

The background features a complex network of thin grey lines and dots, primarily concentrated on the left side, forming a web-like structure. Scattered across the entire background are numerous triangles of various sizes and orientations, some with solid outlines and others with dashed or dotted outlines. The overall aesthetic is minimalist and technical.

THE BUND LANGUAGE

Defining the language.

Version 1.1.0



INTRO

Brief introduction
Into the Bund language

01

NAMESPACE

Defining and operating
namespaces

02

DATA

Placing and manipulating
data in stacks

03

TABLE OF CONTENTS

04



FUNCTIONS, OPERATORS, LAMBDA.

Processing the data

05

LOOPS AND CONDITIONALS

Branching your code

06

FIN

Last, but not least

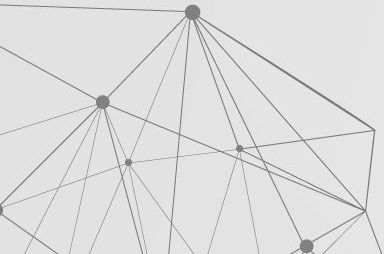
01

INTRO

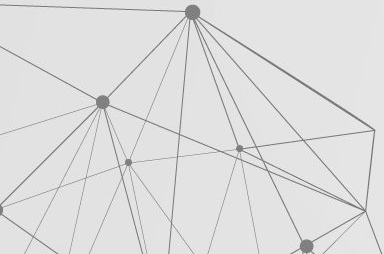
Brief introduction into the Bund language.



Bund is an interpreted high-level general-purpose programming language. It is shared a features from different “ancestors” while not being a next generation of any of them. Out of many BUND ancestors, I can specifically mention FORTH, Lisp and Haskell (as very distant relative).



Like a FORTH, BUND is stack-based, but it is different from FORTH in respect that it is not a procedural language. It is functional. It is also different in how BUND stack machine works. Like LISP and Haskell (remember, that one is very distant relative, but relative nevertheless), BUND is functional. It supports lambda calculus with named and anonymous lambdas. Lambda functions in BUND are functionally strict, they are taking only one parameter and always return a value. And we will discuss them in a separate tutorial. In addition to functions, BUND offers separate concept of “operators”, which essentially functions, but taking two parameters and returning a value at all times. Like in Haskell, BUND offers a “variable immutability” and unlike in Haskell and LISP, BUND does not have variables. Unlike LISP, BUND virtual machine operates with Stack not with List and this BUND’s feature differs it from Haskell as well.



02

NAMESPACE

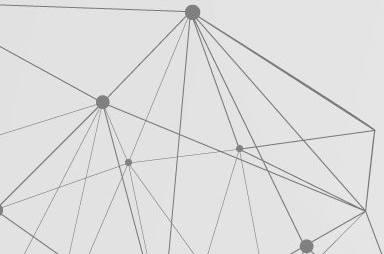
This is first and foremost concept of the BUND



BUND is a stack language. All operations that are performed, they are performed over data stored in the stack. And if you've ever heard the combination of letters "RPN" translated as "Reverse Polish Notation", you shall know that you will use RPN, sometimes known as "postfix notation" a lot. If you never experience RPN, then it is not that difficult to master. In nutshell, instead of "operand -> operator -> operand" or "function(parameter)", you will use "operand operand operation" or "parameter function" respectfully.

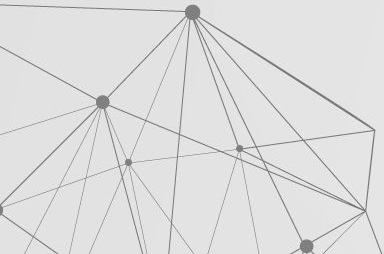
There are some very good reasons to adhere to a postfix notation. First, it's allow us to be a very expressive about what we doing and what is our data. Visually, data is separated from the action. Second, you are in control of precedences and evaluation order. There is no parentheses (well, there are but they are playing a totally different role), there is no operation precedences. How you define order of evaluation is that how BUND engine will execute them. No defaults, no mistakes. Third, RPN makes shorter expressions, which equals "less errors, more efficiency"

And finally, it is fit very nicely with architecture of all stack-based languages, BUND included.



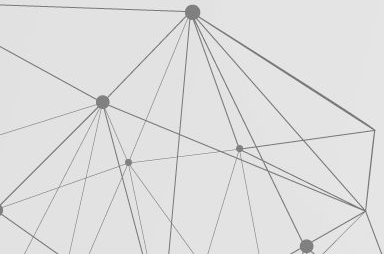
There is no “=” command. It is an equal operator, returning TRUE if operands are equal, FALSE otherwise. But there is not such operation as assigning some value to some variable. There is no variables in BUND. All data is placed in stack according to the stack mode. Stacks are double-ended queues, and by default, you are pushing data to the end of the stack. There are operators which switched stack mode and you can get back and force pushing and pulling data from ether side of stack. Once you’ve placed data to the stack it is immutable. You can generate a new data, based on the old one and place it to the stack, but you can not change existing data that you are stored in stack.

There is no “data definition sections” of any kind in BUND language. Once declared, piece of data goes to the stack.

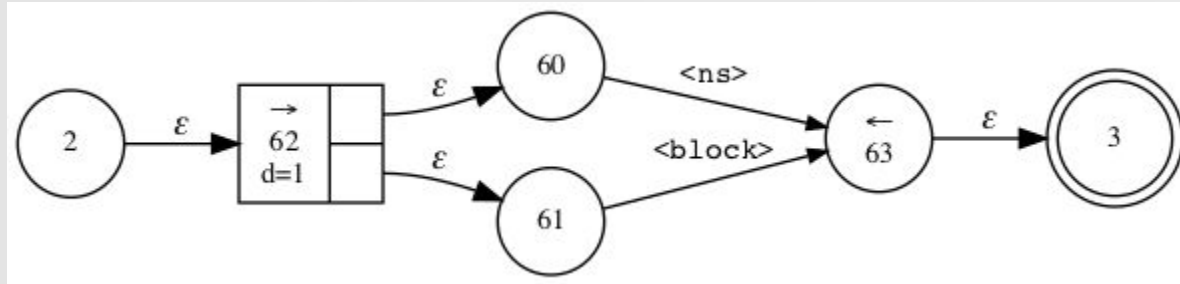


There is a multiple stacks in your BUND application. The only limit is your memory, as all stacks are in-memory data structures. Stack and other associated data is called "Namespace". So Stack + Name + This stack specific data = Namespace. There are named and anonymous Namespaces, although while anonymous namespace do have a name, this name is randomly generated GUID. There is system function "name" using which you can get an access to a current Namespace name.

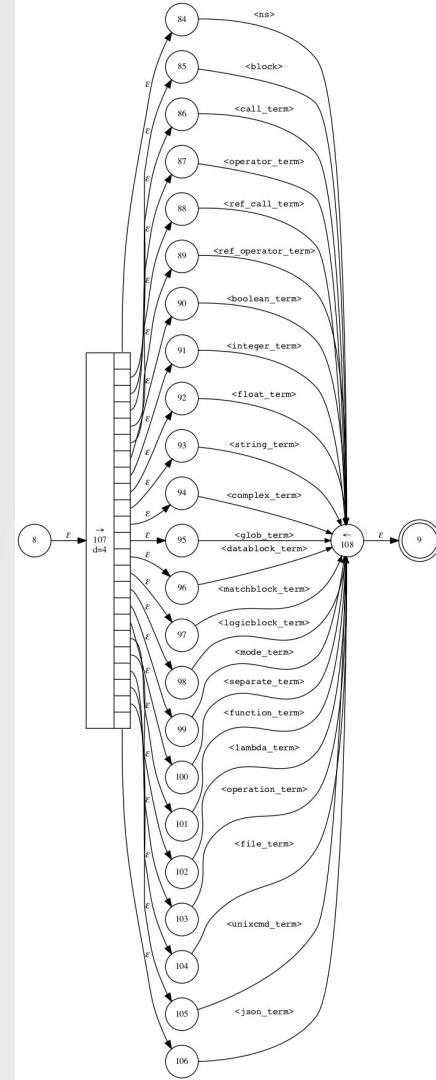
So, besides the GUID name or user-specified name, there is only one difference between named and anonymous namespaces. Anonymous namespace automatically removed when exiting and named spaces are stateful and preserved between re-entries. Before I illustrate this idea, let me introduce how you can create a named Namespace.



So, what the BUND program is made of ? it is made of definitions of ether named, and anonymous namespaces. No other data elements allowed on “root” level.



Next, let's formally review of what is consisting as the "term" that can be the part of the namespace. Any data or function call, operator call, function and operator references, matching block, data block, file, json, glob objects, anonymous and named namespace declarations, lambda functions and references to a lambda functions that are recognizable and defined in embedded BUND grammar are the valid terms. We will discuss more of the data and functions/operators later on.

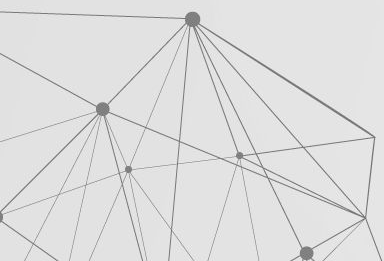


Named namespace are defined by it's header where the name of the namespace are placed between "[" and ":". Name could be a collection of alphanumeric characters and forward slashes. Named namespace terminated with ";;" - double semicolon. This command passes control to previous namespace and if you are in the "root" or the first namespace which is currently last, this will cause application termination.

Namespaces could be called in hierarchy, but information about them are stored in flat list. In our example, namespace "main" and namespace "another/namespace" are equal. "another/namespace" is not a child of "main". And with named namespaces, you can enter and exit them multiple times. They are stateful and named namespaces (and stacks) are preserved up to application termination.

And I will remind you, each namespace defines own stack. When you are entering from "main" to "another/namespace", you are switching default (or current) stack from "main" namespace to "another/namespace", when you reach ";;" you stack is going back to be stack of "main" namespace.

Please do allow me to illustrate.



```
1 [ main :  
2   [ another/namespace :  
3   ;;  
4 ;;
```

We are entering to namespace "main"

```
1 [ main :
```

And we are placing number 3 to the "main" stack

```
2 3
```

Then we are entering to "stack1", which is created for us. "Main" becomes "previous namespace"

```
3 [ stack1 :
```

And we are placing number 21 to the "stack1" stack. While we are in stack1 namespace, we are totally disconnected with "main"

```
4 21
```

We are exiting from "stack1" and as "main" our previous, we are going back to "main"

```
5 ;;
```

And we are placing number 2 to the "main" stack. "Main" stack so far is unaware of activity in "stack1"

```
6 2
```

Then again we are entering to "stack1". We already know about it, so we are having it back.

```
7 [ stack1 :
```

And now, we are placing number 2 not to "main" but to "stack1", which already do have 21.

```
8 2
```

Back to the "main" namespace, mate !

```
9 ;;
```

Back to "stack1" namespace, where we are having 21 and 2!

```
10 * println
```

We executing multiplication (*). Since we are in "main" stack, multiplication will be over 3 and 2 and result is stored in stack. println shall print 6.

```
11 [ stack1 :
```

```
12 * println
```

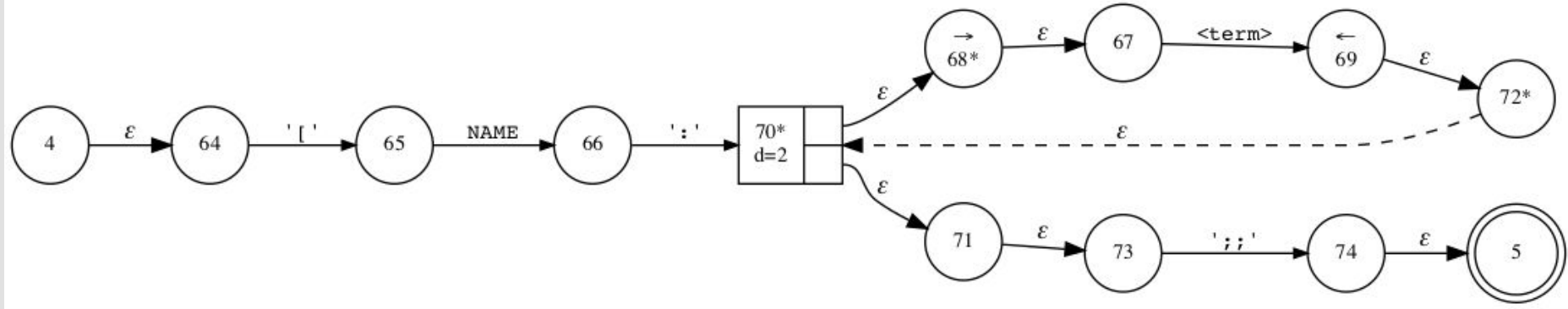
Exit from "stack1", okay. And then exit from "main", which is last stack in the "stack of stacks" and application is terminated.

```
13 ;;
```

We executing multiplication (*), but with data stored in "stack1". Since 21 2 * will be 42, this value will be stored in stack and println will print it.

```
14 ;;
```

Before continue, please review this formal definition of the named namespace. It is consists of header which defines a name, list of terms and termination as “..”

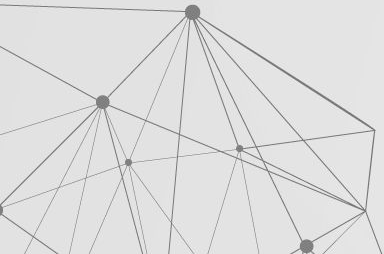


Next, let see how difficult will be to define an anonymous namespace. Easy. Everything placed between "(" and ")" will store and operate over anonymous namespace. When evaluation reaches ")", anonymous namespace deleted. Unlike named, anonymous namespace is not stateful.

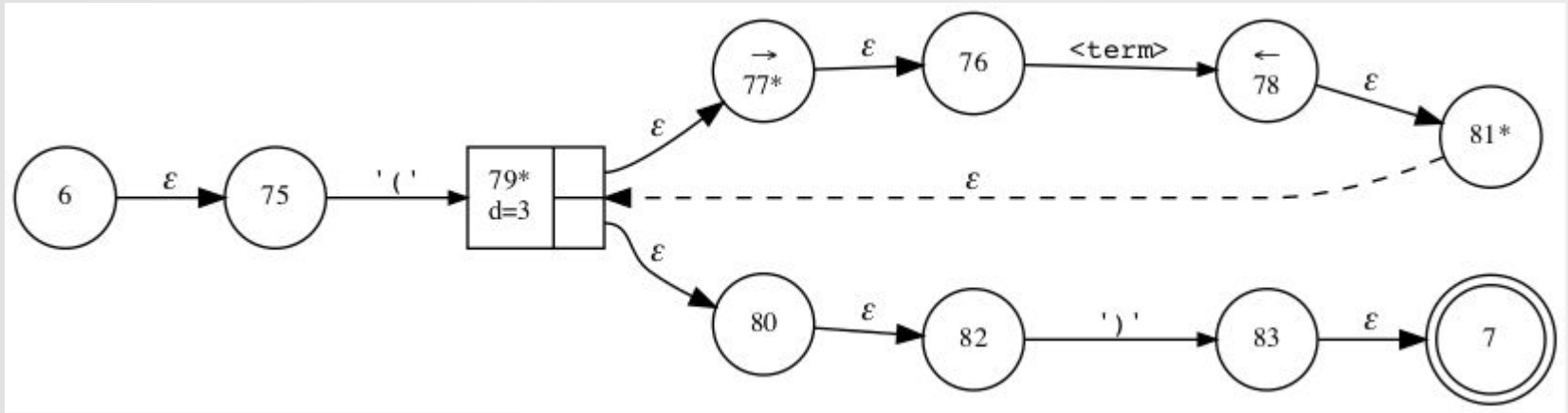
You can mix and match anonymous and named namespace. You can set anonymous namespace as "root" namespace. Or you can enter anonymous namespace inside named. If you have a cascade of anonymous namespace entries, new anonymous namespace will be created every time when "(" will be evaluated.

```
1 (
2   [ main :
3     [ another/namespace :
4       ;;
5     ;;
6   )
```

```
1   [ main :
2     [ another/namespace :
3       ( )
4     ;;
5     ( ( ) )
6   ;;
```

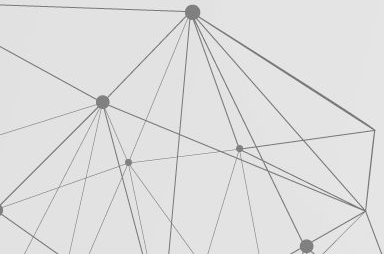


And let's look at the formal definition of the anonymous block. It is simple: space separated list of the terms between "(" - beginning of the namespace and ")" - end of the namespace.





