http://doi.org/10.35784/iapgos.2901 *received: 16.03.2022 | revised: 01.06.2022 | accepted: 15.06.2022 | available online: 30.06.2022*

# CKRIPT: A NEW SCRIPTING LANGUAGE FOR WEB APPLICATIONS

**Wiktor Kania, Radosław Wajman**
Lodz University of Technology, Institute of Applied Computer Science, Lodz, Poland

***Abstract.*** *The project aimed to develop and implement an efficient web server in the C++ programming language. A highly concurrent network server was achieved using system calls such as polls and a limited number of threads. The server has built-in support for a new scripting language called Ckript. It is an original project that exposes most of the server's functionality and is the primary way of developing back-end web applications. Ckript is an interpreted language with a strong typing system, garbage collection, semi-manual memory management, first-class functions, explicit variable references, support for certain object-oriented patterns and many others. In the article the syntax of the language but also the environment architecture has been explained. Finally, the testing procedure has been described with the results' presentation and discussion at each step.*

**Keywords**: HTTP server, scripting language, interpreter, parser, garbage collector, C++

## CKRIPT: NOWY JĘZYK SKRYPTOWY APLIKACJI INTERNETOWYCH

***Streszczenie.*** *Celem projektu było opracowanie oraz implementacja wydajnego serwera przy użyciu języka C++. Zastosowanie wywołań systemowych oraz ograniczonej liczby wątków pozwoliło zbudować wysoko współbieżny serwer. Posiada on wbudowane wsparcie dla nowego języka skryptowego Ckript. Jest to projekt autorski, który udostępnia większość funkcjonalności serwera i jest głównym środkiem budowania aplikacji back-endowych. Ckript to język interpretowany z systemem silnego typowania, mechanizmem porządkowania pamięci, półautomatycznym zarządzaniem pamięcią, wbudowanymi funkcjami, referencjami do zmiennych, obsługą pewnych wzorców zorientowanych obiektowo i wiele innych. W artykule wyjaśniono składnię języka, ale także architekturę środowiska. Na koniec opisana została procedura testowania wraz z prezentacją wyników i dyskusją na każdym etapie.*

**Słowa kluczowe**: serwer HTTP, język skryptowy, interpreter, parser, garbage collector, C++

## Introduction

PHP was one of the first programming languages for building dynamic web applications, even though it has a mixed reputation. However, it is still one of the favourite programming languages due to its simplicity and approach to solving challenges associated with building web applications. When run on a web server such as Apache or nginx, PHP shares many similarities with the project.

This work aimed to build a highly efficient HTTP server written in C/C++ called Mish. To achieve this, research on parallel and concurrent programming techniques was done to find optimal ways to handle a few thousand connections at a time. Ckript is a general-purpose scripting language developed to integrate it with the server's architecture and make it more suitable for back-end development. Ckript has been inspired by other programming languages such as C, JavaScript, and Go. It is an interpreted language with a strong typing system, garbage collection, semi-manual memory management, first-class functions, explicit variable references, and support for certain object-oriented patterns. What makes Ckript different from most other programming languages is its support for high-level and low-level programming and integrated type system while remaining a dynamically interpreted language. Most scripting languages abstract away things like memory management and variable references. Ckript allows the programmer to choose whether to allocate data on a stack or the heap. The programmer can also use explicit references to other variables making it possible to choose between passing by value and passing by reference. This is usually only possible in low-level programming languages such as C/C++, most high-level languages pass all compound type (arrays, objects) variables by reference and primitive types (integers, Booleans) by value. Manual memory allocation is also exclusive to lower-level languages.

The integrated type system and strong typing allows writing more error-free and readable code. Usually, only compiled languages use static typing, scripting languages often opt instead for dynamic typing with the possibility of installing extensions to enable type annotations (Typescript, Ruby RBS).

