The Design of an Experimental Programming Language and its Translator

The Nuua Programming Language

ÈRIK CAMPOBADAL FORÉS

Bachelor thesis submitted as partial fulfillment
of the requirements for the degree of
ICT Systems Engineer

Advisor: Sebastia Vila Marta
Department of Mining, Industrial and ICT Engineering
Technical University of Catalonia
June 2, 2019

Contents

List of Figures

1 Introduction

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."*
>
> — Martin Fowler, *Refactoring: Ruby Edition, p.36*

Programming languages are used every day by millions of engineers as part of their daily routine. A programming language is used to tell a computer what to do. When somebody wants a computer to do something, it needs to write a program using a programming language. Then, a compiler needs to translate it into machine code to be executed.

To design a programming language it's important to understand the theory behind a compiler and to learn about all the steps involved to make a computer understand and execute a program.

1.1 Objectives

The main objective of this thesis is to design an experimental programming language and implement an interpreter to execute any program written with it. The different challenges that are faced during the design and implementation process are also explained and solved in their respective chapters. The experimental language built in this thesis is called *Nuua*.

The objective can be partitioned into the following points.

- Learn all the steps involved in a common compiler implementation and reproduce them according to the project needs.
- Design the Nuua Programming Language. The grammar must be simple, elegant and yet it needs to follow the most common programming language's specifications to have a low learning curvature.
- Choose an efficient programming language to build the compiler and the interpreter with. Among other options, the languages that satisfy the previous statement are low-level programming languages like C [Bri88], C++ [Bja13], D [D L], Rust [Theb] or Go [Thea] among others.
- Define a robust system architecture to design the compiler and the interpreter. The system architecture needs to be scalable.

- Build a compiler and an interpreter for the Nuua programming language and a very simple standard library.

1.2 Preliminary overview

This section briefly introduces some of the concepts found in this thesis, introducing preliminary concepts of language grammar, compilers and interpreters. This preliminary overview won't deal with details and only explains the basics to understand the whole system without deep knowledge. Further chapters contain expanded information respective to some of the details mentioned here.

1.2.1 Language grammar

Context-free grammar is a notation used to specify the syntax of a programming language. Following the syntax definition explained in [Alf06, Section 2.2] a context-free grammar consists of four components:

1. A group of terminal symbols also known as tokens. In a programming language tokens may be literal symbols like '+', '*' or numbers and identifiers.
2. A group of non-terminals that can be reduced to terminals based on the production rules.
3. A group of production rules that consists of a non-terminal on the left side and a sequence of terminals and/or non-terminals on the right side.
4. A non-terminal start symbol.

This thesis will use the *extended Backus-Naur form* also known as *EBNF* to express the context-free grammar representation of Nuua. EBNF is often used in different ways due to the big amount of variants that exist. To EBNF syntax that this thesis is uses is shown in the Table 1.1. More information may be found in the EBNF grammar article [Fed17].

Symbol	Definition
`:`	Used to define a production rule.
space	Used to concatenate patterns (space separated).
`A|B`	Used to define a union of A and B.
`A+`	Used to define a one or more pattern of A.
`A*`	Used to define a zero or more pattern.
`A?`	Used to define an optional pattern.
`(A)`	Used to group a pattern.
`"T"` or `'T'`	Used to define a terminal symbol.
`@A`	Used to indicate anything except A.
`;`	Used to terminate a given production rule.

Table 1.1: Variation of EBNF syntax used by this thesis

As a simple example, to define a language that consists of a single integer, the following EBNF grammar could be used:

```
integer
     :  ("-" | "+")?  digit+
     ;
digit
     :  "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
     ;
```

This variation is often used by many parser generators since it introduces a more visible and versatile approach to write the language grammar.

1.2.2 Compilers

The job of a compiler is to take an input program written in a programming language and translate it into another as shown in Figure 1.1. The compiler term is often used to express a translation to a much different level of abstraction, that usually means that the input is written in a high-level language and further translated into another low-level language.

Figure 1.1: Compiler overview

Phases of a compiler

A compiler can also be decoupled into different parts. Each part does a very different job but they are all connected to each other. In a typical compiler architecture, we may find all the different phases described in Figure 1.2.

Those phases are often found to be different depending on the implementation of the language. However, it's important to note what they do, since they are often implemented in one way or another. More implementation details are explained in their respective chapters but in this section, a small introduction to each phase is needed to understand the Nuua's system.