Every time, when “)” is evaluated, anonymous namespace is deleted. All functions, lambdas, operators and data that you define inside that namespace are lost. There is no way to recover it when “)” is reached. But not all is lost. You can return value from namespace to previous namespace or to any namespace. Let’s talk about this.



Function "\$" when called, will take data value stored on top of the stack, and push it to the stack of previous namespace and not return it back to the stack. So, it is a destructive return. Function "\$\$" when called, will take data value stored on top of the stack, push it to the stack of previous namespace and keep it in original stack. So, it is non-destructive return. You can use "\$" and "\$\$" returns as many times as needed. Let's play with destructive returns first. Let's run this one-liner.

If we do not return, result of computation will be lost at this point

But now, we have number 2 in the stack of this namespace.

We are performing another computation in second and also anonymous namespace

```
( ( 1 1 + $ ) ( 2 0 1 + $ ) * println )
```

We are performing computation in one anonymous namespace

and pushing result from the stack of that namespace to previous stack.

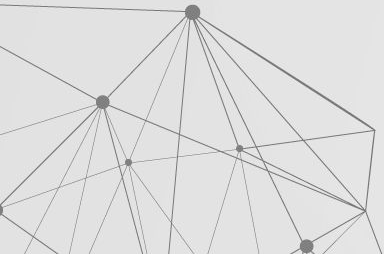
and pushing result from this stack to the previous stack.

We did not defined values for the mathematical operations. They were returned to this stack as result of computation in two anonymous namespaces.

What will be the
outcome of this
computation ?

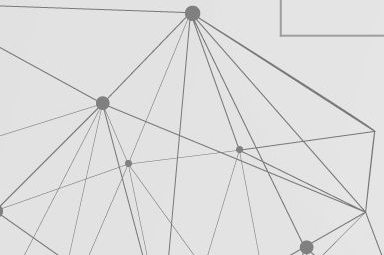


```
( ( 1 1 + $ ) ( 20 1 + $ ) * println )
```



When “\$” and “\$\$” returning value from current stack to previous stack, there are three operators that can return data to or from any non-anonymous, i.e. named namespace.

Type	Name	Description	Top of the stack	Previous
operator	\$_	Move value to named namespace	Value	Name of namespace
function	_\$	Move value from top of the stack of named namespace	Name of namespace	
function	__\$	Copy value from top of the stack of named namespace.	Name of namespace	



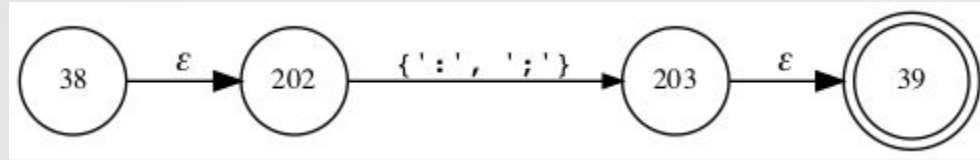
In this example, we are pushing the name of the namespace to which we will push some data, then the data itself. Function “\$ _” will take this data and move it to the namespace whose name defined by previous value in the stack. Note, if namespace doesn't exists, it will be created.

What do you think will be printed on the console ?



```
1 (
2   'main'
3   42 $ _
4   [main:
5       println
6   ;;
7 )
```

You can switch stack mode. By default, data is pushed to the end of the stack, if you send command “:”, BUND virtual machine will be globally set to push data to the head of stack. Any stack for that matter. Command “;” set the stack mode to “push to the end”.



03

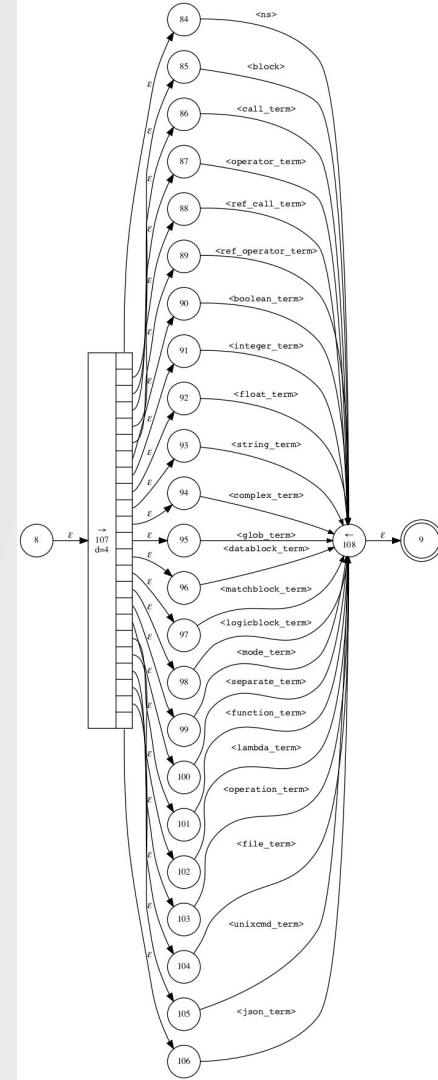
DATA

Placing and manipulating data in stack.

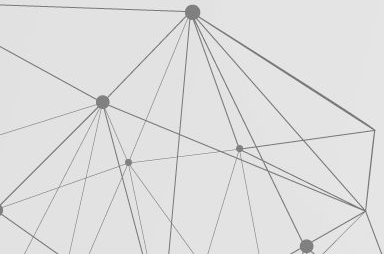
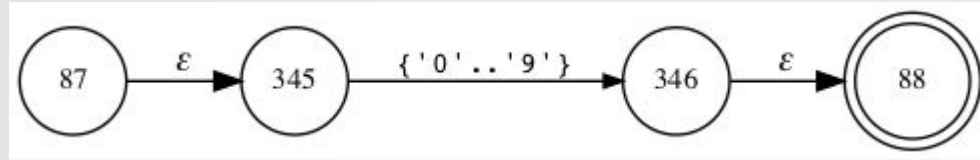


Data is a part of the terms, that stored in stack. So, very important difference between execution and storage is: execution is manipulating with data stored in stack, and data objects as defined, they just go to the stack according to the stack mode. Every time you define some data, it is stored in the stack of appropriate namespace. Data can not be defined outside of namespace. BUND is dynamically typed language. This means that you do not have to define datatype when declare the data. BUND will make an educated guess, about this data type. There are also no different operator or functions for a different data types. BUND will try to automatically accommodate a proper algorithm for each data placed on the stack.

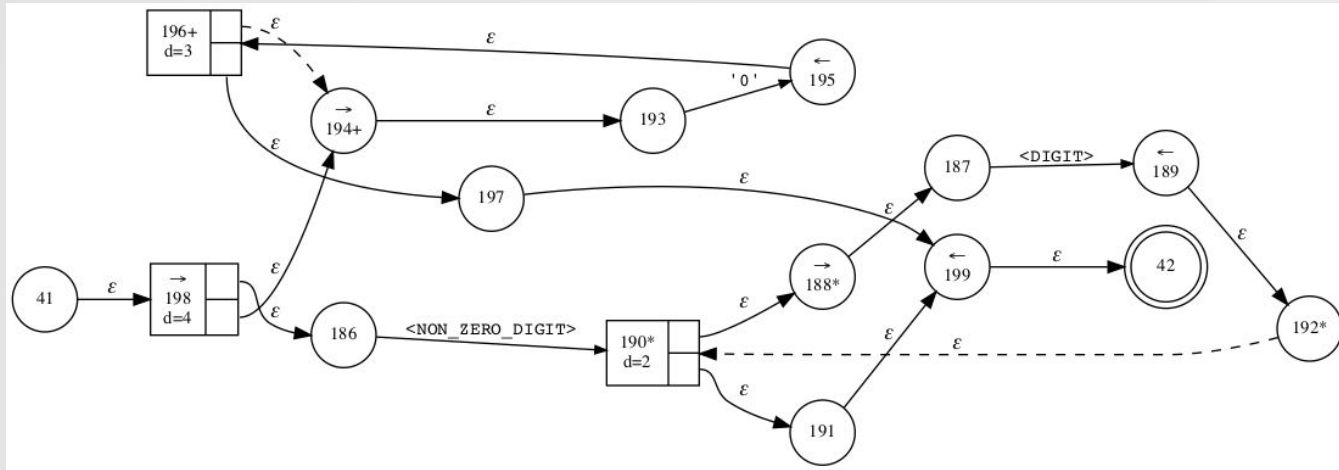
Let's review which data types BUND supports.



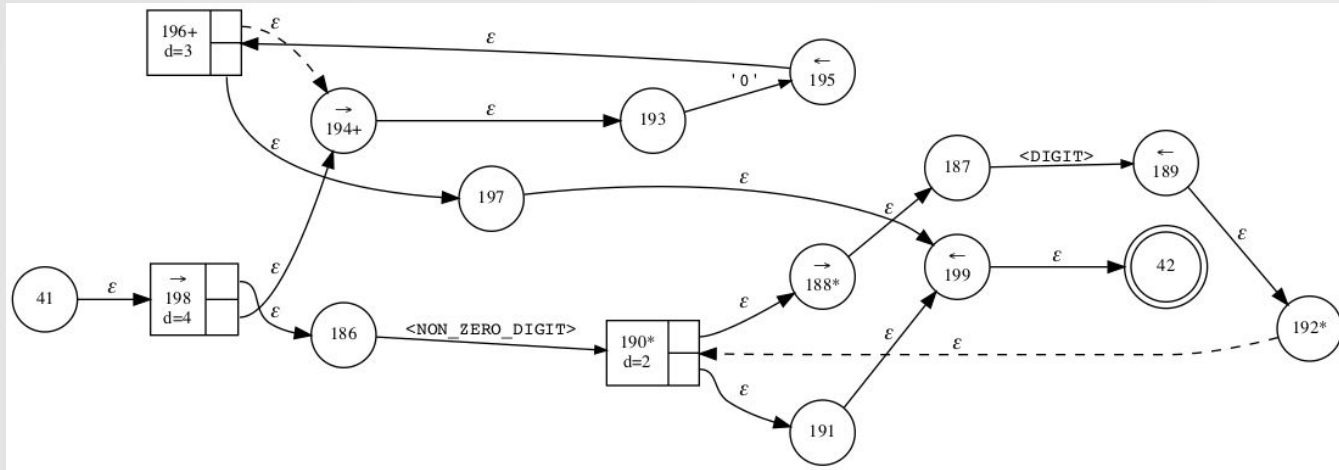
But first, let's define digits. They are your normal numbers from 1 to 9. Computers, are nothing but big calculators, still.



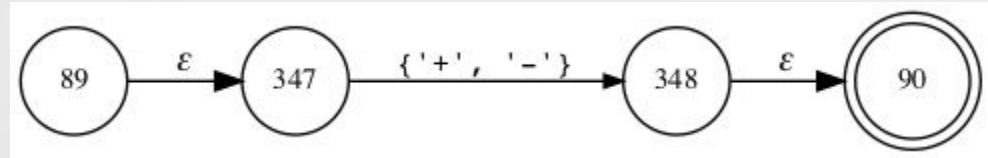
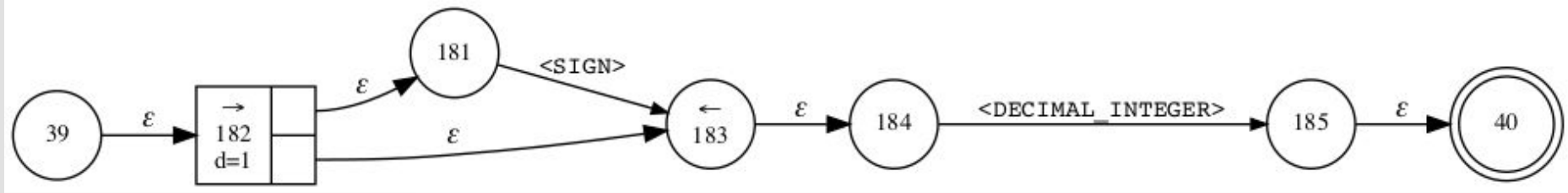
Decimal integers are including '0' - zero and non-zero digits.



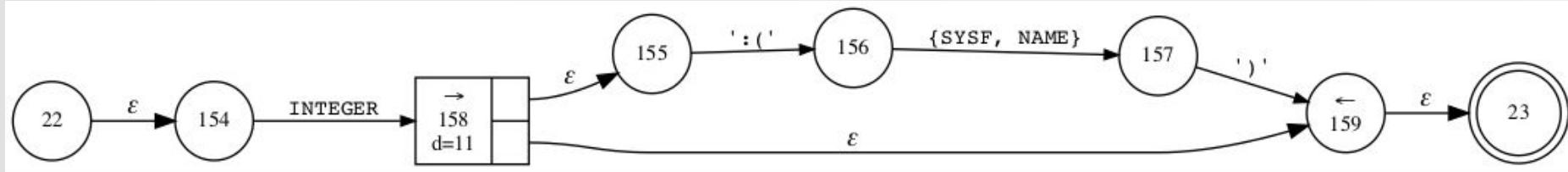
Decimal integers are including '0' - zero and non-zero digits.

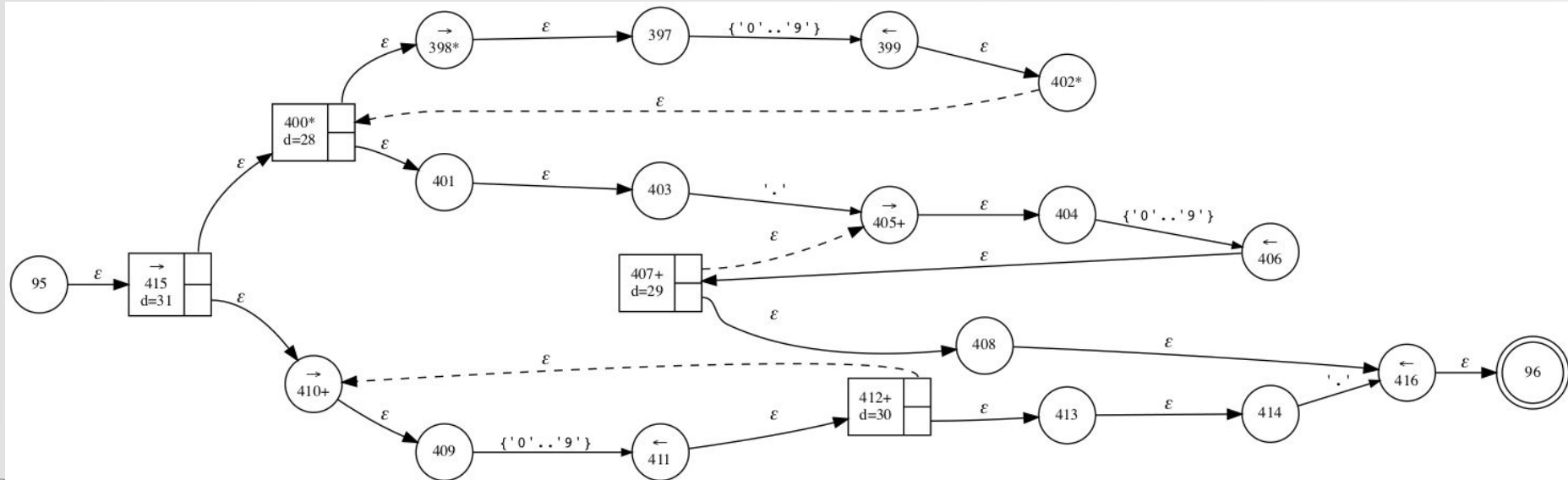


And let say integer, the data type consisting of “decimal integers” with optional plus or minus sign.

