

Composed by Vladimir Ulogov

BUND language programming in 10 minutes

This book briefly introduces a new concatenative programming language based on a paradigm of multiple stacks.

I want to thank my first teacher, who imparted the knowledge and guidance necessary to develop my first programs for the PDP-11 computer.

Introduction

I will introduce a new concatenative programming language called BUND in this work. What is a concatenative language, and how does it differ from the programming languages you're likely familiar with? You're likely acquainted with applicative programming languages like Python, C, or Java. Alternatively, you may have discovered functional programming languages such as Lisp, Haskell, or ML, other examples of applicative programming languages. This category is defined by the way functions are viewed and handled. In applicative languages, a function is treated as a mathematical primitive that computes based on passed arguments and returns a value. In contrast, concatenative programming languages pass a data context from one function to another, external to the function itself. While the stack is the most common method for passing such context, there are concatenative languages that don't utilize a stack. Passing data context enables the concatenation of data processing. Concatenative languages are less known in the software development communities, but you might have heard of languages such as Forth, PostScript, and Factor.

The stack is utilized in many but not all concatenative languages, while applicative languages often use stack structures internally to aid computation. Stacks are indispensable for recursive computation, passing return values computed by functions and storing references to an execution context. What distinguishes concatenative stack-based languages from applicative counterparts is the use of the stack for input data, computational context, and result storage. In essence, everything in concatenative stack-based languages is stored in the stack. In some cases, computational instructions are also stored alongside data on the stack. Since everything, including the context for functions, is stored on the stack, functions in concatenative stack-based languages do not have conven-

tional arguments. Although they function as such, they are often referred to as “words,” as was defined in one of the first concatenative languages to gain popularity - Forth. Another characteristic of concatenative stack-based languages is their reliance on the stack’s Last In, First Out (LIFO) nature. They often employ Reverse Polish Notation (RPN).

So, what will might surprise you in concatenative stack-based language?

- We already mentioned that the functions do not have arguments and no dedicated return value. All input and output data passed to and from the function are passed through the stack.
- You are responsible for ensuring the correct order of the values passed in the data context to the function, as this context is on the stack.
- You are also responsible for interpreting return data placed on the stack. Unlike in the functional language paradigm, there could be more than one return value, depending on your function (or “word”).
- There are no variables. All data are stored on the stack.
- There are no global constants, variables, or values. Everything is on the stack.
- Due to the LIFO nature of the stack, you will deal with RPN.

Why bother?

The concatenative stack-based language might appear unfamiliar at first. The syntax and concepts may seem cryptic, and you must take extra care when preparing the computational context, which could feel like additional effort without clear benefits. However, this class of languages offers substantial benefits that greatly simplify the development and execution of specific computations. It’s important to note that there’s no such thing as a “one size fits all” solution. So, here is my arguments in favor for this concept.

- The concatenative stack-based language promotes the concept of continuation. Your computation flow operates on datasets that are piped to each other. You don’t need variables and constants that you have to

define and babysit all assignments properly. The output of one function, stored on the stack, becomes the input for another function. This approach eliminates the need for curly braces that we all know and love and nested function calls, which are prone to errors. By using concatenative language, you have complete ownership of the data context.

- Concatenative languages feature a straightforward syntax that doesn't necessitate delving into the complexities and mystique often found in applicative languages. I can provide a concise explanation of the fundamentals of BUND syntax in a short amount of time.
- No "goto". We don't need it. Honestly!
- BUND language, as well as many other concatenative languages, supports metaprogramming. You can treat your code as data, and your data may become your code. You can dynamically create anonymous lambda functions and store them as named functions.
- You can create and define lambda functions at runtime and replace already defined ones without restarting your program. With great power comes great responsibility.
- You can redefine system's "words" with your defined lambda functions.
- Thanks to its simple syntax, concatenative programming allows for highly concise code. Minimal effort is needed for data definition or code factoring. Compact, clean code results in programs that are easier to understand, maintain, and debug.
- Concatenative languages are highly interactive and encourage experimentation with the code. Try this out with Java.

Show me the code !