- *Lexical analysis*: In this phase, the input source is transformed from a character string into a token list, this is also called tokenization. Lexemes found in the source program are translated into individual tokens using different patterns. For example, some tokens might include integers, symbols (+, -, *, etc.), identifiers or keywords ('if', 'while', etc.).
- *Syntax analysis*: In this phase, the implementation may vary among compilers, some of them work close to the lexical analysis since they can work together. However, its purpose is to perform operations given the token list to parse the input and create an Abstract Syntax Tree (AST). As seen in [Kei11, Section 5.2.1], an AST is a data structure that represents the input program. (AST). As seen in [Kei11, Section 5.2.1], an AST is a data structure that represents the input program. This stage determines if it's a valid program based on the language grammar and the specified rules. There are also scanner-less parsers that take the lexical analysis and the syntax analysis into a single step. It is harder to understand and debug compared to the modularity of splitting these two phases but it has some advantages like removing the token classification as mentioned in [Wik19].
- *Semantic analysis*: This phase analyzes the AST and creates a symbol table while analyzes the input source with things like type checking or variable declarations, if some operations can be performed (for example adding a number with a string). A symbol table is a structure used by further phases to see information attached to specific source code parts. For example, it can store information about a variable (if it's global, exported, etc.).
- *Intermediate code generator*: An Intermediate representation (IR) can be avoided but it's often used to have a platform independent optimizer. Usually the code

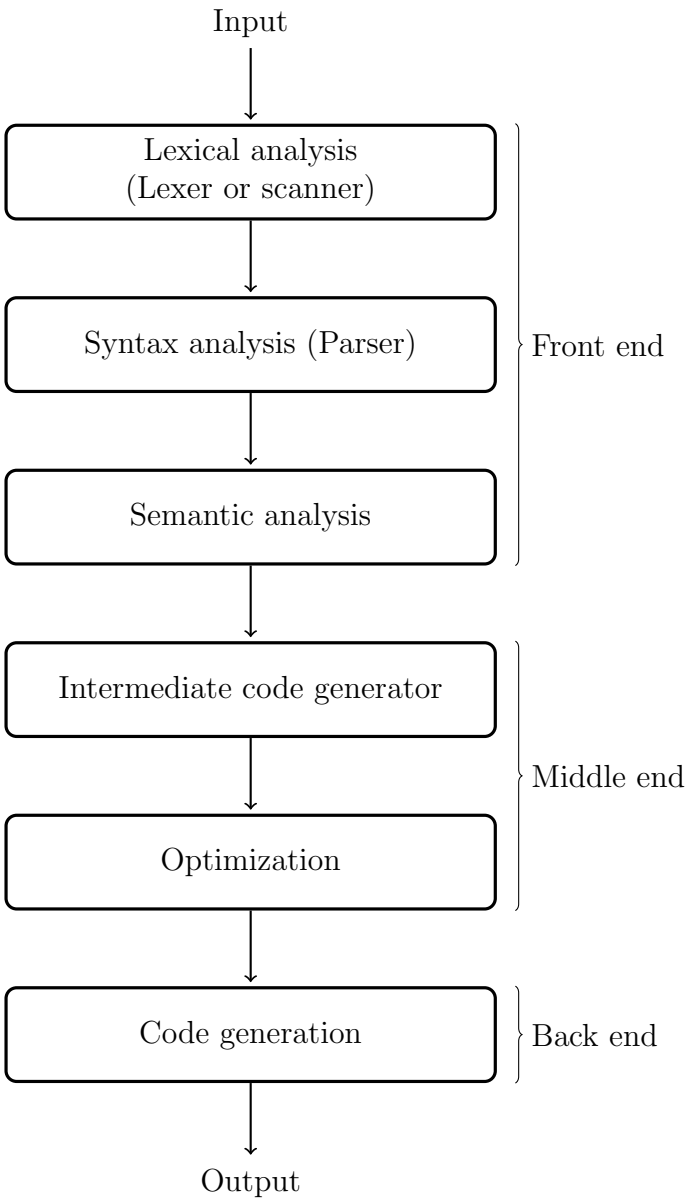

Figure 1.2: Common compiler phases

generation targets a specific architecture and would require different optimizers depending on each architecture. However, by having an IR it's possible to have a single optimizer. A much used IR is Three Address Code (TAC) that can be organized in quadruples or triples as seen by some examples in [Agg12].

- *Optimization*: This optimization is often performed on the IR and performs different tasks to allow a faster and smaller output. For example, it may remove dead code, perform loop optimizations, etc.
- *Code generation*: This is where the real translation takes place, it translates the IR into a different language output. For example machine code. This phase often has to deal with instruction scheduling or register allocation while they have to output a fully working program.

1.2.3 Interpreters

There are different ways to interpret a program. Among the most popular options we can find a bytecode interpreter and an AST interpreter.

