integer) {
                 e = (int) o;
             }
             System.exit(e);
         } catch (Throwable e) {
             System.exit(1);
         }
   }
}
```

11.1.4 *Default Namespaces*

The standard nJexl core comes with the default namespaces :

1. sys : 'java.lang.System'
2. set : SetOperations
3. str : 'java.lang.String'
4. math : 'java.lang.Math'

These are set using the *setFunctions*() call :

```
JEXL.setFunctions(functions);
```

11.1.5 *Default Imports*

To ensure that the import happens, we need to fix two things :

1. Add a namespace that is same as the import id.
2. Add a variable that is same as the import id.

Doing only the first one will only add the namespace, while doing only the second one would only add a local variable with the name of the import id.

```
Map f = JEXL.getFunctions();
// add the namespace
f.set('my_id' , my_obj);
// add the variable to the context
jc.set('my_id' , my_obj);
// set the functions map back
JEXL.setFunctions(f);
// import happened now
```

11.2 PROGRAMMING NJEXL

11.2.1 *Script Object Model*

A script object holds the classes defined in it, as well as the ScriptMethod objects. So, after creating a script, one can access the ScriptMethods by simply calling *methods*() function on the Script :

```
Script sc = JEXL.importScript("script.jxl");
Map<String,ScriptMethod> methods =  sc.methods();
```

11.2.2 *Function Model*

Once we have the functions map now, we can get a function by name :

```
Script sc = JEXL.importScript("script.jxl");
ScriptMethod m = methods.get("my_function") ;
```

However, to call this method one needs to pass an Interpreter :

```
Interpreter i = new Interpreter(
      JEXL, jc, strictFlag, silentFlag) ;
Object result = m.invoke(null,i,
                   new Object[]{ "This" , "are", "args" } );
```

And thus, you can execute any nJexl methods from Java.

11.3 EXTENDING NJEXL

There are two extension points for Java developers who are going to make their classes and functions nJexl ready.

11.3.1 *Overloading Operators*

First of them is the overloading of operators. The operators are defined in ScriptClassBehaviour. The specific operators are to be found under : *Arithmetic* and *Logic*.

Any class implementing these interfaces would be used by the nJexl engine as if the operators are overloaded for them. Thus, one can create Java objects which mixes up with objects of nJexl. Most of the time it won't be necessary, but if it is, there is a way to achieve it.

For example, one can create a complex number class, which would implement Arithmetic.

11.3.2 *Implementing Named Arguments*

If one wants to implement a method which would mix properly with nJexl named arguments, then one must expect a specific object *NamedArgs* found in the Interpreter.java.

```
/* name of the argument */
public final String name;
/* the value that was passed */
public final Object value;
```

Thus, once we know that the argument is *NamedArgs*, we can extract the value, and do the due diligence.

11.3.3 *Using Anonymous Arguments*

If one wants to implement the anonymous blocks like almost all the nJexl functions do, one needs to expect the first argument to be an *AnonymousParam* which is also found in the Interpreter.java. The standard way to process the argument can be said as :

```
/* ensure that the functions are always multiple args - nJexl
   default */
public static Obejct some_awesome_function(Object... args){
  AnonymousParam anon = null;
  if ( args.length == 0 ) { /* due diligence */  return null;  }
  if (args[0] instanceof AnonymousParam) {
            anon = (AnonymousParam) args[0];
            args = TypeUtility.shiftArrayLeft(args, 1);
  }
  // now in here, we may or may not have the anon set:
  if ( anon == null ){
     // try on with the normal way
     return someThing;
  }
  // in here I have anonymous param :
  Object context;
  Object item;
  Object id;
  Collection partial;
  anon.setIterationContextWithPartial(context, item, id ,
     partial);
  Object o;
   try {
       o = anon.execute();
       if (o instanceof JexlException.Continue) {
            if (((JexlException.Continue) o).hasValue) {
                partial.add(((JexlException.Continue) o).value);
              }
               continue;
        }
        if (o instanceof JexlException.Break) {
             if (((JexlException.Break) o).hasValue) {
                 partial.add(((JexlException.Break) o).value);
               }
                   break;
         }
         // in here, use o :
         doSomethingWith(o);
        } catch (Exception e) {
               o = null;
     }
}
```

CHAPTER 12

Examples with Interview Questions

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. Some are simply what I explicitly call IM, but that sales in the interviews for the kids. I have heavily against many of these as practical interview questions. But never the less :

If it is ones lot to be cast among fools, one must learn foolishness.
–Alexandre Dumas, The Count of Monte Cristo

12.1 Assorted impractical examples

This section has almost everything that qualifies as IM. It is good we start with them first, because IM, for many many Software people is a very pleasurable activity.

12.1.1 *A Game of Scramble*

I am sure you guys have know the game of jumbled up words. For example, someone gives you "Bonrw" and you need to say : "Brown" ! Suppose I tell 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 !