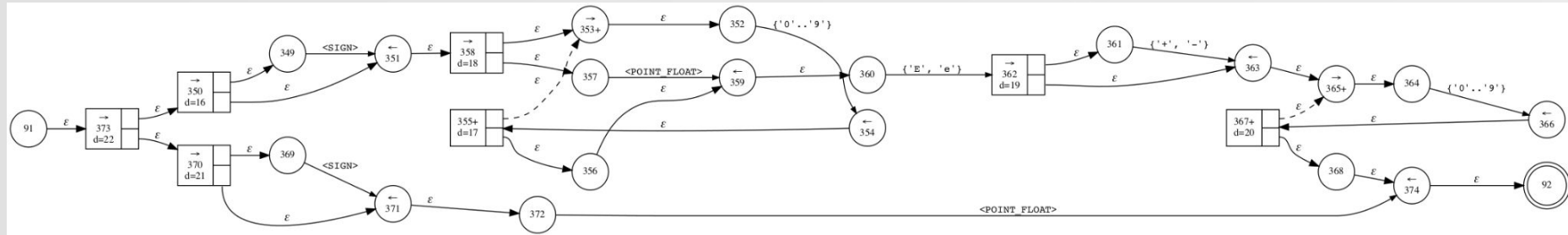


Integers as all other data types are monadic in nature and you can assign a functor to an integer. Functor is high-ordered function, applied to parameterized data type (the integer in our case). Functor name are specified between ":" and ")" attached to the end of the integer value. Functors are optional, if not specified, the actual value will be stored in stack. If specified, the outcome of evaluation of functor will be placed in the stack. More about functors later on.

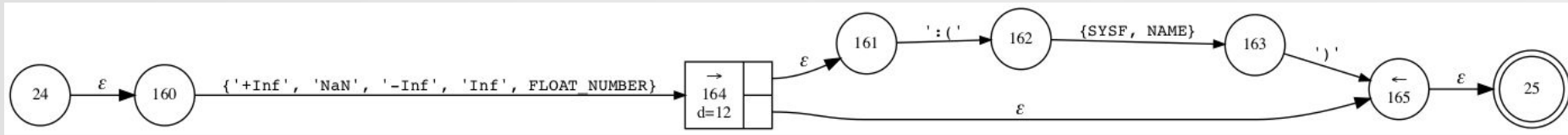




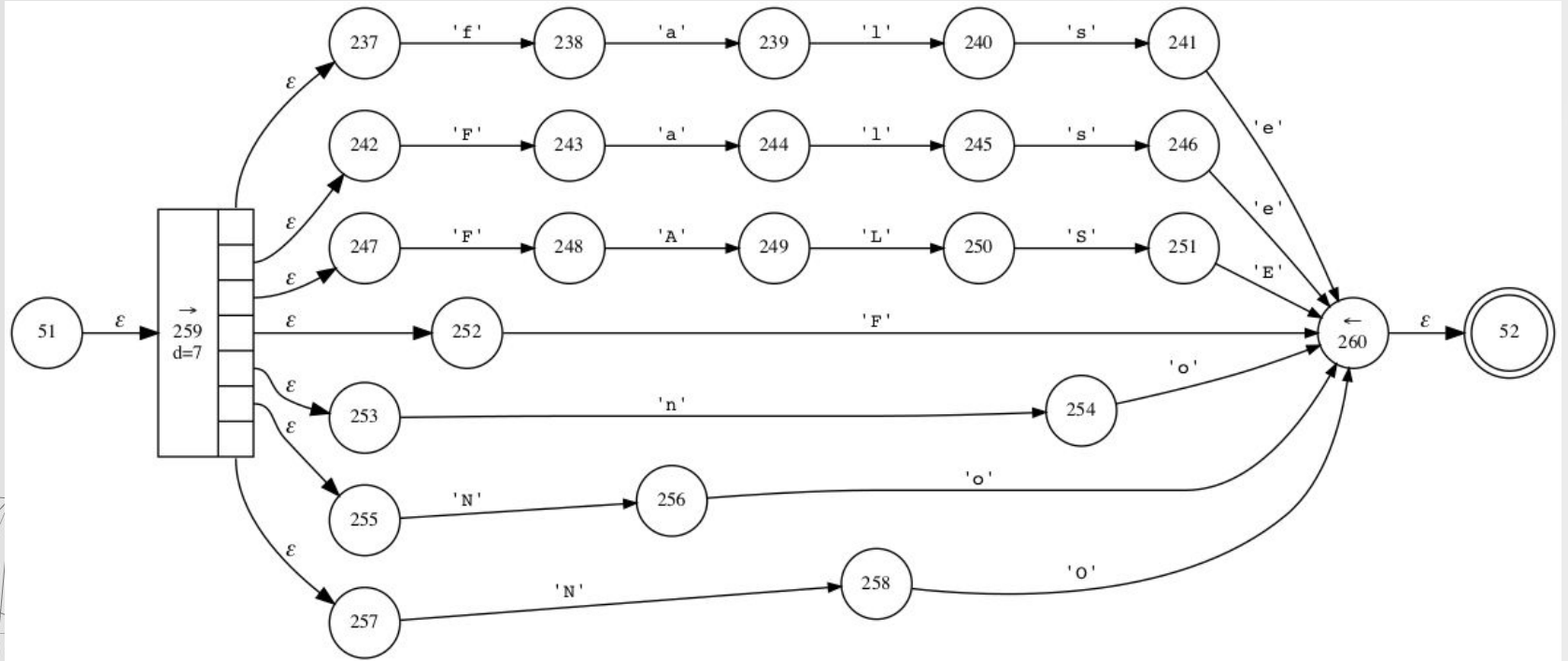
BUND supports declaration of floating point number not only through “dot” notation, but also using “scientific” notation as power of 10.



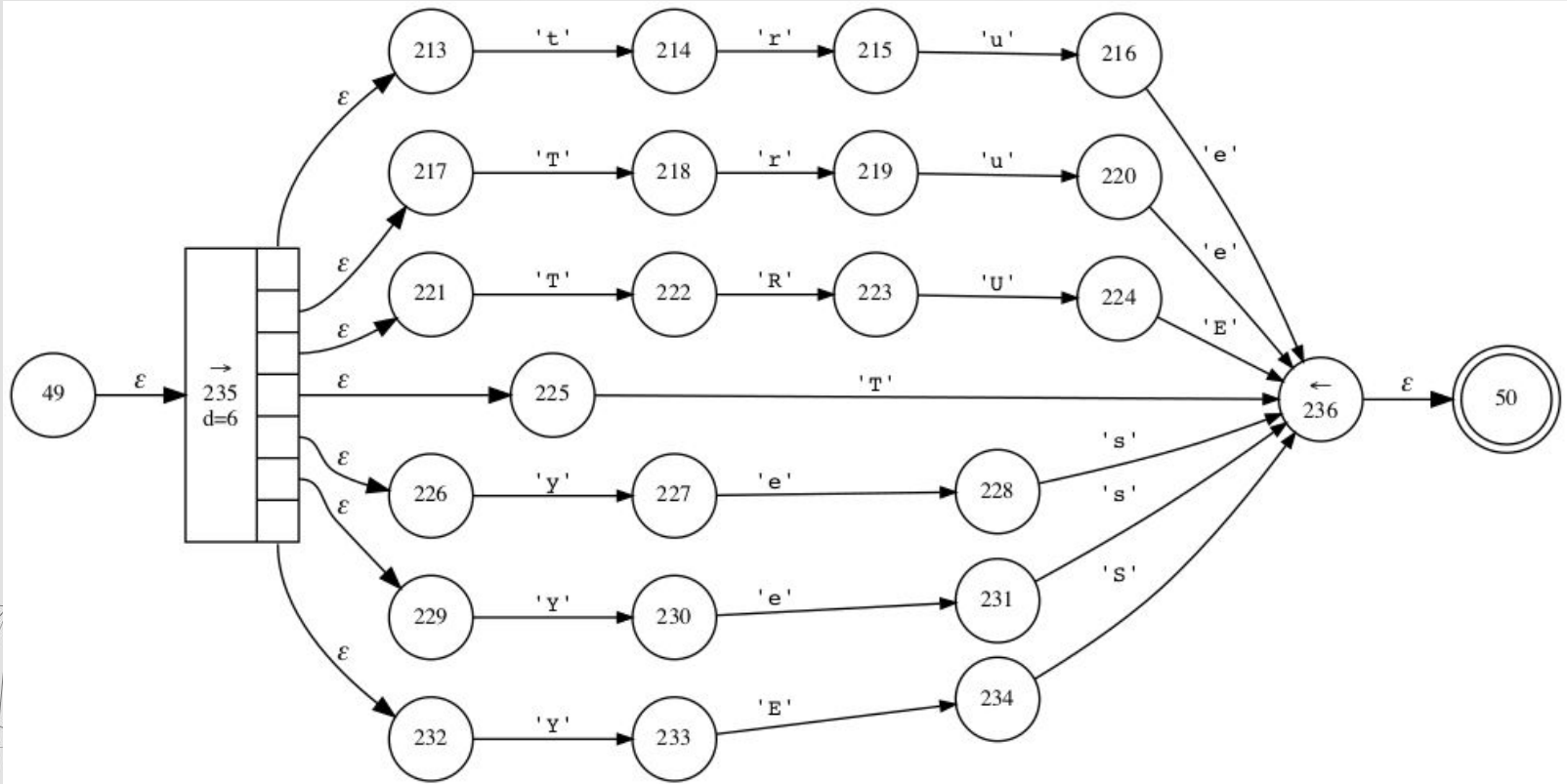
And in the end, definition of the Float point number including both “dot” and “scientific” notations, as well as special case of the floating points: +Inf for a positive infinity, -Inf for a negative infinity, Inf for a infinity without specification of the sign and NaN - not a number. You can specify functor for the float same way as you do for the Integer, between “:(“ and “)”.



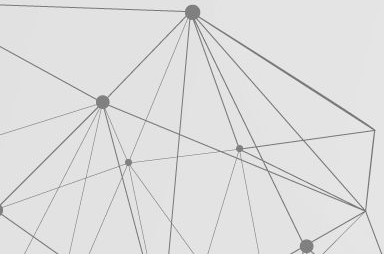
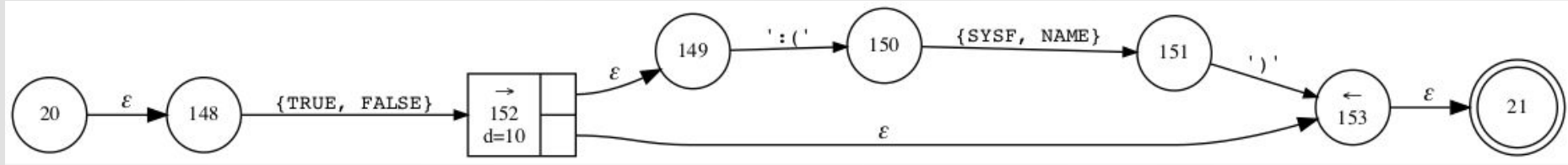
This is a pattern for the FALSE logical value. Basically any of the F, False, false, FALSE, no, No, NO will be treated as FALSE term.



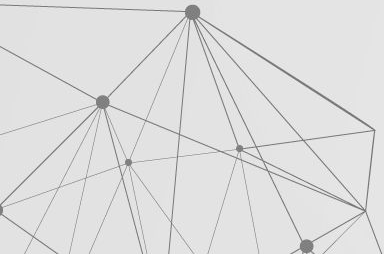
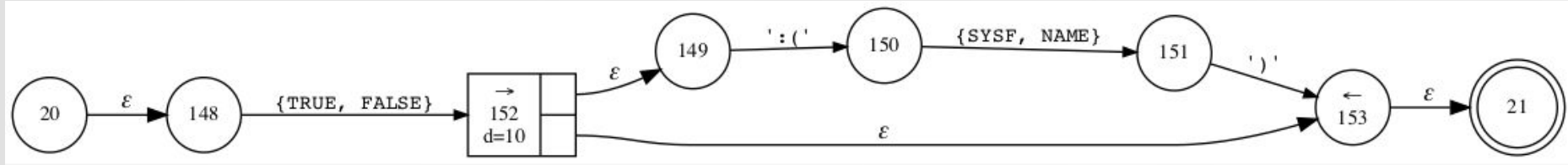
This is a pattern for the TRUE logical value. Basically any of the T, True, true, TRUE, yes, Yes, YES will be treated as TRUE term.



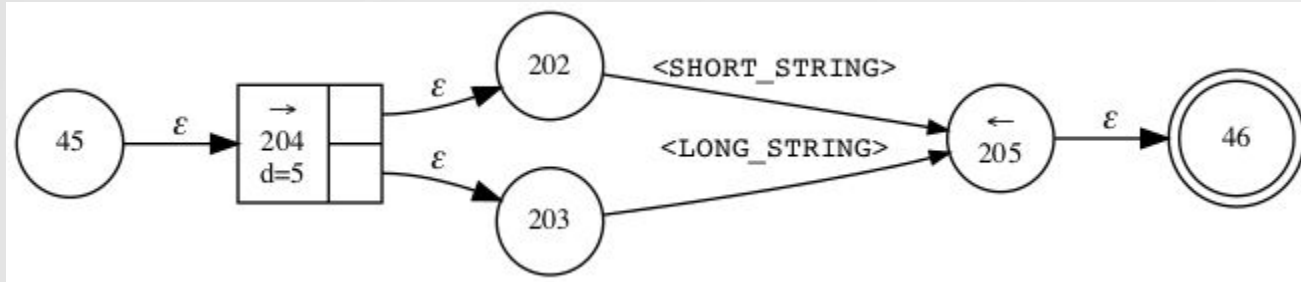
So, the boolean data type is defined as a variance from either TRUE or FALSE patterns, with optional functor, declared between ":(\" and \")\".

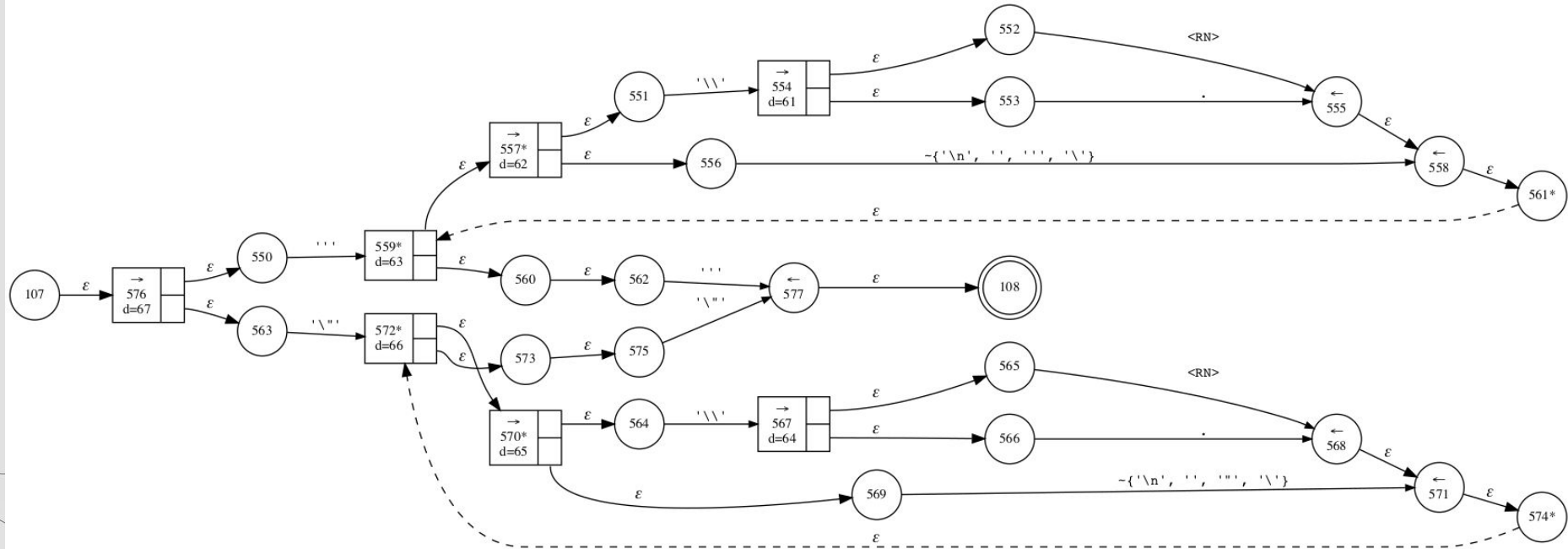


So, the boolean data type is defined as a variance from either TRUE or FALSE patterns, with optional functor, declared between ":(\" and \")\".