- *Bytecode interpreter*: The program is first compiled to bytecode instructions and further interpreted. Bytecode interpreters are often implemented as virtual machines since most of the times bytecode instructions are very similar to real hardware instructions. The usual choices are stack or register machines. There have been many discussions on the advantages and the inconveniences of both of them, more information may be seen at [Yun05]. An example of a virtual machine is the Lua virtual machine [Rob05].
- *Abstract syntax tree interpreter*: This kind of interpreters just need the AST to work with, so no extra compilation to bytecode is needed and therefore, they are easier to implement. However, due it's nature, they are much slower to execute and debug due to the recursiveness of working with tree data structures.

1.2.4 Just-in-time compilers

Just-in-time compilers (often called JIT compilers) are an intermediate approach between a compiler that generates machine code and an interpreter. JIT compilers compile chunks of code at specific moments while the program run to speed up portions of the code that is being interpreted (for example functions that are called frequently). Essentially, the JIT compiler needs to decide when to compile a specific part of the code at runtime and adds a small overhead in exchange for a machine-language performance on specific parts of the program.

More information about JIT compilers and the tools that can be used can be found at [Tim10].

1.3 System architecture

To design the system architecture of Nuua, consideration of existing system architectures needs to be taken since existing architectures often work better than the custom-made ones and they often lead to greater project scalability. It's trivial to make this choice before starting the project since changing a system architecture after it's initial development phases becomes a very bad choice and may lead to a big ball of mud. Two choices in software development might be hierarchical or layered systems. Nuua's architecture is based on a *layered system* [Fel00].

A layered system has specific requirements regarding code communication. Specifically, a layered system consists of different layers arranged vertically. Those layers have a specific criterion that needs to be met. As a matter of fact, each layer can only use the layer below and gives an API for the layer above (if any) to use its functions. For example, the Figure 1.3 shows a simple 3-tier layered system. Layer 3 can only use Layer 2, and the output comes from Layer 2. Layer 3 cannot use Layer 1 nor expect any outputs from it. It's the Layer 2 responsibility to use the Layer 1 and process its output before it can give its own output.

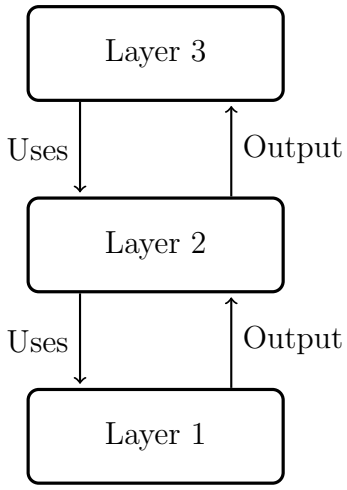

Figure 1.3: Layered system

This system is known to be robust, easy to test and with a high ease of development as mentioned in [Mar15]. It's very easy to understand and a widely used system. This system is also used for other complex software systems, such as operating systems or complex protocols like TCP/IP.

By using a layered system each layer gets completely isolated and works independently by just using the layer below, creating a way to scale-up or upgrade existing parts of the system without damaging the others. This introduces a very powerful *separation of concerns* among all the system layers since each layer has a specific role and only deals with the logic that pertains to it. However, a consistent API should, in fact, be

established from the ground up to avoid backward incompatible changes. If the API is maintained, the individual layers may be upgraded independently without the need for extra work.

Figure 1.4 shows the Nuua architecture. An independent module called Logger is found on the left side of the figure. This module is a logger used by all layers to output messages if needed (for example error reporting).

- *Logger*: Used by all layers to debug or log errors. In case of a fatal error, the logger outputs an error stack in a fancy way and terminates the application.
- *Application*: This layer is used to decode the command line arguments and fire up the compiler toolchain. In short, the purpose is to analyze the command line arguments and fire the application accordingly.
- *Virtual Machine*: The virtual machine is the interpreter that runs Nuua. It's a register-based virtual machine that acts as the Nuua runtime environment.
- *Code generator*: Is responsible for the translation of the AST to the virtual machine bytecode. This acts as the code generation part of a compiler architecture.
- *Semantic analyzer*: Does all the semantic analysis of the compiler and optimizes the AST for faster and smaller output.
- *Parser*: Acts as the syntax analyzer, it uses a list of tokens and generates a fully valid AST.
- *Lexer*: Scans the source code and translates the characters to tokens, creating a list of tokens representing the original source code.