```
fi = lines('words.txt')
word_dict = lfold{
                // generate char array
                ca = $.toCharArray()
                // generate a key by sorting and concatenating
                key = str(sorta(ca),'')
                // add the word to the { key : list of words }
                if ( not ( key @ _$_ ) ) {  _$_[key] = list()  }
                _$_[key] += $
                _$_ // return the partial
              }(fi, dict() )
// now a jumbled up word exists ?
key = str( sorta( jumbled_word.toCharArray() ), '')
matches = word_dict[key]
write(matches)
```

12.1.2 *Find Anagrams of a String*

The problem is, suppose someone gave you a string : "waseem" 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 :

```
// create a set of words
words = set()
// get the sorted version in an array
letters = sorta ( word.toCharArray() )
// generate the args for permutations
args = array{ letters }( [0 : #|letters|] )
perms = join{ continue ( $ != letters )
              v = str($,'') ; continue( v @ words )
              // add the permutation to the words
              words += v ; false // no need to store anything
       }(__args__ = args )
// now words has all the anagrams
```

and this solves the problem. Notice that many advanced concepts are being used in the solution.

12.1.3 *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* :

```
// create a set for solutions
solutions = set()
// get the sorted version of the list
li = sorta ( li )
// generate the args for permutations
args = array{ li }( [0 : #|li|] )
perms = join{ continue ( $ != li )
              // we need the sum
              sm = sqlmath($)
              // sm[2] holds the sum
              continue( sm[2] != sum )
              $ = sorta($) // we need combinations
              v = str($) ; continue( v @ solutions )
              // add the combination to the solutions
              solutions += v ; false // store nothing
       }(__args__ = args )
// now solutions has all the solutions to the problem
```

12.1.4 *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 :

```
// create a set for solutions
solutions = set()
// get the sorted version of the list
li = sorta ( li )
// generate the args for permutations
args = array{ li }( [0 : #|li|] )
perms = join{ continue ( $ != li )
              continue( not P($) )
              $ = sorta($) // we need combinations
              v = str($,'#') ; continue( v @ solutions )
              // add the combination to the solutions
              solutions += v ; false // store nothing
       }(__args__ = args )
// now solutions has all the solutions to the problem
```

12.1.5 *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 0. 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 nJexl solution is :

```
AS = sorta(A)
BS = sorta(B)
e = 0.01 // say?
s = #|AS| // the size
close = ( index{ #|AS[$] - BS[$]| > e  }([0:s]) < 0 )
```

12.2 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.

12.2.1 *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 : $C_i = ({}_OC_i, {}_NC_i)$ where ${}_OC_i$ is the old field, while ${}_NC_i$ 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 :

$$t_o = \Pi_{{}_OC}(O_o)$$

and

$$t_n = \Pi_{{}_NC}(O_n)$$

and now, there are lists of it. Thus, now, these two list should be equal. Hence, in nJexl, here is the code one would write :

```
// old tuple list , generated by projecting old columns
otl = list{ o = $ ; lfold{ _$_ + '#' + o[$]}(OC, '')}(ol)
// new tuple list , generated by projecting new columns
ntl = list{ o = $ ; lfold{ _$_ + '#' + o[$]}(NC, '')}(nl)
// now compare 2 list of strings!
matches = ( otl == ntl )
```

12.2.2 *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 :

$$\forall x \in F \; ; \; P(x) = True$$

and this $F \subseteq C$. The question is how to test for filtering? Given we already have a predicate $P()$ defined :

```
filtered = ( index{ not P($) }(F) < 0 ) and 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 is a set, it is easy :

```
filtered = ( index{ not P($) and not $ @ C }(F) < 0 )
```

But what if, given C is not a set, but a list? That can be solved by invocation of multi set, or $mset()$:

```
l = [1,2,2,3,4,4,4]
ml = mset(l)
/* ml := { 1 : [1] , 2:[2,2] , 3:[1], 4:[4,4,4] } */
```

With this, the new verification declaration would be :

```
m = mset(C)
// replace with counts
mc = dict{ [ $.key, #|$.value| ] }(m)
filtered = index{
               continue( $ @ mc and P($) ){
                      mc[$]-=1 ; mc[$] < 0  }
               // control came here, failed
               true
            }(F) < 0
```

Bibliography

[1] Kanglin Li, Mengqi Wu, Sybex, *Effective Software Test Automation: Developing an Automated Software Testing Tool* . Sybex , 2004.