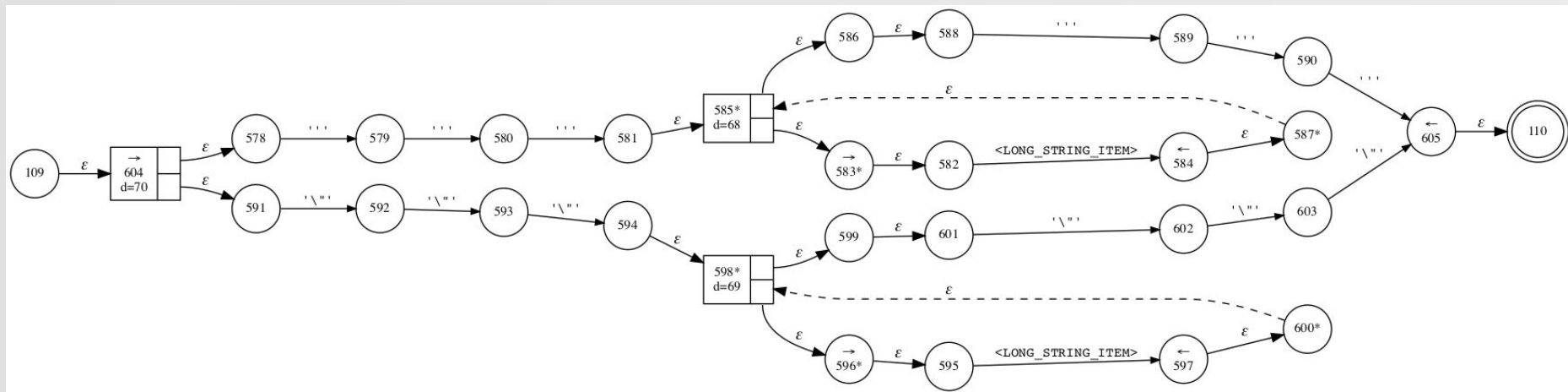


There are two types of the string. There are short string, means string which does not have a newline in it and long strings, the strings which can have a new lines. Both variants essentially represent same data type, that is a unicode string.





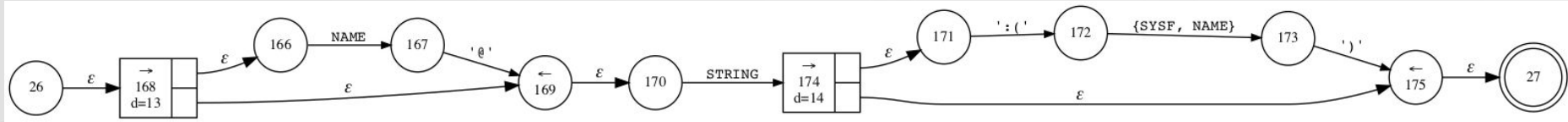
And the long strings is declared as a group of characters, surrounded by either triple \" or triple \' quote.



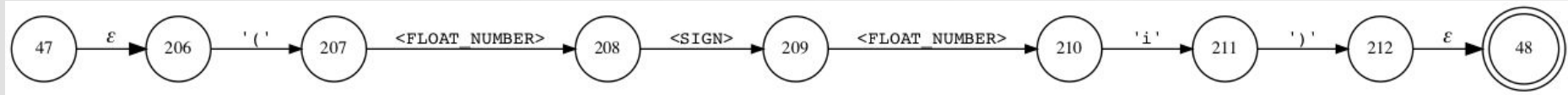
So, the string could be either represented by short string or a long string. The difference is in quotes (single or triple) and in ability to use newlines in the string. Like any other data type, you can add optional functor as suffix, enclosed between ":" and "(" and ")". Unlike other data types, string does have a feature called "prefunction". This is an optional prefix, in form of NAME, suffixed by "@". Example: json@"{}".

Prefunction is a mechanism to convert string to some other data type on the spot. It is working like this:

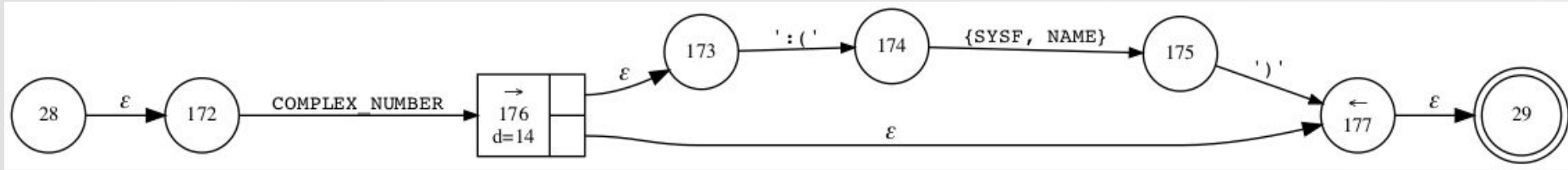
1. If prefunction name matches some data type, BUND will try to convert content of the string to the object of that type. The object will be placed on top of the stack. In above example, BUND will convert content of the string "{}" to a JSON object.
2. If prediction name matches some embedded function, string value converted to string object and passed as parameter to that function. The object returned by the function will be passed to the BUND for further evaluation.



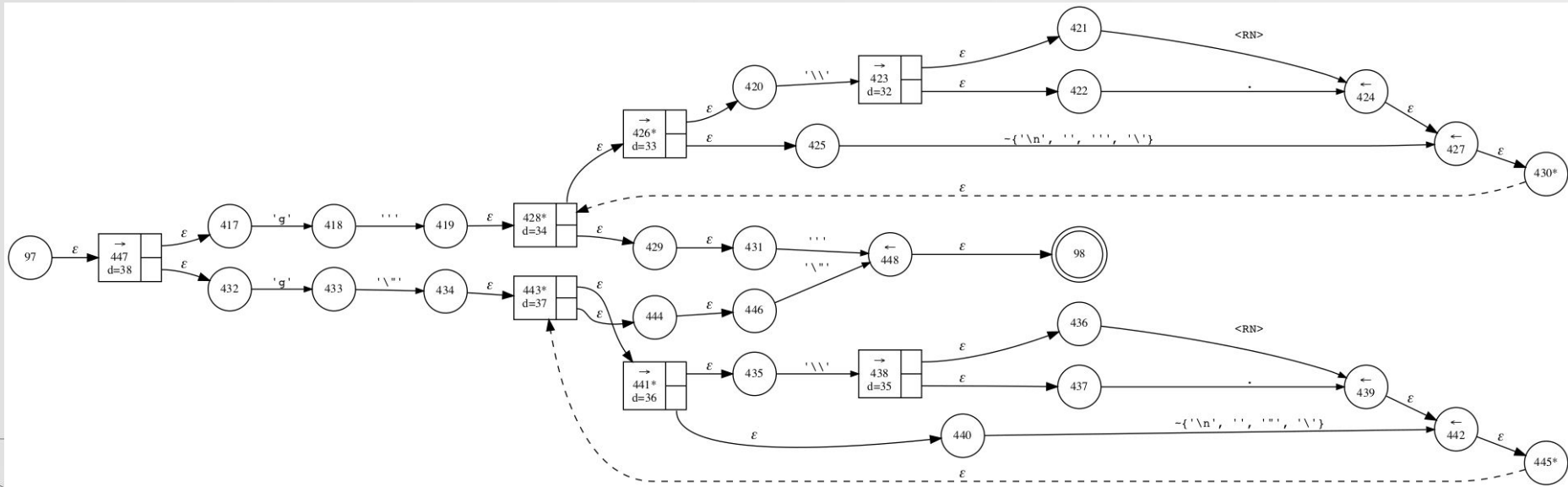
BUND language supports complex number declaration as "(" FLOAT representing a real part of the complex number, SIGN ("+" or "-") another FLOAT representing imaginary unit and "i" Example: (3.14+3.14i)



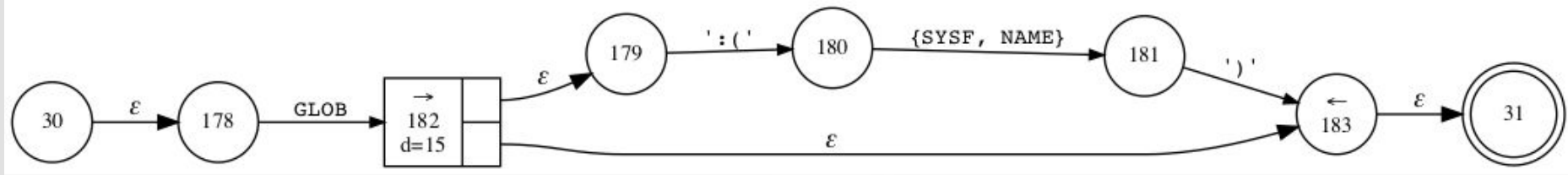
As with any other data types, you can suffix complex number with functor specified between ":" and ")"



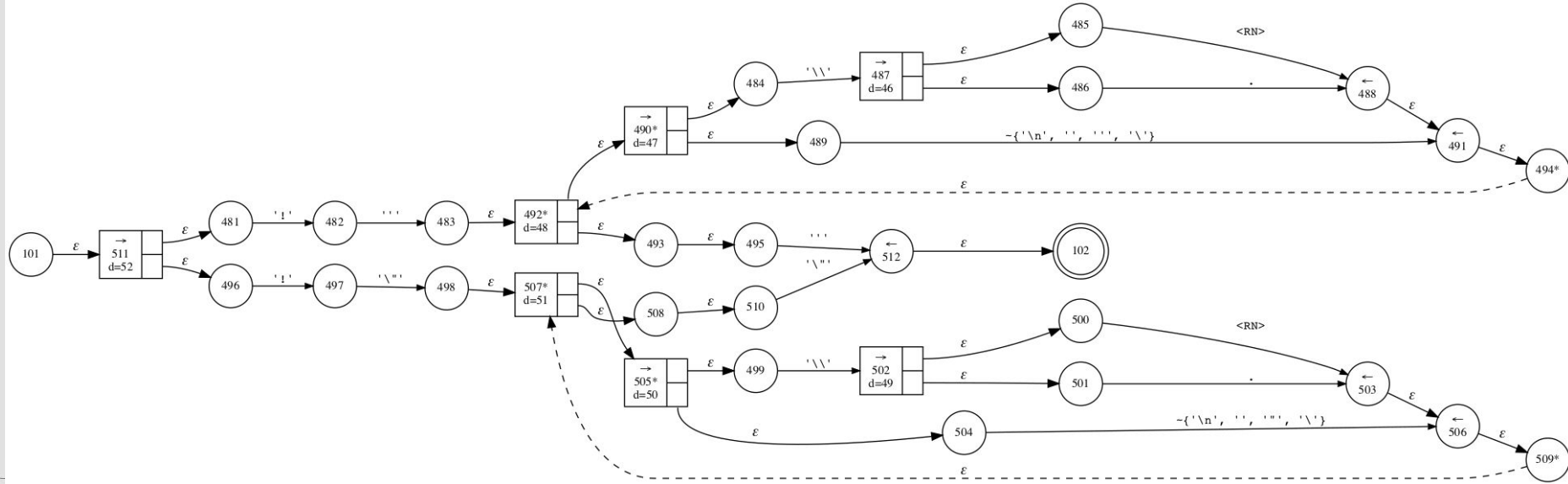
BUND supports GLOB text pattern matching and text patterns are the separate data type. You can define GLOB pattern just like a short string, prefixed with letter "g". Example: g "*.txt"



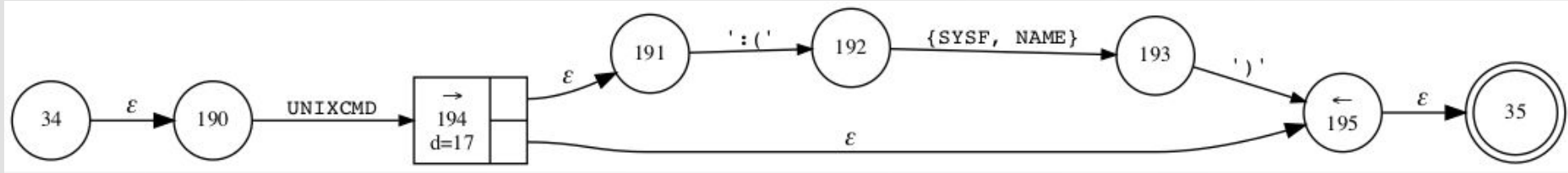
As with other data types, you can suffix your GLOB pattern with functor, specified between ":" and ")"



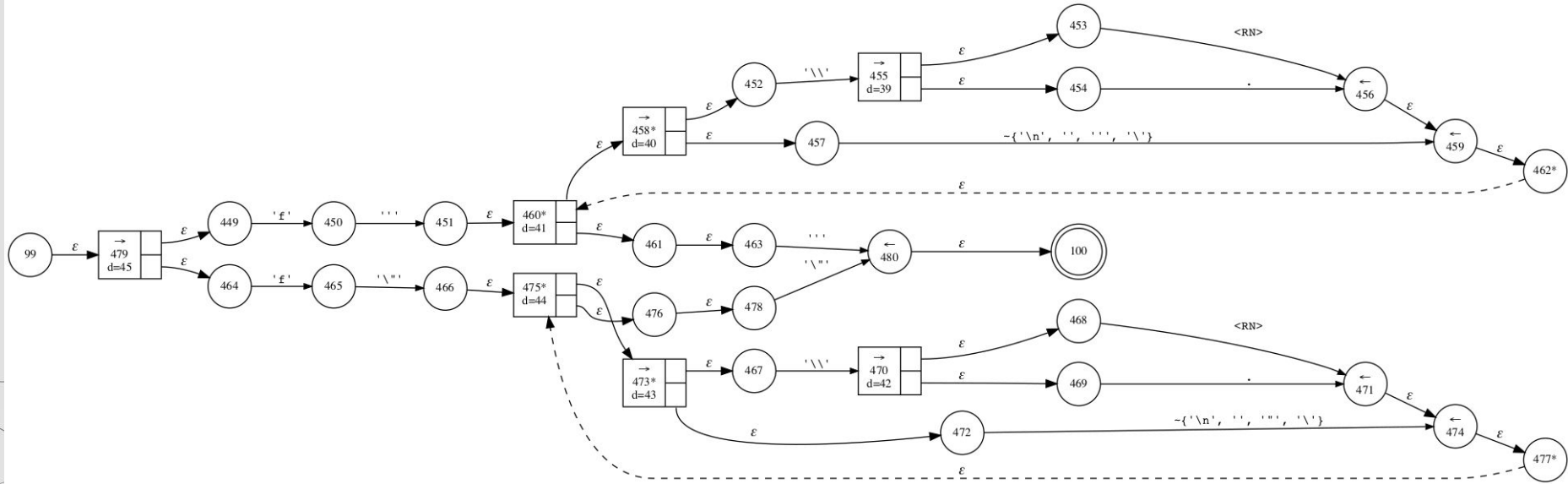
BUND supports "Unix command" term as a distinct pattern matching. If you are store unix command in the string, prefixed with "!", example "!df -h", BUND will automatically execute this command and return standard output of command as a string data.