The “Hello World!” program is often the initial program created in any programming language. It aims to display the “Hello World!” message to the standard output. This example illustrates the stack-based nature of BUND. Initially, a string containing the message is placed on the stack. Subsequently, a function is invoked to retrieve the single element from the stack and print it to the standard output (STDOUT).

```
1 // Bund  
2 // This is famous HelloWorld program written in Bund  
3 //  
4 "Hello world!" println
```

What does the word “bund” mean?

The term “bund” derived from German or Yiddish, can be translated to mean “*association*”, “*bundle*”, or “*bunch*”. Throughout history, this word has been used in various contexts. In the context of the multi-stack concatenative programming language, “bund” refers to the capability of the BUND language to separate data and computation across different and distinct storages and execution contexts instead of dealing with a single data source or context.

sin-
gular
Bun-
des,
plural
Bunde

Conception of stack

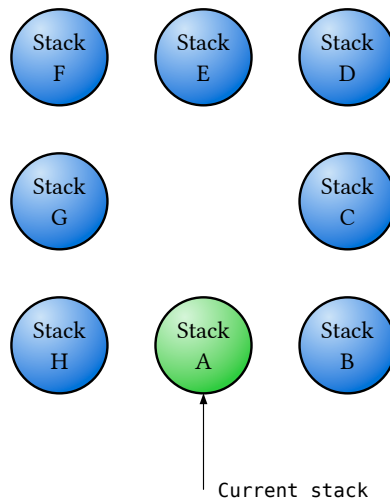
A stack is a fundamental data structure used in computing. It stores values using the Last In, First Out (LIFO) principle, which means that the last data stored will be the first to be retrieved. Stacks are highly efficient for storing data intended for batch processing, as the storage and retrieval process does not require searching or addressing. A stack is a linked list where each element contains references to the previous and next elements in storage, making access to the stored data in the stack linear. Unlike hash tables, the performance of a stack is not degraded with the growth of the number of elements stored. This is a benefit of using a stack, but it also means that all control over data logic is passed to the application developer, which is a limitation of this type of data storage.

BUND is a concatenative, stack-based, dynamically typed, interpreted programming language. Its multi-stack paradigm sets BUND apart from Forth and other similar programming languages. In BUND, the core stack contains references to the stacks that hold the data contexts, which can be anonymous or named. The BUND VM includes a global data stack called the Workbench to facilitate data exchange between stacks. This stack is exclusively used for temporary data storage during computations in different stacks, akin to computations in different data contexts.

You can push data to different stacks and set a stack as the current one using rotation functions or direct positioning. Additionally, you can create new stacks and add them to the stack-of-stacks. It's also possible to push and pull data to and from the current stack and perform various stack operations such as rotation, duplication, and swapping. BUND offers comprehensive support for stack-based data storage while enhancing the stack concept with data context separation.

The concept of Multi-stack

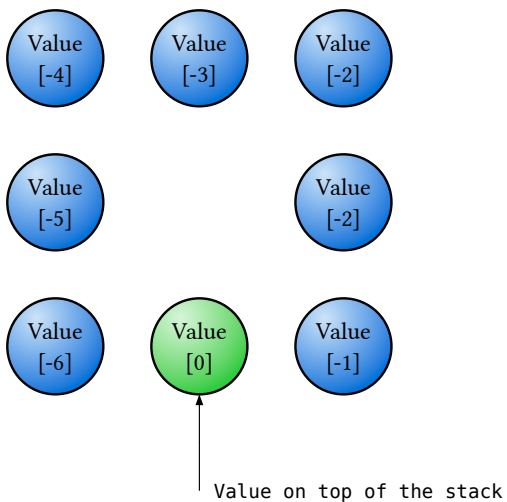
A multi-stack data structure comprises multiple stacks arranged in a circular queue. The elements of this structure store references to data stacks. The current stack is positioned at the end of the queue, with the top of the queue corresponding to the stack on the left of the current one. This configuration enables data to be accessed using a reference to the current stack. Programmatically, the ring of stacks can be rotated to the left or right if needed. A specific stack can also be designated as the current stack by its name, although a user can create “anonymous” stacks where names are randomly generated. This type of structure is commonly called a “stack of stacks.” When a new stack is created, it is pushed to the top of the “stack of stacks” and becomes the new current stack.