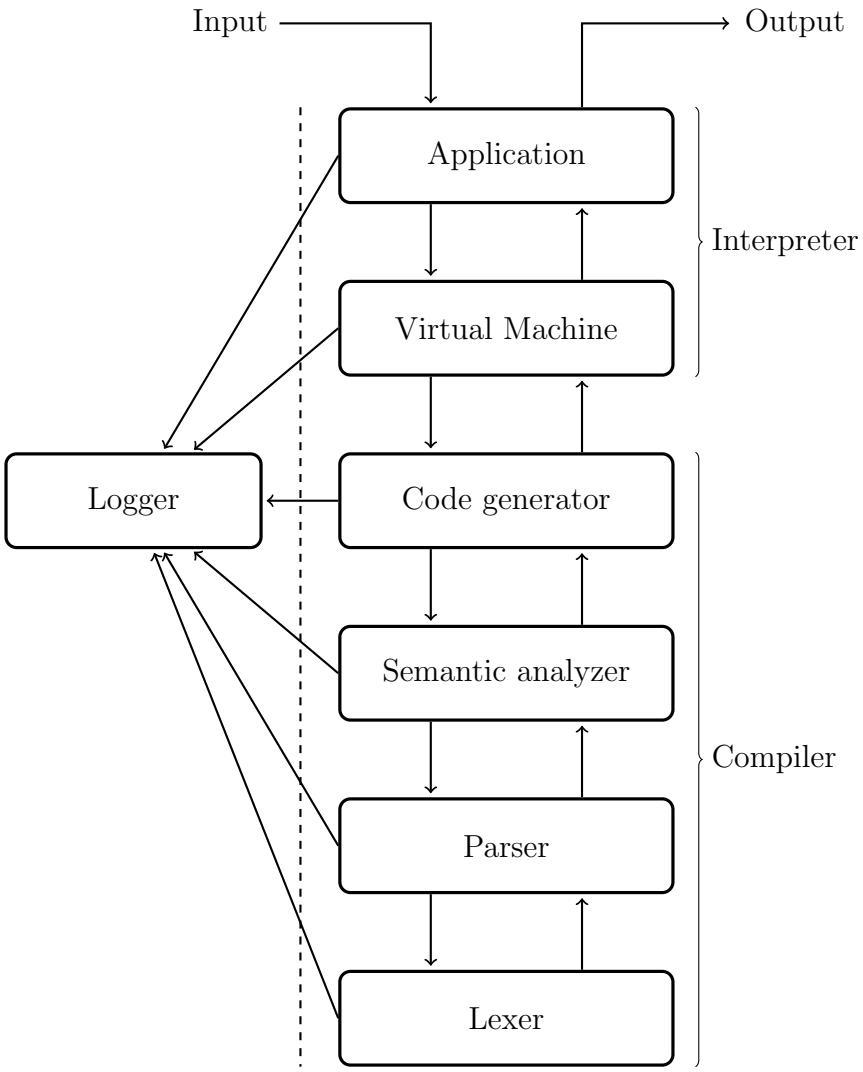

Figure 1.4: Nuua's architecture (Layered System)

2 The Nuua Language

> *"Progress comes from finding better ways to do things. Don't be afraid of innovation. Don't be afraid of ideas that are not your own."*
>
> — Douglas Crockford, *December 21, 2006*

This chapter defines the Nuua programming language. Nuua is a general purpose programming language with an imperative paradigm and a statically typed system. The Nuua compiler and virtual machine explained in this thesis are written in C++ with a zero-dependency policy. The interpreter is a register-based bytecode virtual machine.

Nuua's system architecture built in this thesis, as described in section 1.3, consists of a compiler that translates a program written in Nuua into bytecode instructions and of a virtual machine interpreter that executes those bytecode instructions.

2.1 Grammar

Nuua's grammar is inspired by other existing programming languages by taking advantage of some of the best features some of them offer. Inspiration comes especially from Python [Pyt], Rust [Theb] and Go [Thea].

The precedence relationship between expressions is heavily inspired by C [Bri88], D [D L], Rust [Theb], and Dart [Goo]. The official documentation for those languages exposes similar tables for operator precedences and Nuua has taken akin levels of precedence as those.

Nuua does not make use of the `";"` to separate statements, instead, it uses the same separator as Go. Statements can be separated by a `"\n"` but it does not make use of the `"\t"` to indicate statements inside blocks, and uses the typical block separator `"{" ... "}"`.

2.1.1 Lexical Grammar

Lexical grammar is used by the lowest layer of the Nuua system to scan the source language and identify different terminal symbols. The main difference with the syntax

grammar, as exposed in [Bob18, Appendix I], is that the syntax grammar is a context-free grammar and lexical grammar is a regular grammar.