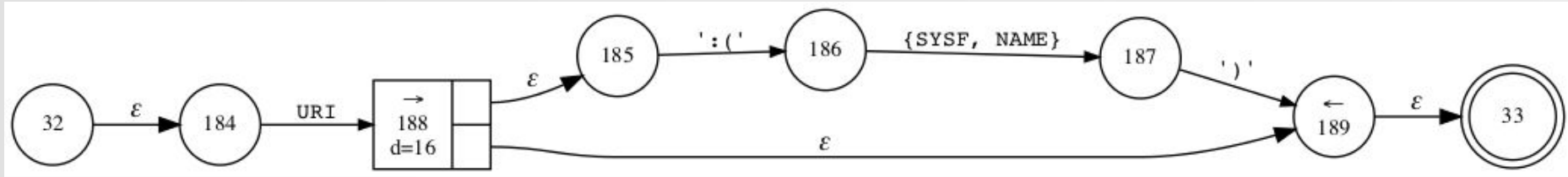
If you want to post-process the string data returned after execution of the UNIX command, you can can functor in the suffix of the command. Example: `!cat /etc/protocols):(str/Lines)` This command returns content of the file, stored as after string is splitted in lines of the text. Result is placed on top of the stack.

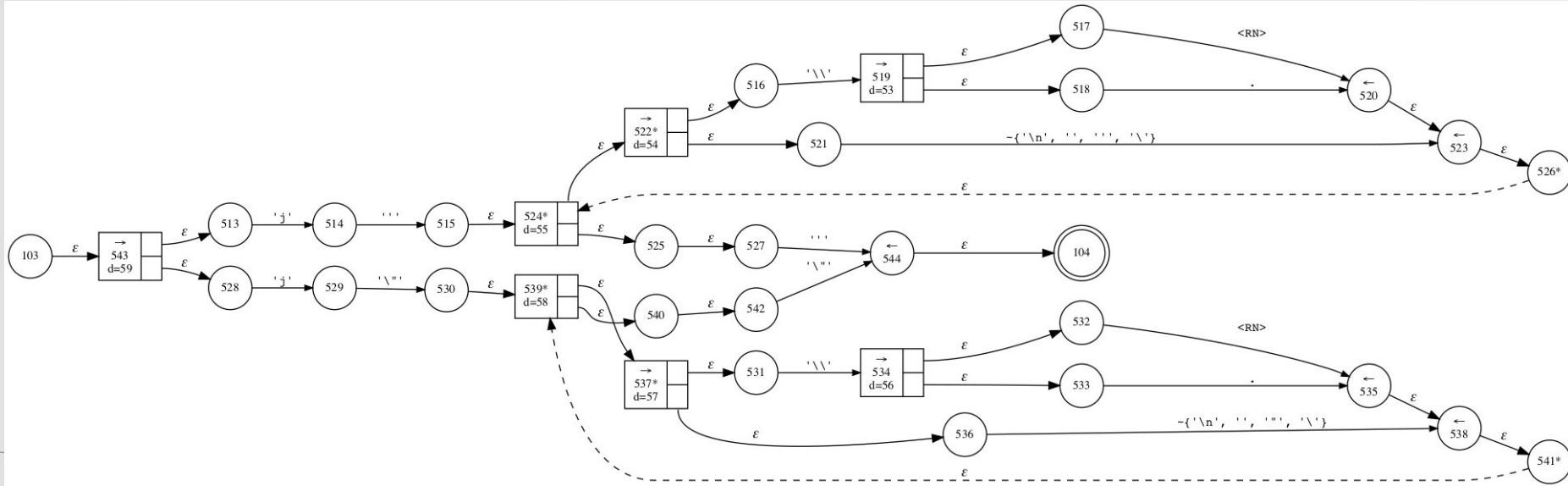


In order to work with files, you have to create a file object. One of the methods of creating a file object is prefix file URI defined in string with prefix "f". Example: `f"file:///tmp/temporary.dat"`. When prefixed string is passed, bund will place on stack a file object instead of string. After you have file object in stack, you can manipulate with content of this file using `fs/*` functions.

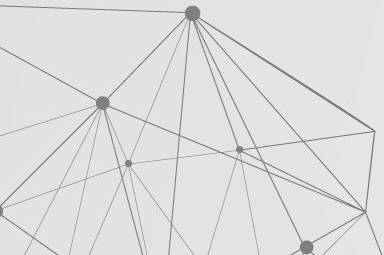
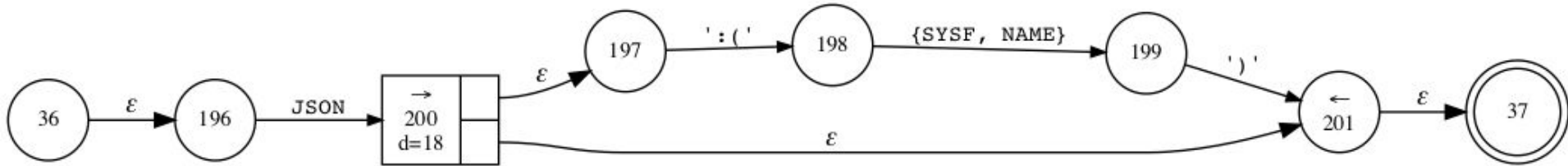


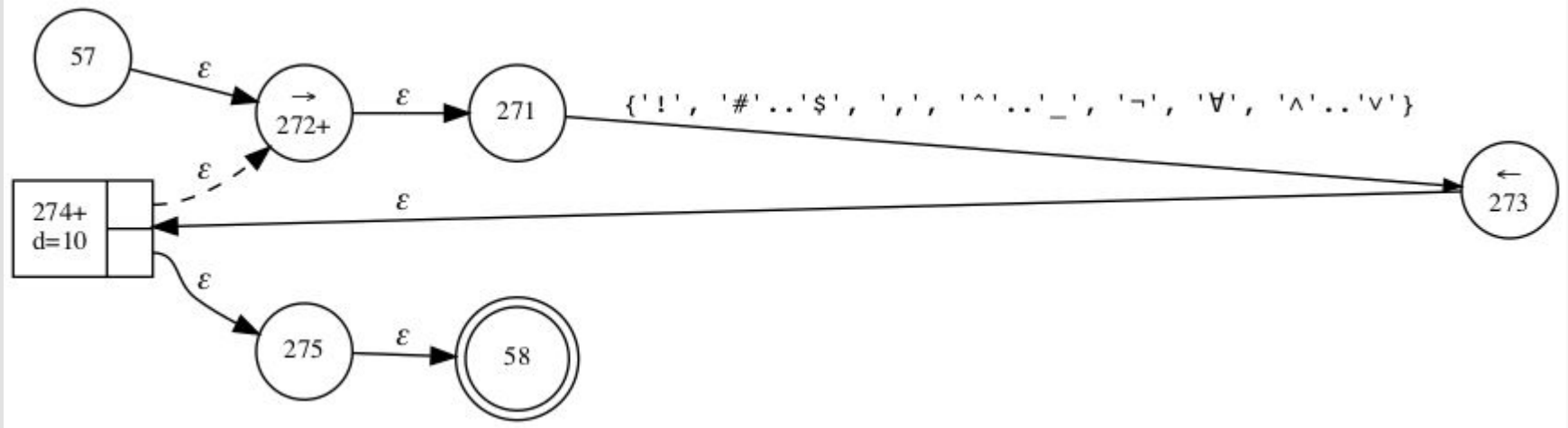
You can suffix your file object with functor name, specified between “: (“ and “)”.



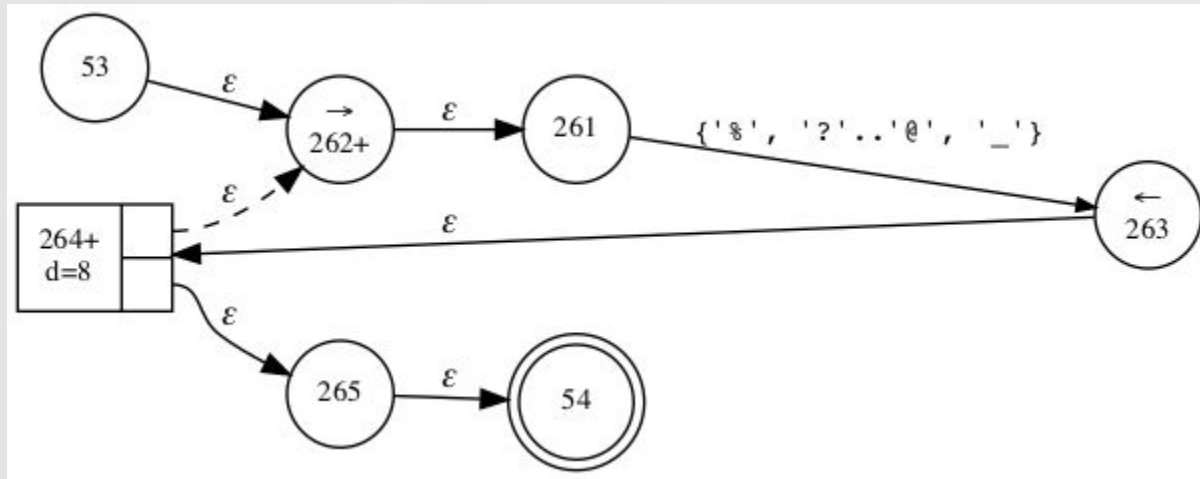


BUND do have a direct support for JSON generation and parsing. You can specify an object of JSON type by adding suffix "j" to a string object. Example: j{"answer": 42}'. BUND will recognize this not as string but as JSON and you will be able to use various functions for query and altering of JSON data.

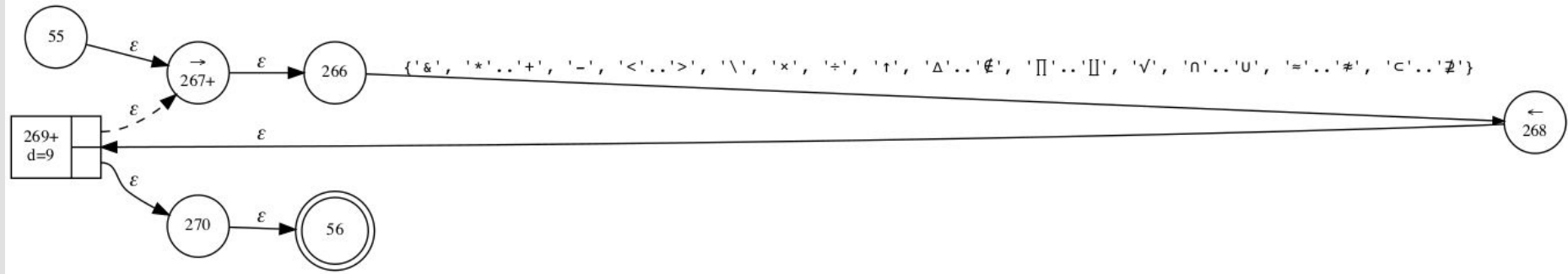




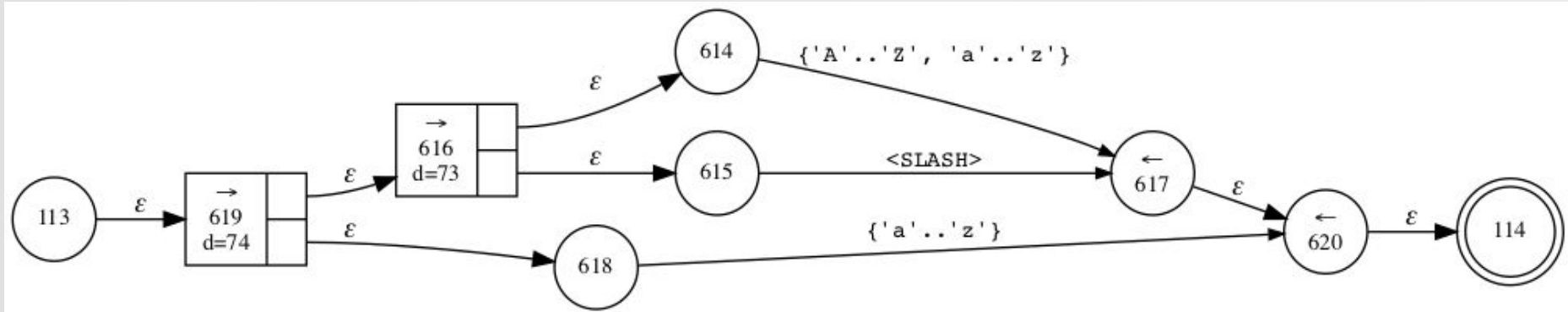
Next pattern is a SYS pattern and it defines a set of characters that I am going to use in system commands.



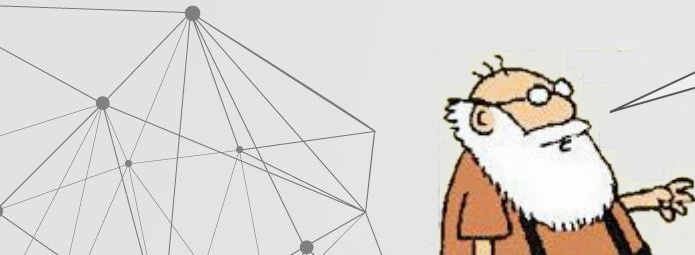
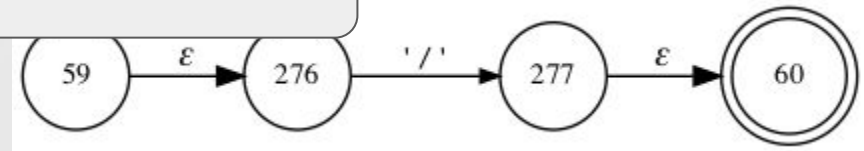
And third, set of characters I am going to use in OPERATOR names. So, if you see any of this in BUND program, you are definitely looking at OPERATOR.



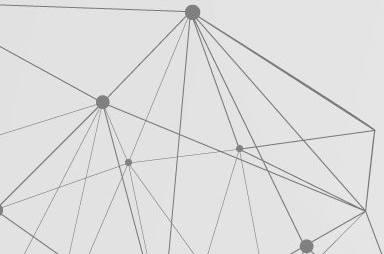
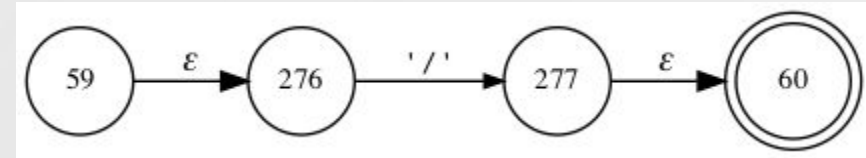
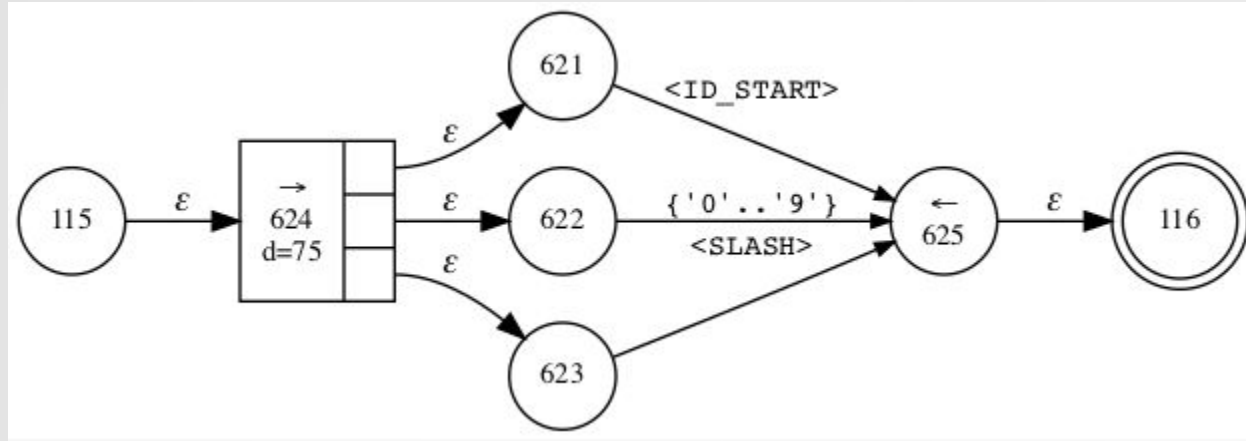
And finally, we do need to declare some names. Those are not variable names though. But every time when you do need to name something, you are going to use NAME. NAME starts with ID_START, which can be either uppercase or lowercase letter, or forward slash "/".