A series of benchmarks was run with the implemented HTTP server to compare the server's capabilities to existing and widely used solutions. The whole project was written in C/C++ in the C++17 standard. C++ was chosen due to its high performance, extensive standard library, and access to some low-level network functionality.

## 1. Ckript syntax characteristics

### 1.1. Variables type system

Ckript is strongly and statically typed. Each variable needs to have a type assigned before the program can run. There is no type inference, and a variable's type cannot change during runtime. Since Ckript is an interpreted language, type checking is done at the program's execution time. Even though the language is strongly typed, some features are implemented to simplify a developer's life, such as implicit casts, e.g., when adding an integer to a float. Ckript defines 7 built-in types:

- **int** – a 64-bit signed integer value,
- **double** – a double-precision floating-point value,
- **str** – a string of characters used to represent human-readable text,
- **bool** – a boolean (false or true) value,
- **func** – a first-class function,
- **arr** – a dynamic array capable of holding multiple different values of the same type,
- **obj** – an instance of a class containing methods and fields.

There's also the *ref* keyword which can be coupled with any of the type mentioned above to indicate a reference to a type e.g., *ref int*.

Additionally, each variable declaration requires an assignment. There are no default or null values. This makes the whole language null-safe and less error-prone. Early versions of Ckript allowed provoking null pointer errors through invalid use of manual memory management e.g., by accessing a dynamically allocated variable after deallocating it. Situations like this are not possible after adding garbage collection and removing the ability to deallocate variables manually.

### 1.2. Functions

All functions in Ckript are first-class functions meaning that they are treated like any other variable. Functions can be passed as arguments to other functions, returned by other functions, stored in variables or data structures such as objects and arrays. There is support for anonymous functions (also known as lambdas in languages like Python or C++) and immediately-invoked function expressions making Ckript a very flexible language. Since functions can be stored in variables, a decision was made not to implement named functions.

Ckript functions must respect the language's type system, which means that all function expressions must define types for their parameters and return values. A function declaration and invocation might look like this:

```
1  func fib = function(int n) int {
2    if (n <= 1) return n;
3    return fib(n - 1) + fib(n - 2);
4  };
5
6  println("fib(10) =", fib(10));
```

*Fig. 1. Ckript functions*

This expression declares a function that accepts one parameter of name n and type int and returns a value of type int.

```
1  int six = (function(int a, int b) int {
2    return a * b;
3  })(2, 3);
4
5  println(six);
```

*Fig. 2. An immediately-invoked function*

This is an example of an immediately-invoked function expression. The function is executed right after declaration without storing it in a variable, instead the result of the function is stored.

```
1  func runTenTimes = function(func f) void {
2    int i = 0;
3    for (; i < 10; i += 1) {
4      f();
5    }
6  };
7
8  func greet = function(void) void {
9    println('Hello');
10 };
11
12 runTenTimes(greet);
```

*Fig. 3. The usage of a first-class function*

Fig. 3 presents an example of a function being passed to another function by argument. The above code should display the message "Hello" ten times on separate lines.

## 1.3. Variable scope

The scope of a variable is the part of a program where the variable's name can be used to access it. In Ckript, all variables are local to the function they were declared in by default and can be only accessed in that function. While there is no way to make a variable visible outside the function, there is a way to make a function capture outside variables by adding the > operator. See the function expression in line 3:

```
1  double PI = 3.1415;
2
3  func f = function>(void) void {
4    println(PI);
5  };
6
7  f();
```

*Fig. 4. Capturing variables*

Even though the language makes this possible, the preferred way of passing values between functions is by arguments or return values as it minimizes the number of possible human-errors, hence why functions don't capture outside variables by default.

An exception to the rule are the function variables. When a function is assigned to a variable, its name is pushed onto its own stack to allow recursion. Ckript allows for variable shadowing.

There is no block scope in Ckript i.e., variables declared in statements like if, while, or for are still local to the function they were declared in.