The concept of a data stack

The data stack structure is represented by a stack containing dynamically typed values. References on data stacks are stored in “stack of stacks” structure. There are no predetermined requirements on how different data types must be allocated so that you can store data of any kind according to your data processing logic. The data stack resembles the “stack of stacks” structure and acts similarly to a ring buffer that can be rotated to the left and the right. In addition, it possesses all the properties of a LIFO queue. The end of the queue represents the “top of the stack,” and the beginning stores the stack’s oldest element. An element is stored on top of the stack during the PUSH operation, while the PULL operation removes an element from the top.

see
pre-
vious
chap-
ter



Syntax of the BUND language

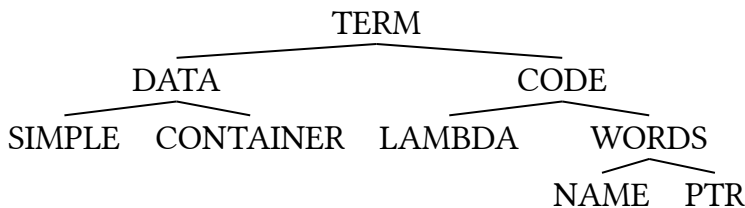
BUND is a concatenative, stack-based programming language inspired by Forth. I've also borrowed a few features from other languages.

- Metaprogramming feature was influenced by Factor programming language.
- The idea of compiling a portable byte code was adopted from Python or Erlang.
- Code-as-data and data-as-code concepts was influenced by Lisp.
- The concept of patching code at runtime was adopted from Erlang.

A sequence of terms represents the BUND program. Terms can be loosely distinguished into two categories:

- Data terms, representing data, stored into stack.
- Code terms, representing the data values related to computation.

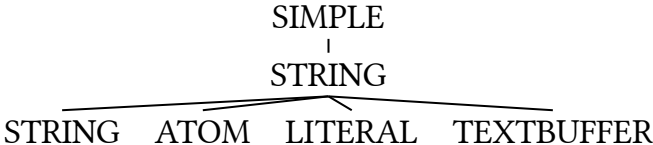
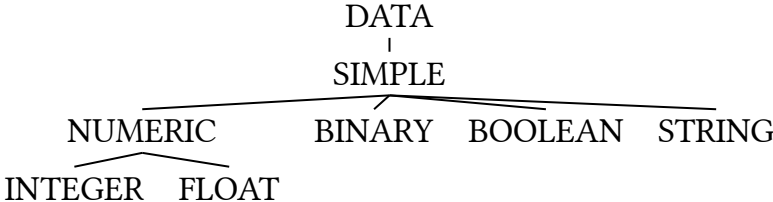
The distinction between these two categories is very loose due to metaprogramming.



BUND simple data types

BUND data values are represented by following dynamic types:

- Integer values, internally represented by **i64** integer number.
- Float-point values, internally represented by **f64**
- Binary values is BLOB's, containing a platform-independent bytecode compiled representation of all supported datatypes.
- Boolean values is a TRUE/FALSE data types, internally represented by **bool**.
- All STRING values internally represented by **String**.



Here is an example of simple data types defined for the BUND programming language.

```
1 // This is example of integer numeric value Bund
2 42
3 3.14      // This is an example of float-point value
4 true      // Those are examples of boolean value
5 false
6 "Hello World!" // Unicode string
7 'Привет Мир!' // Unicode literal
8 // This data type is great for metaprogramming
9 :StringAtom // Unicode atom
```

This example will push to the current stack three values: Integer, Float and String

```
1 // Bund
2 // Pushing data to the stack starting from 42
3 //
4 42 3.14 "Hello World!"
```

This snippet will leave three values in the stack with string “Hello World!” on top of the stack.