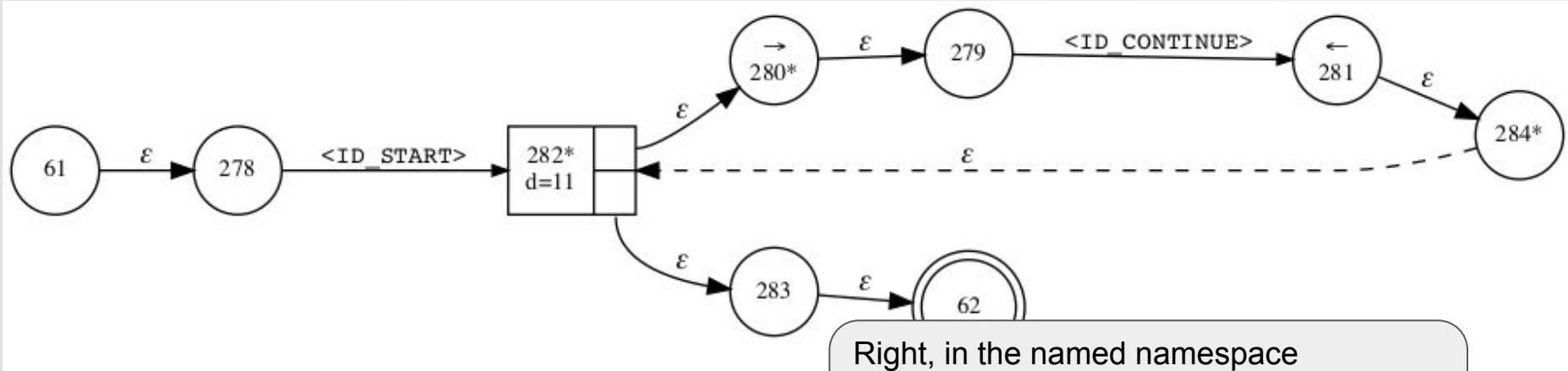
And where we already see the use of the names ?



And ID will continue with either upper or lower case letters, or numbers from 0 to 9 or with forward slash.



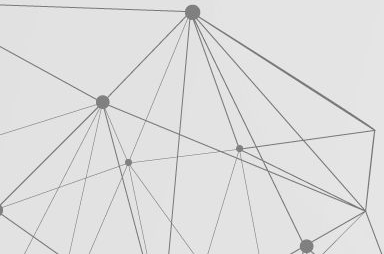
Again, NAME will start as ID_START and continue as ID_CONTINUE. While number can be a part of the NAME, NAME never starts with number. NAME can contain any number of forward slashes.



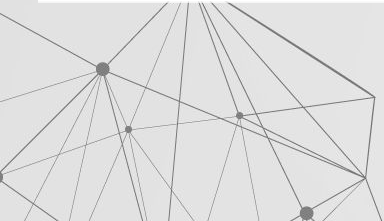
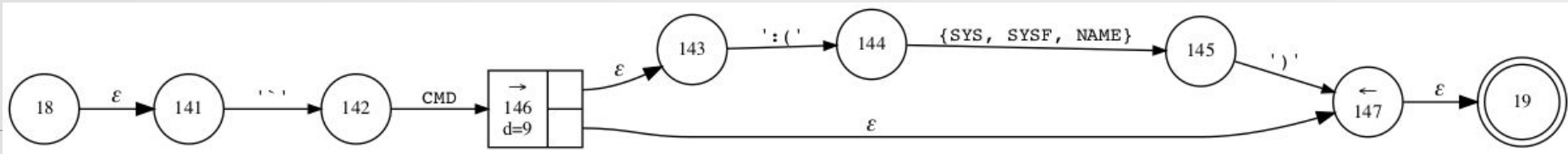
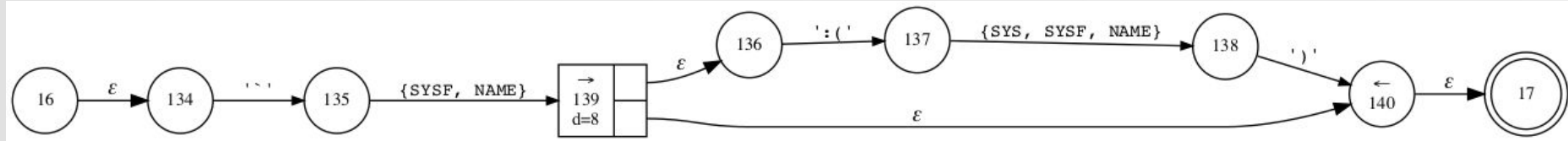
Right, in the named namespace declaration, where we gave the names to our namespaces. Also in references to a functor names



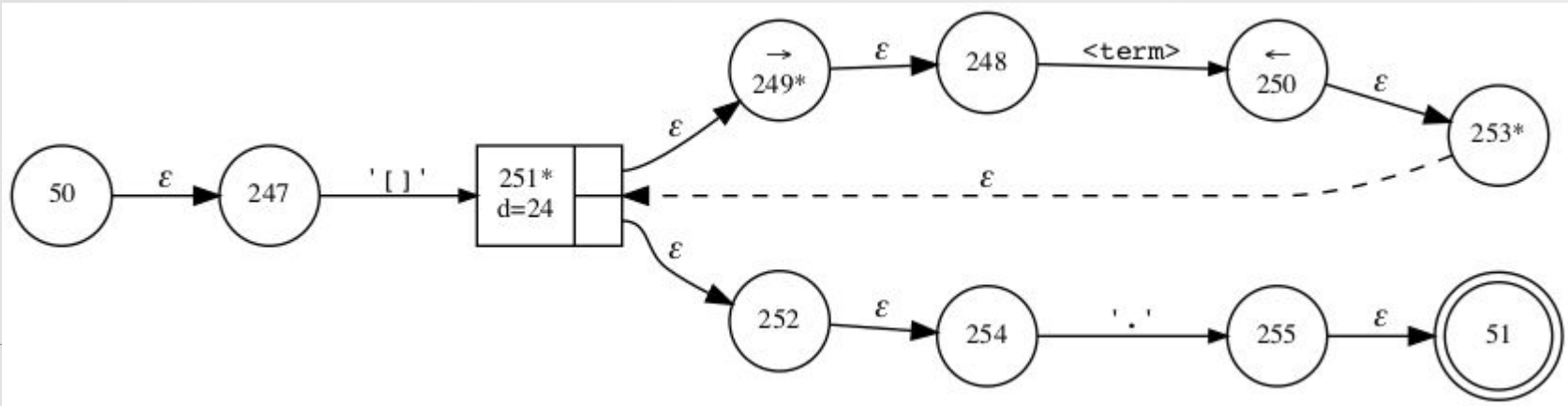
This chapter is all about data types and what is considered to be a data in BUND. But due to a dynamic nature of the language, in chapter 3 we will overstep to a chapter 4. And if something that introduced here about functions and lambdas which can be used as data i not clear, I will talk more about it in the next chapter.



References to the functions or operators. The reference on the function or operator is not the call for the evaluation of function or operator, it is rather placing a special data in the stack, referring BUND function or operator. Speaking in GO or C terms, it is like to store a pointer to a function, so later on you can call and execute this function. We will talk more about how we can execute by the reference in the next chapter, but all you need to know at this point is that you can create reference on function or operator by prefixing function or operator name with backtick - "`". Please note, at the time of creating this reference, but do not checking if the object actually exists. This is an example of late binding in the BUND.

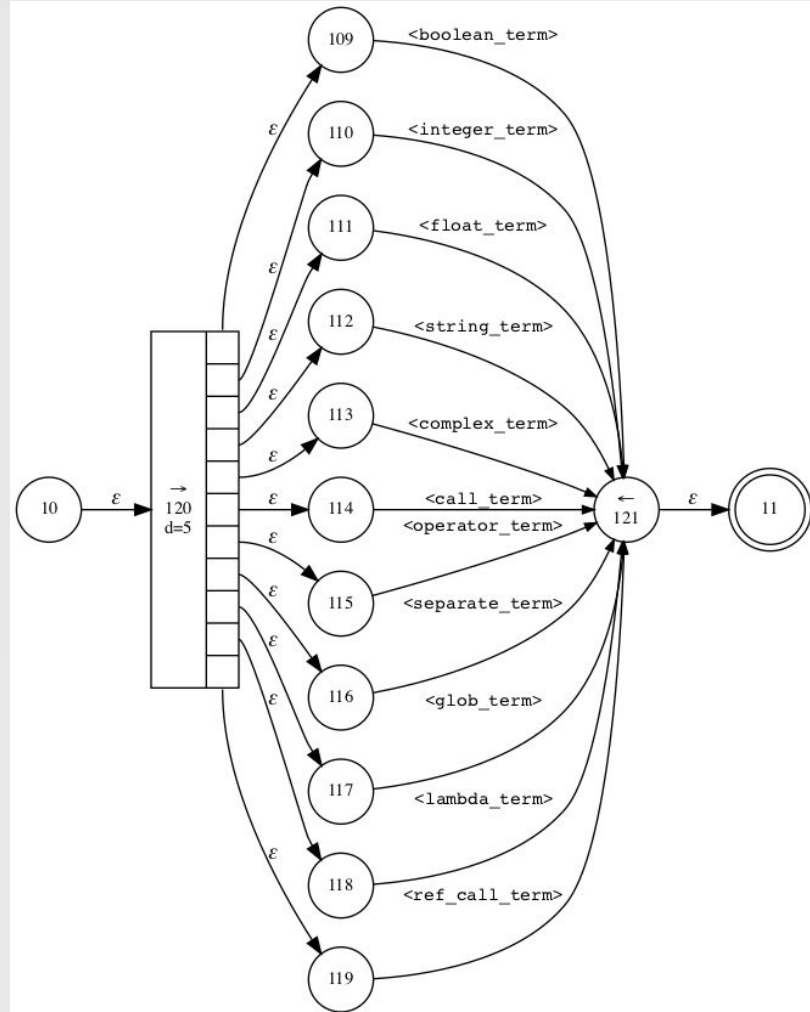
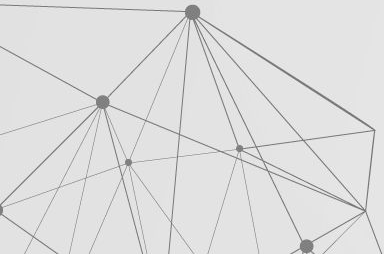


I will talk about anonymous functions or lambdas in the chapter 4, but I will introduce them now, as they are used in data blocks as a data. Anonymous function is a function declared with header “[]” and terminated by “.” You can use all terms supported by BUND in between just as you use them in namespaces. When BUND evaluates “end-of-lambda”, or “.” it will place the reference on that anonymous function on the top of the stack.

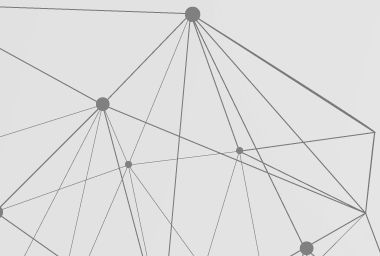
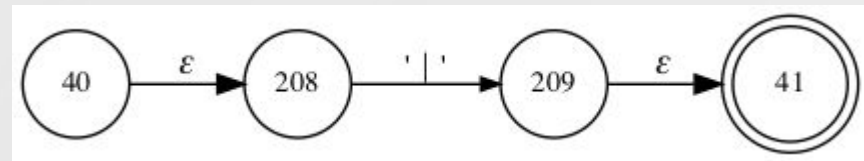
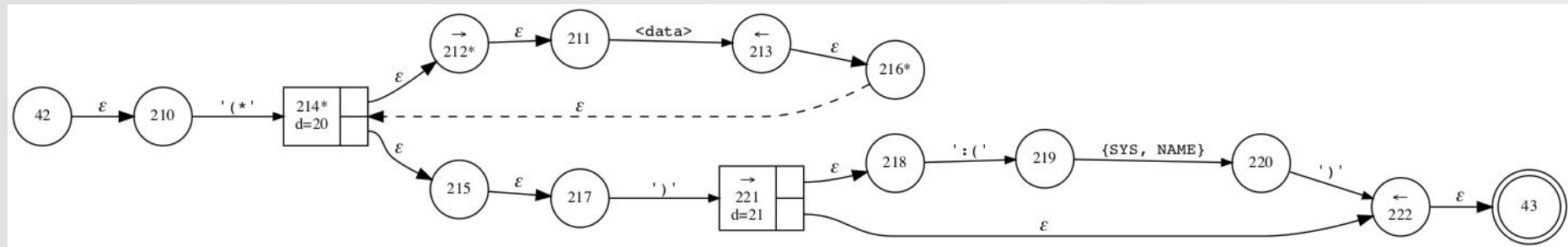


All this maybe premature introduction of the references and lambdas are necessary reading before I introduce the “data” types. Data types are distinctly different from general terms that you can use in namespaces, that they are limited to an entities, that actually can be considered as the data, not controls. And only “data” types are allowed in “Data Block” otherwise known as “DBLOCK” type.

Data block is a sequence or vector of data elements that you can separate in logical blocks using separator.

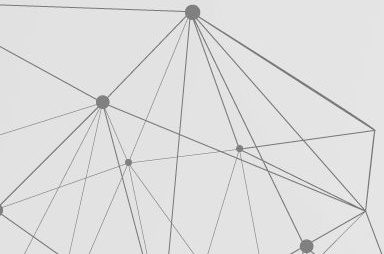


Data elements are placed between “(“ and “)” and normally are space separated. If you want to emphasise a logic segments inside data block, you can use vertical “pipe” - “|” as a separator. There are cases, where BUND is using that logical separation. First, when is it converting data block to a matrix. Separator in this case breaks data into a rows. Second, when you are preparing a training data for a PERCEPTRON data type, you can use separator to separate pattern data outcomes and rows of training data.

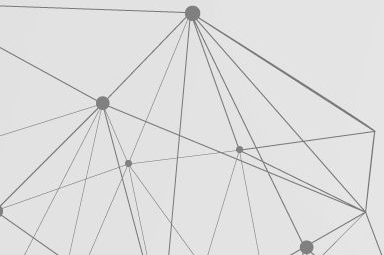


To summarize this long and elaborated description of supported data types, let's put all what we discussed into a table.

Code	Name	Description	Example
int	Integer	GoLang math/big integers.	420000000000000000000000
flt	Floating point	64-bit standard floating point numbers	3.14
bool	Boolean.	Boolean values as True or False	True or False
str	Unicode-8 string	Unicode string	'Hello world', "Привет Мир", """"This is an example of multiline string""""
cpx	Complex number	128-bit complex number	(3.14+3.14i)
dblock	Data vector	Single dimension array of mixed data with optional logical separation.	(* 'This is string' True 3.14 42)



Code	Name	Description	Example
file	File	URI pointing to the file on local filesystem or on S3 or Google block.	f'file:///tmp/file.name'
glob	GLOB pattern	GLOB-style pattern	g"[H h]ello"
json	JSON	JSON object that you can alter or query	j'{"answer":42}'
CALL	Reference to the function	Reference for the function implements late binding	`println
OP	Reference to the operation	Reference for the operation implements late binding	`+
unixcmd	Unix command	Unix command which will be executed in-line	!`df -h`



04

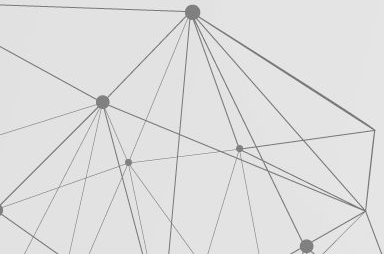
FUNCTIONS, OPERATORS, LAMBIDAS

Here is how you change your data

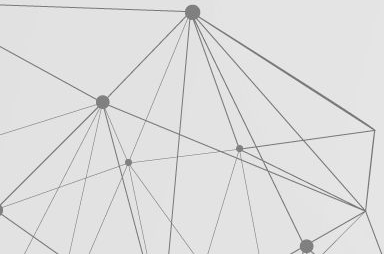


We are successfully defined which types of data you can store in the Stack. Now, let's discuss how you can modify that data. There are three options on how you can manipulate with data.