## 1.4. Classes and objects

Ckript allows defining classes and instantiating objects. Ckript objects are very similar to structures from C – they are composite data types that allow a programmer to hold several variables under the same name. These variables can be of any type.

Even though Ckript functions are first-class functions and are treated like any other variable, there is a way to treat a function as a method. When an object is allocated on the heap and one of its fields is of *func* type, the reference to that object is bound to the function. The virtual machine creates a new variable of type *ref obj* and named ***this*** and pushes it onto the function's stack, allowing it to reference the object it is assigned to.

```
1  class Person(
2    str name,
3    int age,
4    func greet
5  );
6
7  alloc obj Wiktor = Person('Wiktor', 23, function(void) void {
8    println(this.name, 'is', this.age, 'years old');
9  });
10
11 Wiktor.greet();
```

*Fig. 5. The usage of Ckript classes*

Line 8 references the object allocated on line 7 by using ***this*** keyword. If the variable Wiktor was not allocated with the **alloc** keyword, the program would throw an error on line 11 saying that 'this' is not defined.

Class declarations are treated like variable declarations. Classes are allocated on the current function's stack and are local to it. There are no complex object constructors, instead a programmer is encouraged to write their own functions that can act as constructors. This pattern is commonly found in languages such as C and Go.

```
1  alloc func newUser = function(str nickname, str email) ref obj {
2    class User(
3      str nickname,
4      str email,
5      bool verified
6    );
7    alloc obj user = User(nickname, email, false);
8    return user;
9  };
10
11 ref obj bob = newUser('bob', 'bob@email.com');
12 println(bob);
```

*Fig. 6. Constructor's example in Ckript*

## 1.5. Standard library and other properties

Ckript includes a small standard library for most common programming tasks such as file I/O, math operations, string manipulation, and type conversions. The full list may be found in the README.md file on the Ckript GitHub repository. Follow the repository's URL in the chapter 5.

There are also several mathematical functions such as sin, sinh, cos, cosh, tan, tanh, sqrt, log, log10, exp, floor, ceil, round, pow, and abs.

Ckript strings can be concatenated by using the + operator. One of the convenient features of the language is the ability to format strings.

```
1  str tag = "<@1>@2</@1>";
2  str div = tag('div', 123);
3  println(div);
```

*Fig. 7. Strings concatenation and formatting*

Line 1 declares a string variable. Line 2 executes the string like a function and passes two arguments. The corresponding arguments will replace the placeholders @n. Finally, execution of line 3 will print " 123 ". Since this method allows

any type of arguments to be passed, it is more than a mere shorthand for writing functions for string formatting. Functions need to specify the type of their arguments, so the same functionality wouldn't be possible without major code duplication.

Ckript allows creating arrays of any built-in type, including arrays and objects (see Fig. 8). Arrays in Ckript are strongly typed, meaning that each array can hold values of one type, and it's not possible to mix different types together. To create an array in Ckript, the array keyword must be used. Optionally, initial elements inside the parentheses can be defined. At the end of an array declaration there must be the type of its elements defined explicitly.

It is possible to check the size of an array with the size function, iterate over it, and print each individual element. To append or prepend a new element to an array, it simply needs to be added to it. To remove an element from an array, the index of the element can be subtracted from the array.

```
arr numbers = array(1, 2, 3) int;
println("numbers:", numbers);

int i = 0;
for (; i < size(numbers); i += 1) {
  println("numbers[@1] = @2"(i, numbers[i]));
}
```

```
Ckript console
numbers: array<int>(1, 2, 3)
numbers[0] = 1
numbers[1] = 2
numbers[2] = 3
```

*Fig. 8. The usage of arrays (left) and its output (right)*

## 2. The environment architecture

### 2.1. Memory allocation

Most high-level scripting languages do not give the choice of whether to allocate a variable on the stack or heap. In Ckript, all variables are allocated on function stacks by default. However, the ***alloc*** keyword instructs the interpreter to allocate the variable on the heap. What's the difference? There are multiple stacks (one for each function), but only one global heap. Whenever a function finishes execution, its stack is destroyed, and all the variables allocated on it are popped. When a variable is allocated on the heap, it remains there as long as some part of the program has at least one reference to it. It means, it may still be reachable. Variables without references are eventually garbage collected.