Nuua's lexical rules are as follows:

```
DIGIT
    : "0" | ... | "9"
    ;
```

Digits are any character from '0' to '9'.

```
ALPHA
    : "a" | ... | "z"
    | "A" | ... | "Z"
    | "_"
    ;
```

Alpha are characters that are part of the English alphabet in lower or upper case. It also includes '_' as a special character.

```
ALPHANUM
    : DIGIT
    | ALPHA
    ;
```

Alphanum are characters that are either part of the alphabet or are digits.

```
INTEGER_EXPR
    : DIGIT+
    ;
```

Integers are a single digit or more found sequentially without spaces. The integer sign is not represented here.

```
FLOAT_EXPR
    : DIGIT+ "." DIGIT+
    ;
```

Floats are like integers but require a dot followed by a digit or more, creating a decimal number.

```
BOOL_EXPR
    : "true"
    | "false"
    ;
```

Bools are either 'true' or 'false', that are keywords.

```
STRING_EXPR
    : '"' @'"' '"'
    ;
```

Strings represent a character string with the possibility to escape '"' by using a '\' as a prefix, more on that in the upcoming sections.

```
IDENTIFIER
    : ALPHA ALPHANUM*
    ;
```

Identifiers are an alpha character followed by an optional one or more alphanumeric character.

2.1.2 Syntax Grammar

The syntax grammar is used by the parser to build the abstract syntax tree that determines the program's execution flow. These rules include all top-level declarations, statements, and expressions.

Program and Top Level Declarations

```
program
    : top_level_declaration*
    ;
```

A Nuua program is a list of top-level declarations.

```
top_level_declaration
    : use_declaration "\n"
    | fun_declaration "\n"
    | class_declaration "\n"
```

```
    | export_declaration "\n"
    ;
```

A top-level declaration can only be one of the specified rules. Top-level declarations are a special type of declaration that can only be declared on the module and not inside other blocks.

```
use_declaration
    : "use" STRING_EXPR
    | "use" IDENTIFIER ("," IDENTIFIER)* "from" STRING_EXPR
    ;
```

A use declaration is used to import other top-level declarations from other modules. By using the first rule, Nuua imports all the exported targets of the module pointed by STRING_EXPR. Otherwise, Nuua imports the specified targets from the modules.

```
fun_declaration
    : "fun" IDENTIFIER "(" parameters? ")" (":" type)? fun_body;
    ;
parameters
    : variable_declaration ("," variable_declaration)*
    ;
fun_body
    : "->" expression "\n"
    | "=>" statement
    | "{" "\n" statement* "}" "\n"
    ;
```

A function is defined using the keyword `fun` followed by the function name, and a list of optional parameters enclosed in parentheses. The function type is specified after the parameter list by using `:`. The return type might not be present if the function has no return value. The function body is specified in three different ways depending on the type of the function body that is expected.

```
class_declaration
    : "class" "{" class_statement* "}"
    ;
class_statement
    : variable_declaration
    | fun_declaration
    ;
```

A class uses a keyword `class` and expects zero or more class statements.

```
export_declaration
    : "export" top_level_declaration
    ;
```

An export declaration marks the following top-level declaration as exported, making it available for other modules to import it using the use declaration.

```
statement
    : variable_declaration "\n"
    | if_statement "\n"
    | while_statement "\n"
    | for_statement "\n"
    | return_statement "\n"
    | print_statement "\n"
    | expression_statement "\n"
    ;
```

Statements are used to change the program's flow or indicate simple actions like declaring a variable.

```
variable_declaration
    : IDENTIFIER ":" type
    | IDENTIFIER ":" type "=" expression
    | IDENTIFIER ":" "=" expression
    ;
```

A variable declaration may have a type assigned with it, or the declaration type will be inferred from the initializer.

```
if_statement
    : "if" expression if_body el_if* else?
    ;
if_body
    : "=>" statement "\n"
    | "{" "\n" statement* "}"
    ;
el_if
    : "elif" expression if_body
```

```
    ;
else
    : "else" expression if_body
    ;
```

An `if` statement is declared with the `if` keyword followed by the expression of its condition. The `if` body may be declared in two different ways depending on the if body. The `elif` word may be used as a shorthand to an `else` followed by an "if" inside. An optional `else` condition may be added at the end of the `if`.

```
while_statement
    : "while" expression while_body
    ;
while_body
    : "=>" statement "\n"
    | "{" "\n" statement* "}"
    ;
```

A `while` statement uses the `while` keyword followed by the expression of the condition and the `while` body, that can be specified in two different ways depending on the body contents.

```
for_statement
    : "for" IDENTIFIER ("," IDENTIFIER)? "in" expression for_body
    ;
for_body
    : "=>" statement
    | "{" "\n" statement* "}"
    ;
```