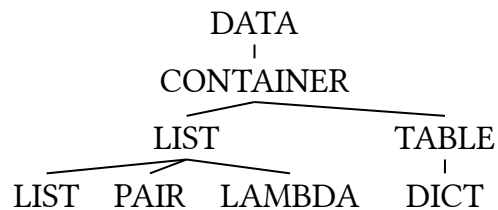
Empty TEXTBUFFER value could be created with the help from word **text**

```
1 // This snippet will create a text buffer Bund
2 // that contains space separated words "Hello World!"
3 text "Hello" , "World!" ,
```

BUND container data types

BUND language supports datatype that could be associated with more than a single data value:

- LIST is a classic in-memory list structure, that can contain any number of SIMPLE, CONTAINER or CODE values.
- PAIR is a limited list, that contains only two values.
- LAMBDA is a special type of LIST, treated by BUND VM as a structure that containing a sequence of instructions passed to VM.



Let me provide an example of how you can create a list and populate it with values. A list is created by enclosing values in square brackets.

```
1 //  
2 // This snippet will create a list value,  
3 // containing three values  
4 // list[0] = 42, list[1] = 3.14, list[2] = "Hello"  
5 //  
6 [ 42 3.14 "Hello" ]
```

Bund

There is no specific syntax similar to creation of the list, designed for creating dictionaries; however, you can create a new empty dictionary using the **dict** word and then populate it using the “**set**” word. Here is an example.

```
1 //
2 // This snippet will create a Dictionary,
3 // and then adds association ANSWER=>42
4 //
5 dict
6   :ANSWER 42 set
```

Bund

You can retrieve a value from a dictionary using the “**get**” word.

```
1 // First, we will create and polpulate dictionary
2 dict
3   :ANSWER 42 set
4 // And sinct now dictionary is
5 // on top of the stack, we can simply
6 // get the value from it.
7   :ANSWER get
```

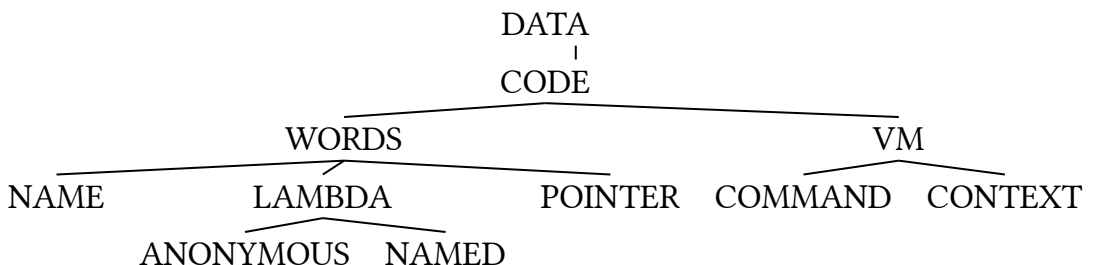
Bund

As the result of this snippet execution, we will have a number 42 on top of the stack.

BUND code data types

BUND language supports following datatype categories that could be associated with execution context:

- “WORDS” refer to data types that act as proper execution references to the system or user-defined functions, also known as “words.” User-defined lambda functions can be either named or anonymous. An example of an anonymous user-defined function is when data values of type LAMBDA are stored on the stack. Conversely, an example of a named user-defined “word” is when a LAMBDA value is associated with a specific name.
- VM-related data types encompass CONTEXT, which are data values instructing the VM to set the current stack to a stack with a specific name, and COMMAND, which denotes data values referring to instructions that alter the state of the VM rather than processing data.



You already seen the examples of a **NAME** values ether being stored in the stack, or just passed to VM for evaluation.

```
1 //  
2 // This code snippet will perform mathematical add  
3 // and leave numeric INTEGER value "4" on top  
4 // of the stack  
5 //  
6 2 2 +
```

Bund

In this example, the first two data values are of type INTEGER, and the third is a NAME reference to the function “+” that performs a math operation.