Using references also avoids the problem of unnecessary data duplication when passing variables to functions – by default, all variables are passed by copy. Passing by reference is especially useful when dealing with bigger objects or arrays.

```
1  func passByCopy = function(int a) void {
2    a += 5;
3  };
4
5  func passByReference = function(ref int a) void {
6    a += 5;
7  };
8
9  int foo = 3;
10 alloc int bar = 3;
11
12 passByCopy(foo);
13 passByReference(bar);
14 println(foo, bar);
```

*Fig. 9. The usage of references in Ckript*

Both *passByCopy* and *passByReference* increment the argument *a* by 5, but *passByReference* uses the *ref* keyword on line 5 to indicate that it accepts a reference to an int. Line 12 will copy foo, and the value of foo will remain the same after the function execution. Line 13 will copy the reference to bar, and the value of bar will change. Line 14 will output "3 8".

### 2.2. Bump allocator

Ckript implements a bump allocator on top of C++'s standard allocator. A bump allocator is simple in design – it can only grow and never shrink. It maintains a list of free chunks (a free list). Whenever a chunk is freed, it is placed on the free list and marked as "free" on the heap as in free to use, but not deallocated. Whenever a new chunk is requested, a chunk from the top of the free list is returned. If there are no free chunks, the allocator allocates a new chunk and expands the heap. This approach enables very fast allocation and free operations, but never allows the heap to shrink during the program's execution time. Since the Ckript programming language is intended for developing back-end scripts that shouldn't take more than a couple of hundred milliseconds, this trade-off bears more positives than negatives. The implementation of such allocator is straightforward and much less error prone.

### 2.3. Garbage collector

The garbage collector (GC) in Ckript is based on the mark-and-sweep algorithm [2]. The idea behind it is very simple – during the mark phase, it goes through all the variables on stacks and marks them as reachable. The sweep phase goes through all the variables on the heap and deallocates all the variables that are not marked as reachable. If a variable is not reachable in any of the program stacks, it will never be reachable, and it can be safely deleted from the heap. The GC in Ckript uses the previously described bump allocator interface to free variables, so they're not deleted, making it faster. This matters because it is a stop-the-world garbage collector, which means that once it runs, the whole program execution halts until garbage collection is finished. Ckript's garbage collector has threshold before looking for data to free. At first, it waits for at least five chunks to be allocated on the heap before it runs and then adjusts this threshold dynamically depending on how many chunks there are.

### 2.4. Lexer

Before the source code can be executed, it needs to be transformed into a more computer-friendly format [2]. The Ckript interpreter uses a linear sequence of specialized components to achieve that, namely the lexer, parser, and evaluator. The lexer performs lexical analysis (also known as tokenization). It is the process of transforming source code, which is a string of characters, into a series of tokens. Source code often contains information that is not very useful to a computer, such as whitespace and comments, and many of the keywords are unnecessarily verbose.