[2] Donald E. Knuth, *The Art of Computer Programming* . Addison-Wesley, 2011.

[3] Alfred V. Aho, Monica S. Lam, Ravi Shethi , Jeffery D. Ullman, *Compilers: Principles, Techniques, and Tools* . Prentice Hall, 2006.

Index

anonymous comparator , 106
atomic : block , 101
atomic : block : reverting states , 102
clock : error , 103
DataMatrix : key() , 96
DataMatrix : matrix(), 93
dict() : serialisation , 54
examples : anagrams , 118
examples : list closeness , 119
examples : Scramble , 117
examples : scramble , 117
examples : sublist predicate problem , 119
examples : sublist sum problem , 118
function : arbitrary no. of args , 64
function : composition operators, 69
java connectivity : adding to namespaces , 112
java connectivity : default imports , 113
java connectivity : function : anonymous argument , 114
java connectivity : Function Model , 113
java connectivity : functions : named argument, 114
java connectivity : operator overloading , 114
java connectivity : programming model , 111
java connectivity : sample embedding , 112
java connectivity : Script Object Model , 113
lines() , 82
logo : meaning , viii
matrix() , 91
nJexl : about , vii
nJexl : challenge from Java Land, vii
nJexl : history , vii
nJexl : thanks , viii
operator : ~, 27
operator : regex : match , 26
operator : regex : not match , 26
str : collections , 51
str : linearise objects , 51
str : replacing toString() , 51
table comparison , 94
table comparison : key , 95
table comparison : key : non unique , 95
table comparison : O(n) , 94
Von Neumann Architecture, 71
xml : json() , 91

apache jexl, vii
arithmetic : chorono, 49
array(), 23
assignment, 15
atomic, 101
atomic : atomic() , 102
atomic : mutable decrement , 102
atomic : mutable increment , 102

Beizer, vii
bool(), 43
business needs, 1
bye(), 105

cat, 85
Chaitin Solomonoff Kolmogorov complexity, 3
char(), 44
Chatin Solomonoff Kolmogorov Complexity, 72
chrono : subtraction, 49
clock, 102
collection : slicing , 42
collections: set() , indexOf(), lastIndexOf(), select(), list() , 78
Combinations, 41
comments, 16
Comprehensions, 35
Comprehensions : array(), list() , 35
Comprehensions : dict(), 36

Comprehensions : set(), 35
Conditionals, 18
Conditionals : else, 18
Conditionals : else if, 19
Conditionals : if, 18
Conditionals : jump, goto, 19
conditions : avoiding, 75
CSK complexity, 3
currying : back tick operator, 74
currying : referencing, 74
currying : reflection, 74

DataMatrix, 91
DataMatrix : aggregate() , 96
DataMatrix : c() , 92
DataMatrix : columns, 91
DataMatrix : data access by column , 92
DataMatrix : data access by row , 92
DataMatrix : diff(), 97
DataMatrix : project, c() , 92
DataMatrix : rows, 91
DataMatrix : select condition , 93
DataMatrix : select transform , 93
DataMatrix : select, select() , 93
DataMatrix : tuple() , 92
date, 47
date(), 47
DEC(), 46
design, 10
dict(), 24
double(), 46

empty(), 27
enum(), 58
Equivalence Class Partitioning, 75
error(), 103
examples : generic result comparison, 120
examples : verifying filter results, 121

factorial, 67
Fibonacci, 70
FizzBuzz : conditional, 76
FizzBuzz : declarative, 77
float(), 46
Flow Control : break, 37
Flow Control : continue, 37
Flow Control : continue, break , 36
Flow Control : usage of continue, 38
fold : factorial, 79
fold : Fibonacci, 79
fold : index, 80
fold : min,max, 80
fold : rindex, 80
fold : select, 80
fold : set, 80
fold : sum, 79
fold functions, 79
fopen(), 85
fopen() : open modes, 85
fopen() : reading, 85
fopen() : writing, 85
format : date and time, 47
function : __args__ , 64
function : anonymous, 63
function : argument overwriting, 66
function : as parameter, lambda , 68
function : closure, 67
function : composition, 68
function : default parameters, 64
function : eventing , 70
function : eventing : after, 70
function : eventing : before, 70
function : excess parameters, 65
function : explicit, 63
function : implicit, 63
function : in a dictionary, 76
function : nested, partial, 68
function : Peano Axioms, 66
function : recursion, 66
function : recursion using composition, 70
function : unassigned parameters, 65
functional approach, 2
functional style, 63
functional style : application, 3
functional style : tenet, 2
functions, 20
functions : anonymous, 21
functions : calling, 20
functions : define, 20
functions : named args , 64