Sometimes, it’s necessary to refer to a function without executing it. This reference is known as a POINTER. When passed to a VM, the referred function isn’t immediately executed; instead, it’s treated as regular data until you explicitly call for its execution. You can define pointer placing LISP-style “backtick” in the front of the function name.

```
1 //
2 // Unline in previous example, "+" is not a function
3 // but pointer to the function
4 //
5 2 2 `+ Bund
```

The actual “+” function will not be executed in this example. Instead, the pointer to the “+” function will be stored on top of the stack. You can use the “execute” function or the word “!” to execute the function referred to by the pointer. These will take a POINTER (or LAMBDA) from the stack and execute it.

```
1 //
2 // Now we executing "word" using POINTER
3 // stored on top of the stack
4 //
5 2 2 `+ ! Bund
```

This snippet will leave INTEGER with value “4” on top of the stack.

You can dynamically create pointers without verifying if the function even exists by using the word “ptr”.

```
1 //
2 // In this example, we creating POINTER
3 // by porovidin string to the word
4 // ptr
5 // stored on top of the stack
6 //
7 2 2 :+ ptr !
```

Bund


To ensure that the function you are pointing in your pointer to exists, you can use “resolve.” This will raise an error if you try to point to a function that doesn’t exist.

```
1 //
2 // In this example, we creating POINTER
3 // by porovigin string to the word
4 // resolve that will create POINTER
5 // only if function exists
6 //
7 2 2 :+ resolve !
```

Bund

Next, I'd like to discuss switching between different data contexts in the BUND language. As you learned earlier, BUND operates on a concatenative stack-based Virtual Machine with a multi-stack paradigm. In contrast to classic Forth-style VMs and languages, BUND operates on multiple named stacks, also referred to as "contexts," with their references stored in a "stack-of-stacks." Switching between named contexts is simple: you provide the stack's name, prepended with "@". This action makes a stack with this name the "current stack."

```
1 //
2 // This is demonstration of swiching between
3 // named stacks, also known as "context"
4 //
5 @A      // Making stack with name "A" current.
6         // Create it if necessary
7 1 2 3   // Three INTEGER values goes into
8         // stack "A"
9 @B      // Make stack "B" current
10 42     // Push number 42 on top
11       // stack "B"
```



Data in different stacks are isolated from each other.

All about anonymous and named lambdas

Metaprogramming allows to treat a programs as data and I already mentioned, that the LAMBDA functions is just a special sub-type of a LIST data type. Therefore, you can treat your function, just as data and create lambdas by pushing values into it

```
1 //  
2 // You can create lambda functions on-the-fly  
3 //  
4 lambda 42 + !
```

Bund

This code snippet creates an empty lambda function that pushes the integer value 42 to the current stack and executes it. As a result, the number 42 will be on top of the stack after the snippet is executed. However, creating the lambda function using specialized “syntax sugar” would be more convenient.

```
1 //  
2 // There is a better way to create and execute lambda  
  functions  
3 //  
4 { 42 } !
```

Bund

Any content enclosed within curly braces will be incorporated into an anonymous lambda function stored on the stack. This code snippet achieves the same functionality as the previous one but is more aesthetically pleasing and concise.

Anonymous lambdas play a crucial role in the logical functionality of the BUND. For instance, when using conditional execution with “IF,” a TRUE or FALSE boolean condition is computed and stored on the stack, along with a LAMBDA function that will be executed if the condition is TRUE. Here is an example:

```
1 // Bund
2 // There is a better way to create and execute lambda
  functions
3 //
4 42 42 == { "Yes, 42 is equal to 42" println } if
```

In this example, we start by pushing two integer numbers to the stack and then perform an arithmetic comparison, which will place a boolean outcome on the stack. The “IF” function will take two values from the stack - a lambda function and a condition. If the condition is TRUE, the lambda is executed.

I want to present a method for converting an anonymous lambda, currently stored on the stack, into a named lambda that can be stored in a table of lambda functions. This enables you to refer to or execute the lambda by name at any time.

```
1 //
2 // This is how you declare a named function
3 //
4 :FortyTwo {
5     42          // We leave 42 on stack
6 } register
7 // Then we can execute
8 // the named lambda as any other function
9 FortyTwo
```

Bund

Function aliases

BUND provides the capability to create an alias for a function name. Aliases do not alter the function but allow assigning alternative names to them.

```
1 //
2 // This is how you declare a named function
3 // and then create an alias
4 //
5 :FourtyTwo {
6     42          // We leave 42 on stack
7 } register
8 // After registering, we can create alias
9 // to the function
10 :FourtyTwo :answer alias
11 // Then we can call the funciton by it's alias
12 answer
```

Bund

The outcome of this snippet will be registering both, named lambda and the alias for the named lambda.