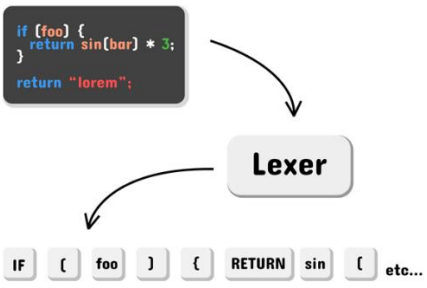


*Fig. 10. The outline of the lexer's workflow*

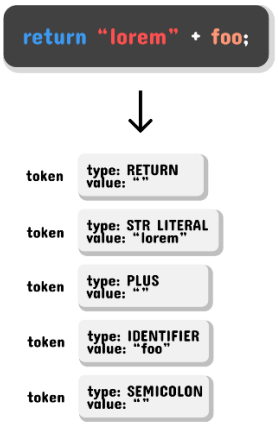


*Fig. 11. The conversion of the source code to tokens*

A single token is a bit more complex. It holds information about its type, a value, and some metadata such as the source file and line to help build meaningful error messages. Tokens of type **`FUNCTION`**, **`RETURN`**, **`TRUE`**, **`FALSE`** etc. are self-contained and don't need to store more information since they're used to describe language keywords. The enum type defining the type of a token is implemented as an integer value in the C++ language making tokens lightweight to use and store. Tokens such as **`STRING_LITERAL`**, **`DECIMAL`**, or **`IDENTIFIER`** need to store additional information about the contents of the token. That's where the value field comes in. See in Fig. 11 a string literal such as `"lorem"` in the example would be transformed to a token of type **`STRING_LITERAL`** and value `"lorem"`.

The Ckript lexer iterates through source code character by character and groups them. A group of characters that means something is called a lexeme. When a new lexeme is found, it is transformed into a token. A set of rules that determines how a language groups characters into lexemes is called a lexical grammar. The Ckript grammar is very similar to that of the C programming language.

## 2.5. Parser

The parser is responsible for syntax analysis. It takes a series of tokens as input and transforms them into an Abstract Syntax Tree (AST). An AST is a tree structure that consists of nodes. Each node can have child's nodes. An AST does not contain information such as braces, semicolons, or parentheses. The structure of an AST is designed in a way to represent these implicitly.

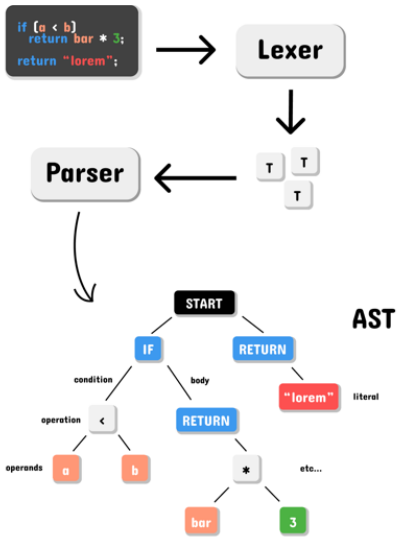


*Fig. 12. The outline of the parser's workflow*

The Ckript parser distinguishes three types of nodes:

- Declaration – a piece of code that introduces a new variable or class i.e., int number = 5;
- Expression – a piece of code that evaluates to a value e.g., a mathematical operation i.e., sin(5)*a;
- Statement – any executable piece of code. That includes expressions and declarations.

In Ckript, all nodes are some kind of statement nodes. Ckript defines a program as a set of statements, that's why treating everything as a statement makes it easier to execute code.


*Fig. 13. The grammar of a declaration statement*

Every **declaration statement** must start with a type, followed by an identifier, followed by an equal sign, followed by an expression, and end with a semicolon.

Every **expression statement** must start with an expression and end with a semicolon.

Every **compound statement** (a statement that can contain multiple other statements) must start with a left brace {, then contain one or more statements, and end with a right brace }.

Every **while statement** must start with the `"while"` keyword, followed by a left parenthesis, followed by an expression, followed by a right parenthesis, and end with a statement.

Expressions are also a complicated concept in Ckript since a lot of code can fall under that category, and most expressions are made up from other expressions.


*Fig. 14. An example of a binary expression*

Binary expressions are made up from two expressions separated by a binary operator (such as `"+"`, `"-"` etc.). The binary expressions can also contain compound expressions on either of the sides, such as in the following example (Fig. 15) where the right-hand side of the binary expression is another binary expression.