A `for` statement acts is a way to interate a nuua iterator Table 2.3. It gets declared by using the `for` keyword followed by an identifier that will be used to store the element in the iterator. An optional second identifier may be given in case the loop index needs to be stored as well. Those values change in every iteration. After the identifiers, the `in` keyword is expected, followed by the iterator expression and the `for` body, that can be declared in two different ways depending on its content.

```
return_statement
    : "return" expression?
    ;
```

A `return` statement uses the `return` keyword and an optional expression to return.

```
print_statement
    : "print" expression?
    ;
```

The `print` expression is a statement that outputs an expression to the screen. This is a temporary statement used while Nuua is not able to properly read and write to files (stdout / stdin) in this specific case.

```
expression_statement
    : expression
    ;
```

An expression statement may be used as an expression whose value is not used or no value is returned from it.

Expressions

Expressions are grouped in different production rules depending on their precedence, this is done due to the parsing strategy used and helps to visualize the precedence by reading the grammar.

```
expression
    : assignment
    ;
```

An expression is reduced to an assignment.

```
assignment
    : range ("=" range)*
    ;
```

An assignment expression is written by an expression on the left-hand side and on the right-hand side with a `=` in the middle.

```
range
    : logical_or (("." | "...") logical_or)*
    ;
```

A range expression is written by an expression on the left-hand side and on the right-hand side with a `..` or `...` in the middle. Depending if the range is exclusive or inclusive.

```
logical_or
    : logical_and ("or" logical_and)*
    ;
```

A logical `or` is a binary operation with the keyword `or` in the middle.

```
logical_and
    : equality ("and" equality)*
    ;
```

A logical `and` is a binary operation with the keyword `and` in the middle.

```
equality
    : comparison (("!=" | "==") comparison)*
    ;
```

An equality comparison is a binary operation with a `!=` or `==` in the middle depending if the check needs to be negated or not.

```
comparison
    : addition (">" | ">=" | "<" | "<=") addition)*
    ;
```

A comparison is similar to equality but checks the values to determine if the left-hand side is greater, greater than, lower and lower than the right-hand side.

```
addition
    : multiplication (("-" | "+") multiplication)*
    ;
```

An addition is used to perform an addition or subtraction of two values.

```
multiplication
    : cast (("/" | "*") cast)*
    ;
```

Multiplication is used to perform multiplication or division of two values.

```
cast
    : unary_prefix ("as" type)*
    ;
```

A cast performs a type cast of the values on the left-hand side to the value of the right-hand side by using the `as` keyword.

```
unary_prefix
    : ("!" | "+" | "-") unary_prefix
    | unary_postfix
    ;
```

The unary prefixes are used to change a value state by prefixing the operation.

```
unary_postfix
    : primary unary_p*
    ;
unary_p
    : "[" expression "]"
    | slice
    | "(" arguments? ")"
    | "." IDENTIFIER;
    ;
slice
    : "[" expression? ":" expression? (":" expression?)? "]"
    ;
arguments
    : expression ("," expression)*
    ;
```

The unary postfixes are used to either access a value or mutate it's content, to slice it's contents, to call a value or to access a value property.

```
primary
    : BOOL_EXPR
    | INTEGER_EXPR
    | FLOAT_EXPR
    | STRING_EXPR
```

```
    | IDENTIFIER
    | LIST_EXPR
    | DICTIONARY_EXPR
    | OBJECT_EXPR
    | "(" expression ")"
    ;
```

Primary expressions have the highest precedence and are mostly native types, with the exception of the expression group.

```
OBJECT_EXPR
    : IDENTIFIER "{" object_args? "}"
    ;
object_args
    : IDENTIFIER ":" expression ("," IDENTIFIER ":" expression)*
    ;
```

An object is defined by an identifier containing the class name, followed by optional arguments surrounded by { and } to initialize the class properties.

```
LIST_EXPR
    : "[" expression ("," expression)* "]"
    ;
```

Lists can't be empty, so at least one expression must be provided.

```
DICTIONARY_EXPR
    : "{" IDENTIFIER ":" expression ("," IDENTIFIER ":" expression)* "}"
    ;
```

Dictionaries, like lists, can't be empty, so at least one expression must be provided.