Autoadd feature of BUND VM

Now, let's delve into the BUND language's auto-add feature. When applying a value to the BUND VM, one of two outcomes occurs: the value is either pushed to the stack or treated as a function call, and the function is executed. But what if you want to modify the values already on the stack? The dynamic values of BUND support the inner PUSH operation, allowing you to push data inside values of CONTAINER types. To facilitate these operations, BUND provides two words that do not push data to the stack or execute words but alter the VM machine's behavior. The word ":" activates the auto-add feature. With auto-add turned on, the VM checks if a value that supports the inner PUSH operation is on top of the stack. If it is, instead of executing or pushing data to the stack, the BUND VM performs an inner PUSH, pushing data inside the CONTAINER value. The word ";" deactivates auto-add and restores the VM to its normal state.

```
1 //
2 // First, let's place an empty list on the stack
3 //
4 list : // Then when we call ":",
5         // we do turn on auto-add
6     1 2 3 // The values are
7         // list[0] = 1
8         // list[1] = 2
9         // list[2] = 3
10        // pushed to the LIST object.
11        // This is an inner PUSH operation
12 ;      // Then we turn off auto-add
```

Bund

The provided code snippet leads to placing a LIST object containing values [1, 2, 3] at the top of the stack.

Basic functions from BUND's standard library

The scope of this manual will not allow me to cover the entire library, which is already quite extensive in the early stages of the project and will continue to grow. However, this brief introduction to some functions will help you grasp the essence of the language and hopefully spark your interest. You have already been introduced to some functions, such as:

Word	Description
list	Adding an empty LIST value to the top of the stack
dict	Adding an empty DICTIONARY value to the top of the stack
lambda	Adding an empty LAMBDA value to the top of the stack
text	Adding an empty TEXTBUFFER value to the top of the stack
if	Performing conditional execution of the anonymous lambda
println	Removing value from top of the stack and printing it to STDOUT
,	Perform inner PUSH to a TEXTBUFFER object with added spaces
register	Register anonymous lambda function as named lambda
alias	Create function alias
set	Set value inside DICTIONARY
get	Get data from DICTIONARY
:	Turn auto-add ON
;	Turn auto-add OFF

The “words,” also referred to as functions defined within the standard compile-in library, can be categorized as follows:

Category	Description
VM words	Words that alter the behavior of the BUND Virtual Machine. Currently, only functions associated with the auto-add feature fall within this category.
Data creation	Words that creating a values on top of the stack
I/O category	Words providing means to exchange information with outside world
Stack management	Managing stacks list, stack state and values on the stack
Application logic	Conditionals, loops and other functions for maintaining an execution logic of a program
Data processing	Words that are handling and processing the data that is stored on the stack

In the next stage, I aim to present an example of a function from each category. I will at some point in the future will develop a thorough BUND Standard Library manual. Until then, I encourage you to review the code.

VM word: Auto-add on

The word “:” will enable auto-add feature for BUND VM. It doesn’t change the state of the stack, but alter state of the VM.

```
1: function :()
2:     ▷ Changing state of BUND VM
3:     if VM::autoadd = FALSE then
4:         VM::autoadd ← TRUE
5:     else
6:         return Error(“Nested auto-add is not supported”)
```

VM word: Auto-add off

The word “;” will restore auto-add status of the VM to default state

```
1: function ;()
2:     ▷ Changing state of BUND VM
3:     if VM::autoadd = TRUE then
4:         VM::autoadd ← FALSE
5:     else
6:         return Error(“Nested auto-add is not supported”)
```

Data creation word: Creating an empty LIST

The word “list” will return an empty value of LIST type on top of the stack.