*Fig. 15. Compound binary expression*

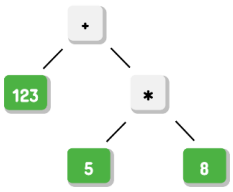


*Fig. 16. The tree representation of an expression*

Parsing mathematical expressions introduces two new problems to solve – operator precedence and associativity. Ckript's operator precedence was heavily inspired by the C++ operator precedence [3]. If there was no precedence, an excessive number of parentheses would be required to write valid mathematical expressions. Thanks to operator precedence, the parser knows that, for example, multiplication should be evaluated before addition. An expression like 123+5*8; should be parsed to an AST node like this in Fig. 16:

By traversing the tree recursively from left to right and evaluating it, the correct result will be calculated.

To parse an expression to a tree, the *Shunting Yard Algorithm* [5] can be used. Ckript uses a slightly modified version of the algorithm that parses tokens to *Reverse Polish Notation* [1] (RPN). For example, the expression `"1 + 2 * 3 + 4"` will be converted to `"1 2 3 * + 4 +"` which can be later reduced to the actual value with an RPN calculator.

## 2.6. Evaluator & Virtual Machine

The evaluator is the place where the actual program execution happens. It takes an AST as input, traverses it, and executes each statement node. Another important part of the evaluator is the expression evaluator which takes an RPN stack and evaluates it to a value. There are also functions responsible for declaring variables, classes, executing built-in and user-defined functions, instantiating objects etc. The evaluator is closely tied to the virtual machine.

The virtual machine (VM) is the place where all the data is stored during program execution. The VM contains the global

heap, the stack trace, output buffers, references to all the evaluators using the VM, and the reference to the client that made the HTTP request that invoked the script. Usually, there are multiple evaluators (one for each executed function), but only one VM. Each evaluator instance holds the reference to the same VM so that all functions can access the exact same data, and the VM keeps track of all the active evaluators to enable garbage collection (since function stacks are stored in their respective evaluators). The VM also defines all the built-in (standard library) functions available globally during program execution, such as **`echo()`**, **`abort()`** etc.

For example, whenever an evaluator defines a new variable on the heap, it needs to use the associated VM to store the variable's value.

The Value class models all available data types, such as integers, strings, arrays etc. A value can be either stored in a Variable object or a Chunk object on the heap, depending on whether it was allocated on the heap or a stack. Values that live on stacks are returned as variables that hold the actual values. Values allocated on the heap are returned as variables with a heap reference value, which basically acts as a pointer to the location of the Chunk object on the heap that holds the actual value.

The Heap class holds an array of Chunk objects, it also has a Cache instance that holds references to unused Chunks.

The NativeFunction class in an abstract class defines a standard library function. Other classes can implement their own ***execute*** methods – see Fig. 17. The standard library is a map of key-value pairs where the keys are the names of the functions, and the values are NativeFunction pointers - each having its own implementation.

```
297  class NativeEcho : public NativeFunction {
298    public:
299      Value execute(std::vector<Value> &args, std::int64_t line, CVM &VM) {
300        if (args.size() == 0) {
301          VM.throw_runtime_error("echo() expects at least one argument", line);
302        }
303        std::size_t i = 0;
304        const std::size_t end_index = args.size() - 1;
305        for (auto &arg : args) {
306          VM.output_buffer += VM.stringify(arg);
307          if (i != end_index) VM.output_buffer += " ";
308          i++;
309        }
310        return {Utils::VOID};
311      }
312  };
```

*Fig. 17. C++ source code of the echo function implementation*

For example, the echo function accepts at least one argument and saves the string representation of all of them to the output buffer (which is later rendered to the client) and returns void.

The CVM (Ckript Virtual Machine) class is the heart of the whole VM. It is closely tied to the Heap object, contains garbage collection utility methods, keeps track of stack traces, and can throw all kinds of errors.