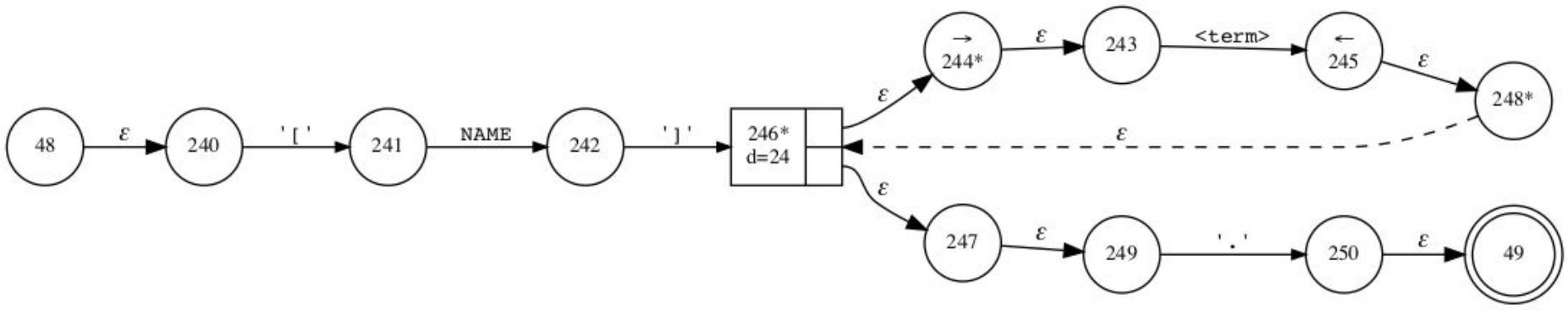
- ❑ Named functions
- ❑ Anonymous functions or Lambdas
- ❑ Operations



Named functions are always taking one value from the stack and place value to the stack. Essentially, the size of the Stack after you applied a function remains the same. There are handful of exceptions, where ether function doesn't require anything to be present in stack or doesn't return anything. But those exception are very few and served very specific system-level tasks. Named function always do have a name, and this name are ether global, if this is an embedded function, i.e. function which is not user-defined and are the integral part of the BUND interpreter. Embedded functions resolved globally, i.e. they are available from any namespace. Generally, those functions defined at the core of interpreter and implemented in Go. To add new embedded function, you have to ether modify BUND sources or embed BUND in your own application. We will not discuss how to do it in this document. User-defined function are implemented in BUND language and local to the namespace. If you are planning to use function defined in another namespace, you have to explicitly use "->" operator to make function available in a local namespace.

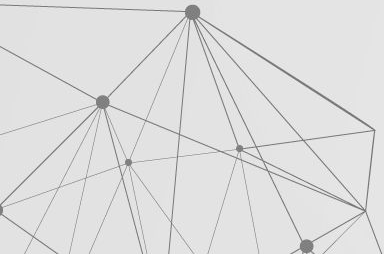


Named function defined as follows, first, a name enclosed in "[" and "]" then the list of terms that will be executed in current namespace when function is executed, then function is terminated with "." Declaration of named function doesn't changed stack.

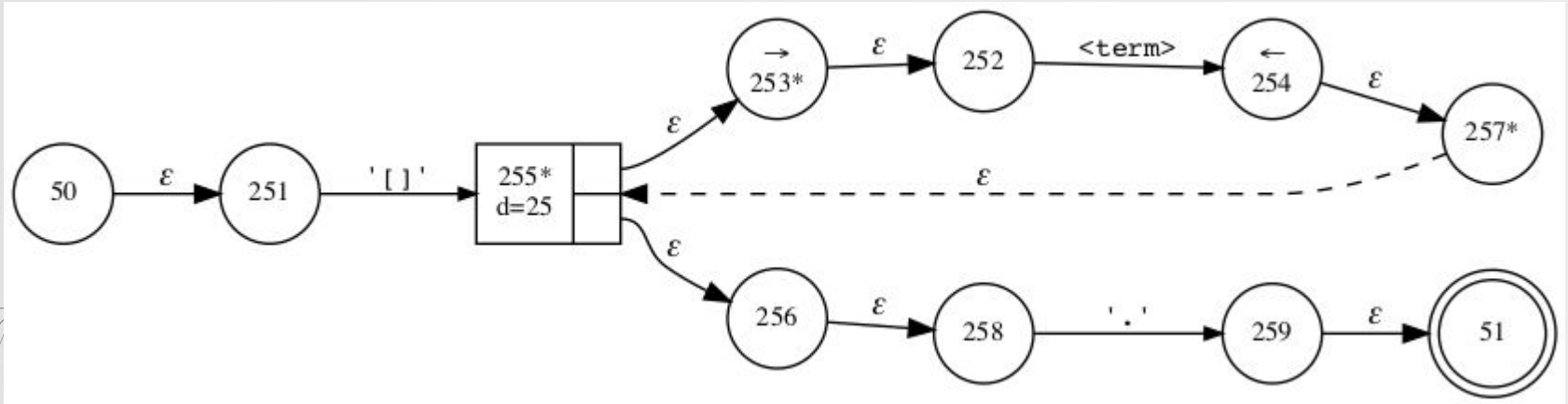


In order to use user-defined function, that been defined in different namespace, you can use a “use” operation, which is looks like “->” or unicode symbol “→”. On top of the stack, you must have a namespace name and next is a function name. If the function with this name is declared in that namespace, it will be propagated with the same name to a current namespace. This operator leaves reference to that function in the current stack.

```
1 (  
2   [ a :  
3     [ans] 42 .  /* Defining function "ans" in namespace "a"  
4   ;;  
5   'ans' 'a' ->  /* Importing function 'ans' from namespace 'a' to current namespace */  
6   ans println  /* Calling the function */  
7 )
```



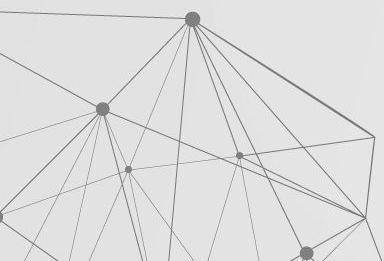
Anonymous function declared similarly to the named function. And as with named function, you can expect that one item will be taken from the stack and one item will be returned. Since name for the anonymous function is automatically generated, you normally do not have an access to that function by name. When you terminate your function declaration with final dot - "." the reference to the function will be placed to the stack. You can store it for a later execution, or immediately execute this lambda with operator "!"



Operation "exec" or "!" will expect a reference to the function or operator on top of the stack. Then it is taking this reference from the stack and execute it in current namespace. Returned value will be placed on the stack.

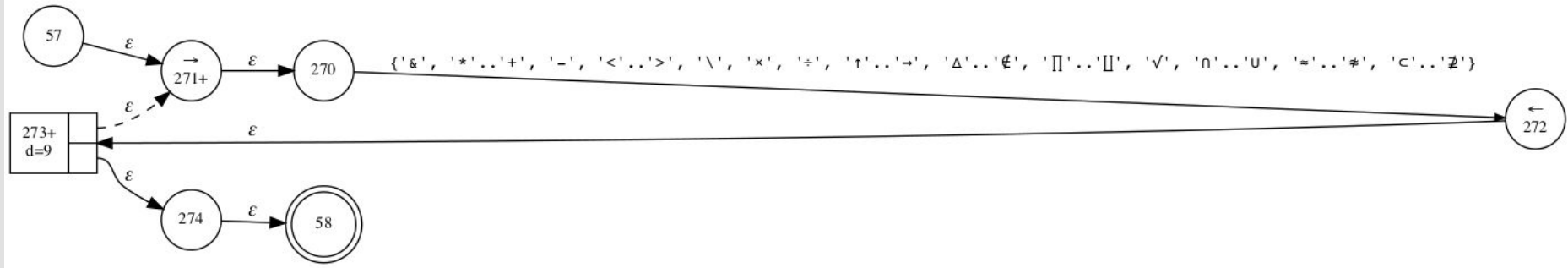
You can create reference by declaring lambda function, or prefix function name or operator name with backtick "`". Reference is acting like any other data item. You can print it, duplicate it, zip it to data block and so on.

In this example, we declaring a function in current namespace, called FortyTwo. And this function is doing something like taking element from the stack and multiply it on 21. Then we are placing number 2 on the stack, then reference to the function FortyTwo and then calling for execution by that reference.



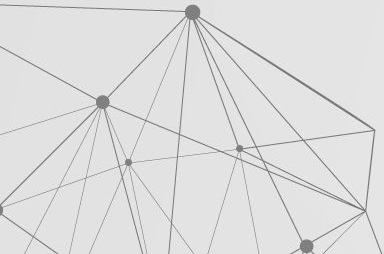
```
1 (
2     [FortyTwo] 21 * .
3     2 `FortyTwo ! println
4 )
```

Operators always taking two values from the stack performed computation over those two values and return one item back to stack. If you see symbols from that list, you are definitely looking at operator. NAME, could be ether function or operator.



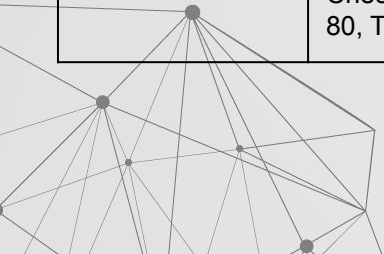
Let's take a look at list of embedded functions and operators defined in BUND. Start with basic arithmetic operators.

Operator	Description	Example
+	Arithmetic add. BUND is trying to be smart about how to perform "add" operator to the data of different types.	2 2 +
-	Arithmetic subtract. In the example given we are performing (43 - 1)	1 43 -
× or *	Arithmetic multiply. In the example given, we multiply a number with elements of the data block.	2 (* 2 3) *
÷ or \	Arithmetic divide. In the example given we divide 6 on 2.	2 6 \
**	Pow operator. In the example given we calculate 2 in a power of 64	64 2 **



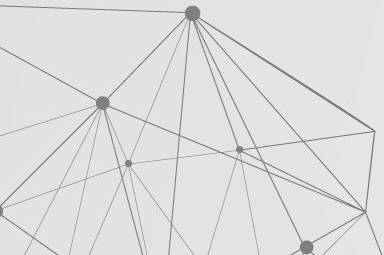
Then continue to a common comparator operators. Bund will make a best guess on how to compare a different data types.

Operator	Description	Example
>	Check if more. In example given, we are checking if 2 more than 3, True or False value will be placed on the stack.	2 3 >
<	Check if less. In example given, we are checking if 1 less than 43, True or False value will be placed on the stack.	1 43 <
=	Check for equality. If two objects on the stack are equal, value True will be placed on stack. False otherwise.	(* 2 2) (* 2 2) =
<>	Check for non-equality. If two objects on the stack are not equal, value True will be placed on stack. False otherwise.	3.14 3.15 <>
<=	Check if less or equal. In example given, we are checking if 4 less or equal than 4, True or False value will be placed on the stack.	4 4 <=
>=	Check if more or equal. In example given, we are checking if 3 more or than 80, True or False value will be placed on the stack.	3 80 >=



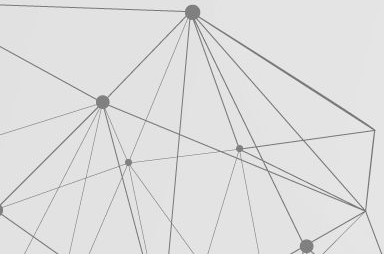
GLOB pattern matching operators

Operator	Description	Example
===	Pattern match string against GLOB pattern. Returns True or false on the Stack	'Hello world' g'[h H]* world' ===
<===	Pattern match string against GLOB pattern to search for suffix. Returns True or false on the Stack	'342233' g'33' <===
===>	Pattern match string against GLOB pattern to search for prefix. Returns True or false on the Stack	'Xello world' g'world' ===>
==<==	Pattern match string against GLOB pattern to search for longest prefix. Returns True or false on the Stack	'Hello world' g'Wo' ==<==
=>=	Pattern match string against GLOB pattern to search for longest suffix. Returns True or false on the Stack	'Hello world' g'ld' ==>=



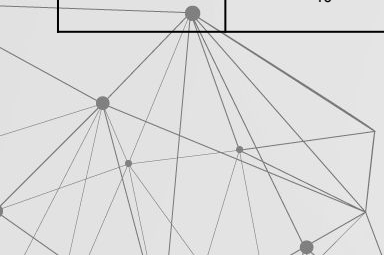
ZIP operator. This operator aggregate two elements on the stack in data block. If one of the elements is data block, it will be altered, otherwise, new data block will be created.

Operator	Description	Example
++	Aggregating elements in stack into a data block. The outcome of the example given will be: (* Result must be false and it is %v ,False ,). As it happens, this is an example of the data that digested by console/* functions.	False (*) ++ 'Result must be false and it is %v' ++



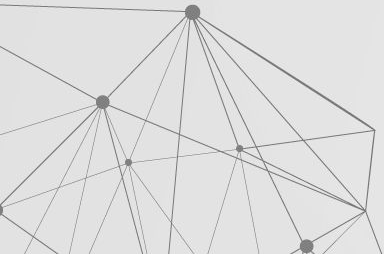
Next, let's review some mathematical functions and operators. Bund is making the best to recognize and properly handle different datatypes.

Operator	Description	F/O	Example
math/Sin	$y = \sin(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	0 math/Sin
math/Cos	$y = \cos(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	3.14 math/Cos
math/Tan	$y = \tan(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	(* 1.0 2.0) math/Tan
math/Sqrt	$y = \sqrt{x}$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	2.0 math/Sqrt
math/Exp	$y = \exp(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	1000000 math/Exp
math/Log	$y = \log(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	10 math/Log
math/Log10	$y = \log_{10}(x)$ function. Float numbers, Integers, data blocks and matrixes are supported.	F	2 math/Log10



One of the common calculations is a percent calculations. In the BUND we've got you covered.

Operator	Description	F/O	Example
percent	Calculates number of percent fraction of the number. 25% of 200 is 50	O	25 200 percent
percentOf	Calculates percent fraction of number. 300.0 is a 12.5% of 2400.0	O	2400.0 300.0 percentOf
changeOf	Calculates percent of change between two numbers. From 50 to 100 is 100% of change.	O	50.0 100.0 changeOf

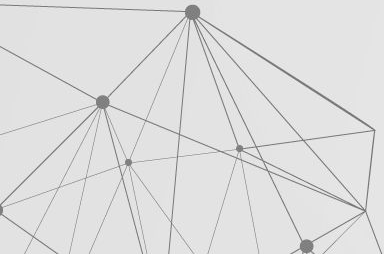


In the next step, let's review the Matrix operations. Matrix data type, with signature "MAT" is a library-defined data type. There is no BUND language syntax features that you can use, to define Matrix. Matrix is a two-dimensional data structure, for effective storage and operations over floating-point numbers. Here is the functions and operators for creating and manipulating with Matrix data.