2.1.3 Operator Precedence

Given that expressions are grouped by their precedence, the operator precedence table of Nuua is as follows:

Level	Operators	Associativity
1	`A[B]`, `A[B:C:D]`, `A(B, C, D, ...)`, `A.B`	Left-to-right
2	`!A`, `+A`, `-A`	Right-to-left
3	`A as B`	Left-to-right
4	`A/B`, `A*B`	Left-to-right
5	`+A`, `-A`	Left-to-right
6	`A>B`, `A>=B`, `A<B`, `A<=B`	Left-to-right
7	`A!=B`, `A==B`	Left-to-right
8	`A and B`	Left-to-right
9	`A or B`	Left-to-right
10	`A..B`, `A...B`	Left-to-right
11	`A=B`	Right-to-left

Table 2.1: Nuua operator precedence from highest to lowest with the associativity

2.1.4 Keywords and Reserved Words

Keywords are a special subset of identifiers that have a special meaning in a Nuua program. A reserved word is an identifier that can't be used as such, and in Nuua, no keywords can be used as identifiers therefore making all keywords reserved words at the same time. All keywords can already be identified by looking at the grammar rules, the following list shows all of the keywords in Nuua.

(i)	`true`	(ii)	`false`	(iii)	`as`	(iv)	`or`
(v)	`and`	(vi)	`if`	(vii)	`else`	(viii)	`for`
(ix)	`in`	(x)	`while`	(xi)	`return`	(xii)	`print`
(xiii)	`class`	(xiv)	`fun`	(xv)	`use`	(xvi)	`from`
(xvii)	`elif`	(xviii)	`export`				

More information about the `print` keyword and why it does exist can be found in subsection 2.5.10.

2.2 Scopes

Scopes refer to the visible area of the variables, in other words, it determines where the association between a variable name and its value (known as name binding) is valid. This area is known as a scope block.

Nuua have two levels of scope blocks:

1. *Module scope*: Any top-level declaration found in Nuua is bound to the module scope. Any other module won't see that top-level declaration unless it is exported and the module is trying to use it.
2. *Block scope*: Any declaration found in any statement that contains blocks (statements found in Table 2.2) is only valid inside of the block.

Statement	**Blocks**
Function declaration	Body
Class declaration	Body
If statement	Then and Else
While statement	Body
For statement	Body

Table 2.2: Nuua statements with scope blocks

2.3 Entry point

A Nuua program requires an entry point to start executing the instructions. The entry point in a Nuua program is a function that must be called `main`. This function must exists in the initial module. If the function does not exist an error is thrown prior to execution. The `main` function needs at least one argument of type `[string]` that does contain the command line arguments.

The Nuua virtual machine does automatically call the `main` function upon starting executing the bytecode with the command line arguments of the call.

An example `main` may be as follows:

```
fun main(argv: [string]) {
    // ...
}
```

2.4 Data types

This section defines all the Nuua data types that are supported. Each value in Nuua has a type associated with it, meaning that each value must belong to a certain data type. A value can't belong to multiple data types at once but can be cast to others if a change is required.

2.4.1 Integers

Integers are named as `int` and they are a subset of $\mathbb{Z}$. It includes 0 and the integers are stored using 64 bits using two's complement. Meaning its range for a given integer x is:

$$-2^{64-1} < x < 2^{64-1} - 1$$

2.4.2 Floats

Floats are named as `float` and they are C++ `double` precision points that use a total of 64 bits. 52 fraction bits, 11 bits of exponent and 1 sign bit.

2.4.3 Booleans

Booleans are named as `bool` and they are simple booleans, they can be either `true` or `false` and they are stored in a C++ `bool` type, usually using 8 bits to store it.

Examples

```
true
false
```

2.4.4 Strings

Strings are named as `string` and they are used to manipulate arrays of chars. It's implementation uses a C++ `std::string` and it's planned to support wider characters as mentioned in chapter 3. It can store any text that's surrounded by `'"'`.

2.4.5 Lists

Lists are named as `[type]` and they are used to manipulate a list of other values. They can only have a single type as the inner list items, so all the list items need to be of the same type.

2.4.6 Dictionaries

Dictionaries are named as {`type`} and they are used to store values of the same type. However, unlike lists, they use a string-based mapping, allowing each value to be bound to a specific string key, instead of an integer index as the key. Dictionaries, like lists, can only store 1 type of values. So each key can only store the same type.

2.4.7 Functions

The only way to define a function is by using the "fun" keyword as noted in section 2.1.2. However, functions in Nuua act as first-class values, meaning that values can contain a function, allowing the function to change without actually changing its type. The function type needs to be consistent. So even if the function changes, it will always accept the same arguments and it will return the same data type. Function types are named as `(T1, T2, ..., TN -> TR)`. where `T1` to `TN` are the types of the function parameters. If the function has a return type, say `TR`, it needs to be specified with a simple arrow pointing at it the end of the function type.