global variables, 20
Gödelization, 71

hash : base 64, 106
hash(), 106
Hello Curry, 1
Hello, World, 14

identifiers, 15
Imperative, 1
import, 57
import : enum, 58
import : function call, 60

import : inner class, inner enum, 59
import : java, 57
import : load(), 58
import : nJexl Script, 57
import : relative, 60
import : script as function, 60
import : static field, 59
index(), 28
inspect(), 53
instant, 47
instant(), 48
INT(), 45
int(), 44
isa : arr, 53
isa : array, 53
isa : chrono, 53
isa : error, 53
isa : list, 53
isa : map, 53
isa : num, 53
isa : Q, 53
isa : regex matcher, 53
isa : set, 53
isa : Z, 53
item rewriting, 29

join(), 40
json, 85
JSONPath, 86

lfold(),rfold(), 79
lines(), 84
list(), 23
logic : 2nd order, 25
logic : higher order, 25
long(), 45
Loops, 19
Loops : for, 19
Loops : for, conditioned, 20
Loops : for, range, 19
Loops : while, 19

maven, 111
Minimum Description Length, 3, 72
minmax(), 22, 109
mset(), 121
multiple assignment, 104
multiple assignment : splicing left, right, 104
Multiple Collection Comprehension : join(), 40
multiple return : error, 104
mutability, 24

namespace, 57
namespace : my, 60
new(), 58
njexl : philosophy, 9
njexl : repl, 12
njexl : tenets, 9

operator : === , 52
operator : at , 26
operator : cardinality , 27
operator : compare, chrono , 48
operator : isa , 52
operators, 16
operators : Arithmetic, 16
operators : Comparison, 17
operators : Logical, 17
operators : Ternary, 18

partition(), 29
permutation, 65
Permutations, 41
predicate, 7, 40
Predicate Logic, 25
Predicate Logic : for all : select() , 29
Predicate Logic : there exist : index() , 28
Predicate Logic : there exist : rindex() , 28
Predicate Logic : there exists, 26
project, 41
python, vii

random() : random value with types , 110
random() : select many, 110
random() : select one, 109
random() : strings , 110
range, 23
range : char , 77
range : chrono interval format, 78
range : date, 77
range : symbol, 78
range(), 77
range: list(), reverse(), inverse() , 78
read(), 81
read() : http get, 81
read() : http get : url generation , 82
read() : url timeout , 81
reflection : field access, 55
reflection : property bucket, 55
regular expressions, 26
relation between exists and forall, 25
reserved words, 15
restful api : get, 82

scala, 9
Searching for a tuple, 41
select(), 29
send(), 83
Set Algebra, 30
Set Algebra : Equals, 30
Set Algebra : Intersection, 30
Set Algebra : Minus, 30
Set Algebra : Subset Equals, 30
Set Algebra : Subset,Proper, 31
Set Algebra : Superset Equals, 31
Set Algebra : Superset,Proper, 31
Set Algebra : Symmetric Difference, 30
Set Algebra : Union, 30
Set Operations : Collection, 31
Set Operations : there exist : Collections, 34
Set Relations : Collection, 33
set(), 23
short(), 44
shuffle(), 109
Sieve of Eratosthenes : declarative, 39
Sieve of Eratosthenes : imperativel, 39
size(), 27
SOAP, 83
sorta(), 108
sortd(), 108
sorting, 108
sorting : custom comparator , 108
sqlmath(), 108
str : date,time,instant , 50
str : float,double,DEC, 50
str : int,INT, 50
str(), 49
Sublime Text, 12
system(), 99

tenet : good code, 9
testing : validations, 4
thread(), 100
time, 47
time(), 47
tokens(), 105
try(), 104
Tuple Operation : Starts With, Ends With, 34
Turing Machine, 71
type(), 51

until(), 100

Vim, 12

wc : word count, 84
where(), 29
write(), 82
write() : file , 82
write() : formatting, 83
write() : url : post, 83

xml, 87
xml : attribute, 89
xml : element(), 90
xml : elements(), 90
xml : encoding, 87
xml : exists(), 90
xml : pojo, 89
xml : xpath(), 90
xml : xpath() default , 91
xml(), 87
xml: serialisation, 87