Operator	Description	F/O	Example
M	Creates initial matrix. Expects the data block with matrix definition such as dimensions and initial value. In example given, BUND will return to the stack a Matrix of 4 rows, 3 columns, initialized with value 3.14	F	(* 3.14 3 4) M
M/Make	Creates Matrix based on data stored in data block, separated in rows with Separator " ". In example given, we are creating 2 row, 3 column matrix, filled from the end of the data block. [6 5 4] [3 2 1]	F	(* 1 2 3 4 5 6) M/Make
M/ToBlock	This function will convert Matrix to data block and place separators in proper positions of the block. In example given, M/ToBlock will reconstruct back initial datablock used to create this matrix. (* ,6 ,5 ,4 , ,3 ,2 ,1 ,)	F	(* 1 2 3 4 5 6) M/Make M/ToBlock
M/Set	This operator sets value in the matrix. On top of the stack it expects to have a Set data block, and before that is Matrix. Set data block is dblock, where the first two elements is a row and column and last is a value that you are setting to that row and column. In example given, we are storing 2.0 to the coordinates 1:1 [3.14 3.14 3.14] [3.14 2.00 3.14 [3.14 3.14 3.14] [3.14 3.14 3.14]	O	(* 3.14 3 4) M (* 2.0 1 1) M/Set
M/Get	This operator returns the value stored in Matrix. It does expect to have a Matrix and Get data block, where first element is a row and second is a column.	O	(* 3.14 3 4) M (* 2.0 1 1) M/Set (* 1 1) M/Get

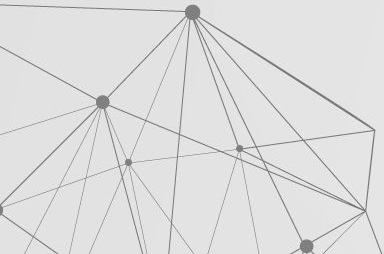
Have a decent source of pseudo-random data is an important feature of many applications. BUND language have a three functions designed to give you a random number values

Operator	Description	F/O	Example
rnd/Integer	Takes two numbers from the stack, indicating upper and lower range of the random numbers you are looking for. In example given, 10 is a lower range and 1000000000000000 is upper. Return randomly generated number to stack.	O	1000000000000000 10 rnd/Integer
rnd/Float	Takes two numbers from the stack, indicating upper and lower range of the random numbers you are looking for. In example given, 10.0 is a lower range and 100.0 is upper. Return randomly generated number to stack.	O	100.0 10.0 rnd/Float
rnd/Prime	Calculate Prime number of the specified number of bits. In example given, it will generate a prime number for 256 bit.	F	256 rnd/Prime



Here is the current list of the functions and operators that designed to work with string data types

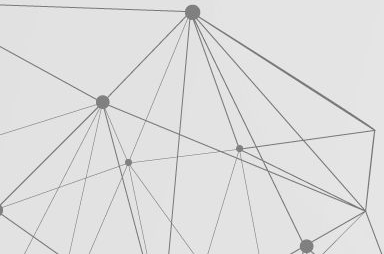
Operator	Description	F/O	Example
str/Strip	Remove prefixed and tailed spaces and newlines. Altered string is stored in the top of the stack.	F	" Hello world ! " str/Strip
str/Lines	Take a string from the stack and return datablock with string splitted in lines	F	"One\nTwo\nThree" str/Lines



There is one data type, called "file" and number of functions that are dedicated to work with files. In the future, BUND will support file on local filesystem as well as files on S3 and GCP platforms, but currently, only local filesystem is supported. To open or create a file, you have to open string as URI in one of the few different ways:

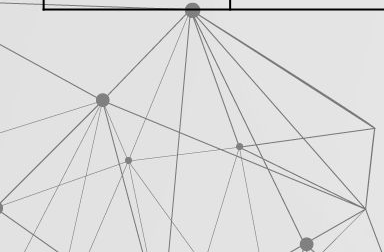
1. Open as file data type: `f"<URI>"`. Example: `f"file:///tmp/file.txt"`. This directive will be taken from the stack and file object pushed to the stack.
2. Use prefunction. Example: `file@"file:///tmp/file.txt"`.
3. Use functor. Example: `"file:///tmp/file.txt":(file)`
4. Use function. Example: `"file:///tmp/file.txt" fs/File`

No matter, which option for open/create a file that you are choose, the file object will be placed at top of the stack.



Functions/Operators for work with file object.

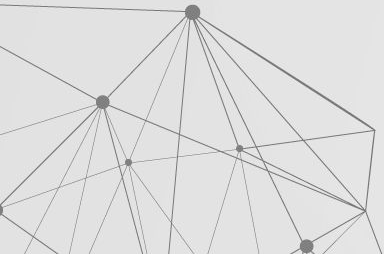
Operator	Description	F/O	Example
fs/File	Takes a string data item from the stack and push file object to the top of the stack.	F	"file:///tmp/file.txt" fs/File
fs/File/Exists	Takes the file data object from the stack check, if file described by this object exists, then pushes file data element back to stack and push True or False to the stack.	F	f"file:///tmp/file.txt" fs/File/Exists
fs/File/Write	Takes file object, then string object and writes that string to the file defined by that file object.	O	'Hello World' ffile:///tmp/file' fs/File/Write
fs/File/Close	Closes the file defined by file object on top of the stack.	F	'Hello World' 'file:///tmp/file':(file) fs/File/Write fs/File/Close
fs/File/Read	Reads content of the file defined by the file object on top of the stack, stores reference to the file on the stack and return file content as string. In example given, first, we place a string into stack, than open file and store reference to the file on the stack. Next, we write that string and put reference to the file on the stack. Then we close file and put the reference to the stack then we read file content, put the reference back and string to the top of the stack	F	'Hello World' 'file:///tmp/file':(file) fs/File/Write fs/File/Close fs/File/Read



BUND language natively supporting operations with JSON data. You can generate or query JSON objects at will. There are four ways to define JSON object:

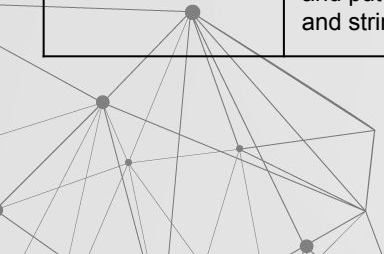
1. Open as JSON data type: `j"<JSON data>"`. Example: `j'{"answer": 42}'`. This directive will be taken from the stack and JSON will be object pushed back to the stack.
2. Use prefunction. Example: `json@"{}"`.
3. Use functor. Example: `'{"Pi": 3.14}':(json)`
4. Use function. Example: `'{"data": []}' json/New`

No matter, which option you are to choose, the JSON object will be placed at top of the stack.



Functions/Operators for work with JSON object.

Operator	Description	F/O	Example
json/New	Takes a string data item containing JSON formatted data from the stack and create and push JSON object to the top of the stack.	F	'{}' json/New
json/Value/Set	Takes JSON object and data block from the stack. In the data block, BUND expects to find pairs key(string) -> value(JSON-supported value) and will set those values to this keys	O	(* 'answer' 42 'not_answer' 41) j'{}' json/Value/Set
json/Value/Get	Takes JSON object, then string object representing key and returns value corresponded with this key	O	'answer' * 'answer' 42 'not_answer' 41) j'{}' json/Value/Set json/Value/Get
json/Array/New	Takes JSON object and data block and creates arrays assigned to the keys listed as strings in data block. In example given, resulting JSON will be as {"a":[],"b":[]}	O	(* 'a' 'b') j'{}' json/Array/New
fs/File/Read	Reads content of the file defined by the file object on top of the stack, stores reference to the file on the stack and return file content as string. In example given, first, we place a string into stack, than open file and store reference to the file on the stack. Next, we write that string and put reference to the file on the stack. Then we close file and put the reference to the stack then we read file content, put the reference back and string to the top of the stack	F	'Hello World' 'file:///tmp/file':(file) fs/File/Write fs/File/Close fs/File/Read

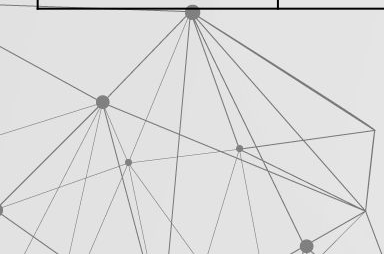


Printing and console operations.

Operator	Description	F/O	Example
print	Takes element from the stack, convert it to the string representation and print it without sending "end of line". Returning element back to stack.	F	"Hello" print
println	Takes element from the stack convert it to the string representation and print it with sending "end of line" after printing the element. Returning element back to stack.	F	42 println
console/Print	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string to the console without "end-of-line"	F	False (*) ++ 'Result must be false and it is %v' ++ console/Print
console/Println	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string to the console with "end-of-line"	F	(* "Hello world!") console/Println
console/Debug	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string as Debug message	F	(* "Debug message") console/Debug
console/Info	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string as Info message	F	(* "Info message") console/Info
console/Warning	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string as Warning message	F	(* "Warning message") console/Warning
console/Error	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string as Error message	F	(* "Error message") console/Error
console/Fatal	Takes dblock with print data from the stack, format string using first element of the dblock as string with format commands and the rest of the values as formatting data. Sends formatted string as Fatal message and terminates program	F	(* "Fatal message") console/Fatal

Let's review some of the BUND system functions and operators.

Operator	Description	F/O	Example
passthrough	BUND NOOP function. Doing nothing except leaving line in Debug log.	F	42 passthrough
dumpstack	Dumps content of the stack on the STDOUT	F	dumpstack
, (comma)	Drops element from top of the stack. Stack mode impacts which side of the stack are top. In example given, only 42 stays in the stack.	F	42 41 ,
_,	Drops element from top of the stack just like regular comma “,”	F	42 41 _,
,	Drops element from opposite side of the stack. Stack mode impacts which side of the stack are top. In example given, only 42 stays in the stack.	F	41 42 ,,
^	Duplicate element located on top of the stack. In example given two 42 integers will be in the stack.	F	^ 42
! or exec	Takes the reference to the function from top of the stack and execute it	F	[] 42 . !

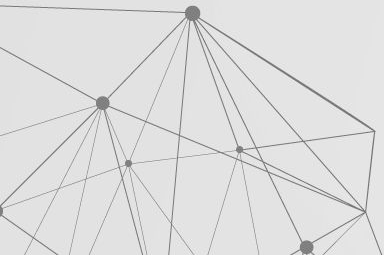


Here we continue to review BUND system level functions and operators.

Operator	Description	F/O	Example
setAlias or ≡	While BUND do not have variables, you can set names to the certain data elements. In example given, you are set alias 'fourtytwo' to data element stored in the stack. The content of the string must conform with NAME format. After you set the alias, you can use this name to return the value on top of the stack. In this example, call <i>fourtytwo println</i> will print 42.	O	42 'fourtytwo' setAlias
alias	Taking an alias to the value stored as string on top of the stack and return the aliased value on top of the stack.	F	dumpstack
sleep	Takes the float number and use this float number as number of seconds for the sleep. For a fraction of the second sleep, you can go below 1.0. In example given, your script will sleep for 1/10 of second. This function doesn't return anything to the stack.	F	0.1 sleep
name	Stores the name of current namespace as string on top of the stack. In example given, stores name of the namespace 'main' and string on top of the stack	F	[main: 42 name ;;
type	Takes the data element from the stack, push it back and push a string representation of that element data type. In example given, the state of the stack after this call will be [42 "int"]	F	42 type
seq	Takes the number from the stack and return data block from the stack containing growing numbers from 0 to up to this number. In example given, the (* 2 ,1 ,0 ,) will be on top of the stack.	F	3 seq
exit	Takes the number from the stack and terminates script with this exit code.	F	0 exit

And finally, let's talk about functors. Functor is a name of the function, that suffixed to the BUND data item, like this: `{“answer”: 42}:(json)`. What is the functor in BUND ? It is a function, that takes the single argument and return a single value. When functor applied to the data, the data ite passed to the function as parameter and returned value will be treated as outcome of the operation. In example given, functor “json” converts string representation of the JSON into a JSON object. Simple. At this point BUND support only handful selection of functors, tailored mostly for string to data conversion, such as:

Functor name	Description
json	Convert string to JSON
file	Convert string to file
glob	Convert string to GLOB pattern
unix	Convert string to UNIX command, execute it and return the standard output



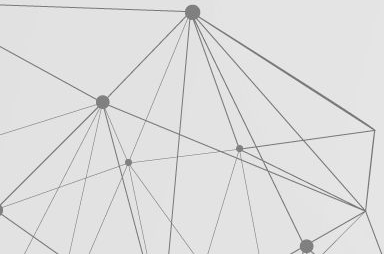
05

LOOPS and CONDITIONALS

Checking for TRUE-th and branching your code.



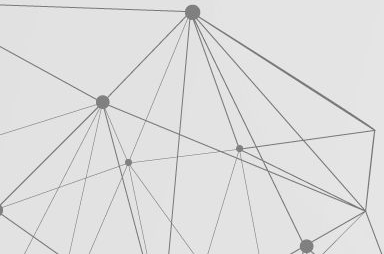
As any language designed to define some algorithms, we can not avoid to create mechanism for testing some conditions and branching. As well as loops. How BUND sees branching and loops is slightly different though.




First, let's introduce idea of the "logical blocks". Logic block is a sequence of the stack data placements and function/operators calls that happens in current stack only if the last element of the stack before definition of the logical block matches logical block type. What does it mean ? This mean, there is a two types of the logical blocks, defined as

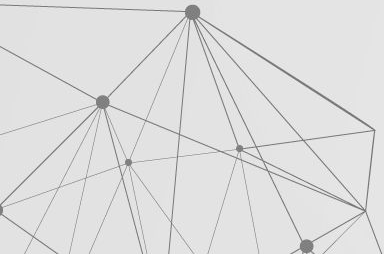
(true <sequence of terms>)
(false <sequence of terms>)

then the <sequence of terms> will be executed only if last value in the stack is matched the type of logical block: boolean True for the "true logical block" and boolean False for the "false logical block" If not executed, logical block doesn't changes the status of the stack.



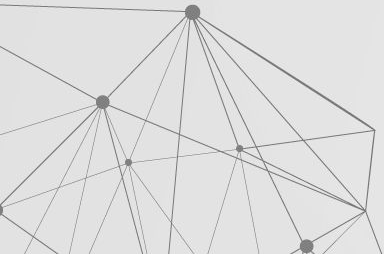
Let me bring some example of logical block power. For a full description of logic operators, see the Chapter 4.

Example	Description
True (true 'This code will be executed' println)	The last value in the stack before true block is True, therefore this code will be executed
False (true 'This code will not be executed' println) (false 'But this one will' println)	Because we do have False in the stack before "true block", this block will not be executed. But stack is not changed and before "false block" we still do have False, "false block" will be executed. After execution of this snippet, you will have on the stack [False, 'But this one will']. Do you know why ? 



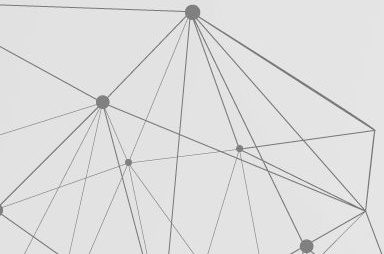
And how True or False may end up on stack ? Through the functions or operators which returning boolean value.

Example	Description
<code>110 < (true 'Yes, 10 more than 1' println)</code>	You can utilize logical comparators <code><</code> <code>></code> <code><=</code> <code>>=</code> and <code><></code> if data type support comparing.
<code>'Hello' g'X*' == (false 'Yes, no match')</code>	Text pattern matching also returns True or False on the stack



There is a number of loop primitives, provided by BUND. First, you do need to know is loop expects to have a reference to a function or operator on top of the stack. Loop routine will execute that reference until loop condition is matched.

loop	Description
loop	Will execute referenced function until application externally interrupted
times	Expects reference and integer. Will execute referenced function number of times defined by the integer.
while	Will execute referenced function until boolean value False will not be found in the stack before next iteration.
until	Will execute referenced function until boolean value True will not be found in the stack before next iteration.



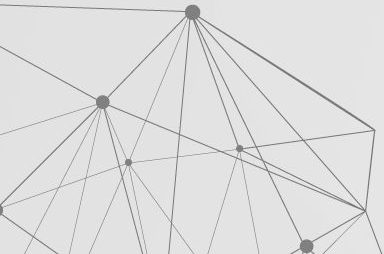
06

FIN

What else can I say ?



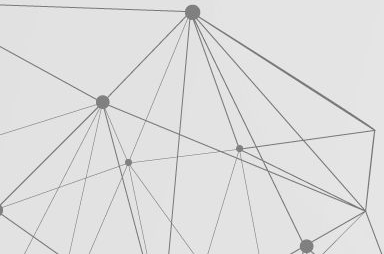
There is no shortages of the programming languages in the world. Some of them, old and gone with a History. Like MUMPS. Some of them, old and still kicking in one way or another, like LISP. Some of them, more or less newcomers in this field, like Julia. But you just learned a few basic ideas about perhaps newest addition into a programming languages ranks - BUND. As all programming languages, this one was created because author feels that "other languages are not expressive enough for my tasks". Or maybe he just want to experiment with some ideas that he have.



Regardless of author initial intentions, he is looking to make BUND to a practical tool, that can be useful and will allow people to create expressive, non-bloated and simply beautiful code.



```
( 'day wonderful a have shall you' )
```





THANK YOU

Does anyone have any questions?

vladimir.ulogov@me.com

