The

nJexl

Programming Language

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

Nabarun Mondal

Contents

Preface

nJexl initially was a continuation of Apache Jexl project. That project was not active for 17 months, and I needed to use it, so I forked it. The reasons 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.

An Open Challenge From Java Land

This actually was an open challenge:

A noted PhD from Sun, read this 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. anonymous functions) out of the box, and every function by design can take functional as input. There are tons of in built methods which uses functional.

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.

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 systems 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.

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 the very problems 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 is 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 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 :

```
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.

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*:

"Validation should be like SQL"

Being imperative or OOP like would be to overtly complicate the test code. Thus, we, moved to declarative paradigm. This post 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.

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 is not a surprise, that is an example of convergent evolution.

2.1 njexl philosophy

My experience in Industry is aptly summarized 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. To boldly go where no developer has gone before - attaining Nirvana in terms of coding; - get out of the cycle of bugs and fixes by writing scientific code (see Minimum Description length).
4. Good code is, once written, forever forgotten, i.e. : limiting to 0 maintenance.

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 1 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 – 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 an 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>0.2</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
```

FIGURE 2.1 – Using Sublime Text.

FIGURE 2.2 – Using Vim (MacVim).

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!

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

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. See here for a list of keywords.

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 parameters, 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
```

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.

4.1.2 *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
```

4.1.3   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.

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

This pose a delicate problem. Given we have can say if *"y in container x"* or not, what if when asked a question like : *"is there a y in x such that f(y) is true"* ?

  Notice this is the same problem we asked about sorting. Is there an element (y) in x, 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.

4.1.5 *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)
```

4.1.6 *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
```

4.1.7 *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* .

4.2   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 :
$$U_{AB} = A \cup B := \{x \in A \text{ or } x \in B\}$$
   b. Set Intersection is defined as :
$$I_{AB} = A \cap B := \{x \in A \text{ and } x \in B\}$$
   c. Set Minus is defined as :
$$M_{AB} = A \setminus B := \{x \in A \text{ and } x \notin B\}$$
   d. Set Symmetric Difference is defined as :
$$\Delta_{AB} = (A \setminus B) \cup (B \setminus A)$$
   e. Set Cross Product is defined as :
$$X_{AB} = \{(a,b) \mid a \in A \text{ and } b \in B\}$$

3. The following relations are defined:
   a. Subset Equals :
$$A \subseteq B \text{ when } x \in A \implies x \in B$$
   b. Equals :
$$A = B \text{ when } A \subseteq B \text{ and } B \subseteq A$$
   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
```

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
      break
    }
    // do something else
}
```

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 ){ write('FizzBuzz') }
    continue( n % 3 ){ write('Fizz') }
    continue( n % 5 ){ 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 (https://en.wikipedia.org/wiki/Predicate_(mathematical_logic)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 :

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

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.

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
2. short
3. char
4. int
5. long
6. INT : Java BigInteger

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
```

Bibliography

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

[2] Raj Jain, *The Art of Computer Systems Performance Analysis* . Wiley , 1991.

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

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

[5] David Stutz , Ted Neward , Geoff Shilling *Shared Source CLI Essential* . O'Reilly Media, 2003.

Index