## 3. Tests and results

To test the performance of the HTTP server, **Siege** [4] utility program was used. The tests compared Ckript scripts running on the server with similar PHP scripts running on an Apache server.

The machine used for testing was running on an Intel Core 10th Gen i5-10210U Processor with 4 cores, 8 threads, clocking 1.60 GHz at base frequency and 4.20 GHz at max frequency. The machine had 16GB DDR3 RAM. The tests were run on Ubuntu 20.04 on WSL2 on Windows 10.

All the tests used the same Siege settings – there were no delays between requests, there were 200 concurrent users simulated at the same time, and the tests were repeated 200 times, meaning that each test ran 40000 times. The command used to invoke the tests looked like this:

*siege -r200 -c200 -b <URL>*

The following test cases were performed.

**Hello world test**

hello.php

```
1  <?php
2    echo 'Hello world';
3  ?>
4
```

hello.ck

```
1  <&
2    echo('Hello world');
3  &>
4
```

The first test ran a script that only returned the "Hello world" string back to the client. It tested the speed of parsing simple HTTP requests and constructing HTTP responses.

| | Elapsed time | Transaction rate | Throughput |
|---|---|---|---|
| Mish server | 4.97 secs | 8048.29 trans/sec | 0.08 MB/sec |
| Apache server | 3.59 secs | 11142.06 trans/sec | 0.12 MB/sec |

**Simple HTML page test**

The second test requested an HTML page from the server. The page used for testing was the Apache2 Ubuntu Default Page. The results are as follows:

| | Elapsed time | Transaction rate | Throughput |
|---|---|---|---|
| Mish server | 10.75 secs | 7441.86 trans/sec | 50.56 MB/sec |
| Apache server | 9.95 secs | 8040.20 trans/sec | 24.83 MB/sec |

One might notice an oddity here - even though the elapsed time and transaction rates are comparable, the Mish server doubled the throughput. It is due to caching mechanisms that the Apache server is using, some page resources were not transferred multiple times (such as images).

**Fibonacci sequence test**

The last test ran a script that generated and displayed first ten Fibonacci numbers recursively (see Fig. 18). This method is known for being a CPU-intensive task.

| | Elapsed time | Transaction rate | Throughput |
|---|---|---|---|
| Mish server | 49.88 secs | 801.92 trans/sec | 0.11 MB/sec |
| Apache server | 4.11 secs | 9732.36 trans/sec | 0.78 MB/sec |

The speed of parsing HTTP requests and serving HTTP responses is comparable between the two tested servers, with the Apache server performing a bit better.

fib.php

```
1  <?php
2    function fib($n) {
3      if ($n ⩽ 1) return $n;
4      return fib($n - 1) + fib($n - 2);
5    }
6
7    for ($i = 1; $i ⩽ 10; $i++) {
8      echo "fib(" . $i . ") = " . fib($i) . "<br>";
9    }
0  ?>
```

fib.ck

```
1  <&
2    func fib = function(int n) int {
3      if (n ⩽ 1) return n;
4      return fib(n - 1) + fib(n - 2);
5    };
6
7    int i = 1;
8    for (; i ⩽ 10; i += 1) {
9      echo("fib(" + i + ") = " + fib(i) + "<br>");
10   }
11 &>
```

*Fig. 18.Recursive Fibonacci function in PHP (top) and Ckript (bottom)*

It is noticeable that Ckript is a rather slow programming language and was a bottleneck in the Fibonacci test, it performed about ten times slower than PHP. This is because Ckript uses a tree-walk interpreter, while PHP has been a just-in-time compiled language since version 8.