- 1: **function** LIST()
- 2: ▷ Add an empty LIST value
- 3: *current stack* ← Value::list()

I/O word: printing the value taken from stack

The word “println” will take a value from the stack and print it to STDOUT. No value returned to the stack.

```
1: function PRINTLN()  
2:   ▷ Print value to STDOUT  
3:   Value ← current stack  
4:   if Value = None then  
5:     return Error(“Stack is too shallow”)  
6:   Output ← Value::conv (STRING)  
7:   if Output = None then  
8:     return Error(“Conversion error”)  
9:   PRINT(Output)
```

Stack management word: duplicating data on the stack

The function “dup” will duplicate alue located on top of the stack. Two values, original one and duplicated are returned to the stack.

```
1: function DUP()  
2:     ▷ Duplicating value that is on top of the stack  
3:     Value ← current stack  
4:     if Value = None then  
5:         return Error(“Stack is too shallow”)  
6:     Value2 ← Value::dup (Value)  
7:     current stack ← Value  
8:     current stack ← Value2
```

Application logic word: the “loop” function

The “loop” function retrieves two values from the stack: an anonymous lambda function and a list. Subsequently, it iterates through the list of elements, pushing each element to the stack and executing the lambda function after each push.

```
1: function LOOP()
2:     ▷ Get the anonymous lambda from stack
3:     l ← current stack
4:     ▷ Get the list of values from stack
5:     v ← current stack
6:     while e ← v do
7:         ▷ Push value from list to the stack
8:         current stack ← e
9:         ▷ Execute anonymous lambda function
10:        l()
```

Data processing word: the “string.upper” function

Word “string.upper” as name suggests converts the case of string value to uppercase. The string value is taken from stack and string value returned to stack.

- 1: **function** STRING.UPPER()
- 2: ▷ Get string value from stack
- 3: $v \leftarrow \textit{current stack}$
- 4: ▷ Return converted value to the stack
- 5: $\textit{current stack} \leftarrow \mathbf{string.upper}(v)$

Conclusion

BUND is a very new language. It is currently in its early stages of development, and the language's runtime has many limitations. The standard library requires improvement, and the author or contributor must address several potential bugs. However, the *bundcore* crate and its dependencies have successfully passed all their test cases, which is a promising sign. Although the language is simple and its underlying dependencies are generally stable, there are no guarantees against critical bugs. The license is attached for reference. While concatenative, stack-based programming languages are not widely used in general programming practices, they have stood the test of time and deserve more attention from the software development community. BUND aims to address design gaps in this concept, and the author hopes to spark interest with his ideas and inspirations that brought BUND into existence.

You can get in touch with my via [in](#) my LinkedIn profile.
The BUND project is hosted on my GitHub page [vulogov](#)



License

Apache License Version 2.0, January 2004 <http://www.apache.org/licenses/>

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

“License” shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

“Licensor” shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

“Legal Entity” shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, “control” means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

“You” (or “Your”) shall mean an individual or Legal Entity exercising permissions granted by this License.

“Source” form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

“Object” form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

“Work” shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright no-

tice that is included in or attached to the work (an example is provided in the Appendix below).

“Derivative Works” shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

“Contribution” shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, “submitted” means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as “Not a Contribution.”

“Contributor” shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as

stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a “NOTICE” text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You dis-

tribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. **Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. **Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. **Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. **Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for dam-

ages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. **Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets “[]” replaced with your own identifying information. (Don’t include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same “printed page” as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or im-

plied. See the License for the specific language governing permissions and limitations under the License.

Content

Introduction	3
Why bother?	4
Show me the code !	6
What does the word “bund” mean?	7
Conception of stack	9
The concept of Multi-stack	11
The concept of a data stack	12
Syntax of the BUND language	13
BUND simple data types	14
BUND container data types	16
BUND code data types	18
All about anonymous and named lambdas	22
Function aliases	25
Autoadd feature of BUND VM	26
Basic functions from BUND’s standard library	27
VM word: Auto-add on	29
VM word: Auto-add off	29
Data creation word: Creating an empty LIST	30
I/O word: printing the value taken from stack	31
Stack management word: duplicating data on the stack	32
Application logic word: the “loop” function	33
Data processing word: the “string.upper” function	34
Conclusion	35
License	37