## 4. Summary

The server at its current state is fully capable of powering small and medium-sized web projects. It's fast enough to handle the traffic of a semi-popular website, and the scripting language allows for developing most common back-end tasks. Ckript enables web developers to build complex page templates when combined with HTML code. Most other programming languages require external templating engines or front-end libraries to achieve that. Most popular front-end frameworks such as React, Vue, or Angular rely on client-side rendering, which is not great for a site's SEO. There are libraries (such as Nuxt or Next) that work on top of these frameworks to solve this problem. Ckript coupled with the web server makes server-side rendering possible without having to install any additional software. It is possible to define business logic directly in the presentation layer or make reusable components by creating template strings and then interpolating them later. These components can be exported to external files and then included in the presentation layer if needed.

There are a few ways that could improve the usability of the server, such as implementing user sessions which would streamline developing authentication and authorization systems. Right now, this can be achieved by sending tokens to the server with each request, but most web servers available today offer tools that handle cookies and sessions under the hood. Database drivers could be added to enable integration with databases such as MySQL, PostreSQL, or MongoDB and allow for easier persistent data storage. Although Ckript offers file I/O capabilities, these can prove to be hard to work with if there's structured data to be stored. The scripting language could be made faster by optimizing the interpreter or writing an ahead-of-time or a just-in-time compiler for it. Ckript is not a blazing fast language due to its tree-walk interpreter. Traversing an AST is rather slow since its nodes are all over the place in memory, which leads to cache misses. A way to speed it up would be writing a compiler targeting bytecode. Bytecode can be represented as an array that can be processed linearly, reducing the number of cache misses and so-called pointer chasing. This would also require writing an entirely new virtual machine, since the existing one can only work with ASTs but would speed things up significantly.

## 5. Additional resources

Language repository with documentation: https://github.com/Roller23/ckript-lang

HTTP server repository with language fork and documentation for the server only: https://github.com/Roller23/Mish

Online interpreter for language. It uses WebAssembly (C ++ compilation to WASM) to run the interpreter in the browser: https://ckript.netlify.app/ and the repository for the online interpreter: https://github.com/Roller23/ckript-online.

Ckript implementation in Javascript in the Node.js environment. This version is being developed in parallel with the C ++ version. It has some improvements such as access to network functions or a simplified type system for numbers: https://github.com/Roller23/ckript-js.

## Acknowledgments

The authors want to thank Dr Piotr Duch (ORCID ID 0000-0003-0656-1215) and Dr Tomasz Jaworski (ORCID ID 0000-0001-8600-3760) for their inspiration, support and fruitful advice.

This work was financed by the Lodz University of Technology, Faculty of Electrical, Electronic, Computer and Control Engineering as a part of statutory activity no. 501/2-24-1-2

## References

[1] Hamblin C. L.: Translation to and from Polish Notation. Comput. J. 5, 1962, 210–213. [http://doi.org/10.1093/COMJNL/5.3.210].
[2] Nystrom R.: Crafting Interpreters. Genever Benning, 2021.
[3] C++ Operator Precedence – cppreference.com, (n.d.). https://en.cppreference.com/w/cpp/language/operator_precedence (18.02.2022).
[4] Siege: HTTP/HTTPS stress tester – Linux man page, (n.d.). https://linux.die.net/man/1/siege (18.02.2022).
[5] The Shunting Yard Algorithm, (n.d.). http://mathcenter.oxford.emory.edu/site/cs171/shuntingYardAlgorithm/ (18.02.2022).

**Eng. Wiktor Kania**
e-mail: victorkaniaweb@gmail.com

Wiktor Kania is a graduate of Lodz University of Technology. His interests lie in modern web development, network programming, operating systems, and computer architecture.

http://orcid.org/0000-0002-0128-2762

**DSc. Ph.D. Eng. Radosław Wajman, prof. TUL**
e-mail: radoslaw.wajman@p.lodz.pl

Radosław Wajman is a professor at the Institute of Applied Computer Science at the Lodz University of Technology. In his work, he deals with the issues in the field of electrical capacitance tomography, fuzzy inference, software engineering, two-phase gas-liquid flow recognition, image reconstruction, and recognition.

http://orcid.org/0000-0002-6372-5960