To see how functions are declared head to subsection 2.5.2

Examples

Function without parameters nor return type.

```
()
```

Function with 2 parameters of type int and a float return type.

```
(int, int -> float)
```

Function without parameters and a return type of a list of strings.

```
(-> [string])
```

Function with two parameters of type int and bool and no return type.

```
(int, bool)
```

2.4.8 Objects

Objects are named according to the class they represent. If a class is named `Person` the type name is `Person`.

2.5 Statements

This section explains and gives examples to all statements found in Nuua.

Some of the statements may have already been mentioned briefly in section 2.1.2.

2.5.1 Use Declaration

The use declaration is used when a module needs to use a top-level declaration that is found in another module. The target module is the string given in the use declaration. A path system is used to search for the file, first trying to find it relatively from the current module path, ending at the standard library folder that comes with Nuua.

The use declaration comes in two different shapes. By using the use declaration with a single string it imports all the top level declarations that are exported in the target module. Instead, by determining the "use" identifiers, you may import only selected top-level declarations.

Caveats

- The target module path given in the use declaration can use a relative or absolute path.
- If the target module path does not have the `".nu"` extension, it will be added automatically.
- If the target module is not found in any path, an error is thrown before any execution starts.
- If a target top-level declaration is not found in the target module an error is thrown before any execution starts.
- If the top level declaration is not exported another error will be thrown prior to execution.

Examples

Import all exported targets in a relative file path named "test.nu"

```
use "test"
```

Import `a` and `b` from a relative file path named "test.nu"

```
use a, b from "./test.nu"
```

Import `a` from an absolute file path in "C:/Nuua/test.nu"

```
use a from "C:/Nuua/test.nu"
```

2.5.2 Function Declaration

A function declaration creates a function value and a data type given the function parameters and return value. Once the function value is created, it's then added in a new variable with the function name. That variable can be modified since functions are first-class values in Nuua. The variations in a function body exist to minimize the code length in certain situations (When a function is a single return expression, a single statement or a block of statements).

Caveats

- No function overloading is allowed.
- Function parameters do not allow default values.
- If the function returns a value, you are expected to, at least, write a single return statement in the top level of the function block.
- Functions without return type can't be used as formal expressions since they contain no values.

Examples

Function without parameters and no return type.

```
fun a() {
    print "Hello, World"
}
```

Function without parameters and return type.

```
fun b(): string {
    return "Hello, World"
}
```

Function with parameters and no return type.

```
fun c(x: string) {
    print "Hello, " + x
}
```

Function with parameters and return type.

```
fun d(x: string): string {
    return "Hello, " + x
}
```

Single statement function.

```
fun e(x: string): string => return "Hello, " + x
```

Single expression function.

```
fun f(x: string): string -> "Hello, " + x
```

2.5.3 Class Declaration

A class declaration creates a data type of the given class structure. This type can then be used as a regular type to specify values of the given class. To create an object of a given class you can use the Object expression [**sec:object˙expression**]. Classes act as structs with the fact that they can also contain methods bound to them. Class methods have a variable called `"self"` as a self-reference to the object to mutate its state.

Caveats

- Class properties can't have default values. Values are defined when creating the object using an object expression.
- Self-references to the same type are allowed.
- There is no class inheritance.

Examples

Simple class to represent a person.

```
class Person {
    name: string
    age: int
    fun show() {
        print self.name + ", " + self.age as int
    }
}
```

2.5.4 Export Declaration

The export declaration is used when a module wants to make a top-level declaration available to use for other modules. Marking a top-level declaration as exported allows other modules to import and use it.

Caveats

- You can't export another export.

Examples

Export a function

```
export fun add(a: int, b: int): int {
    return a + b
}
```

Export a class.

```
export class Person {
    name: string
    age: int
}
```

2.5.5 Variable Declaration

Variable declarations are scoped to the block where they are declared as mentioned in section 2.2. They can be used from the block they have been declared and on the lexical blocks that may exist inside of it. A variable with the same name can't be declared in the same lexical block but multiple lexical blocks may have the same variable name. When getting the value of a variable, the lookup starts from the current block and goes back to previous blocks.

Caveats

- Even if multiple blocks have the same variable name, they all point to different values.
- If a variable is declared with an initializer, the type can be inferred by leaving the type empty.
- If a variable is declared without an initializer, the value is default initialized to a zero-state.
- If a variable is declared in a block that already contains a variable with the same name, an error is thrown before execution.

Examples

Simple variable declaration (defaults to `int`'s zero state, in this case 0).

```
a: int
```

Variable declaration with an initializer.

```
b: int = 10
```

Variable declaration with an inferred `int` type.

```
c := 10
```

2.5.6 If Statement

An "if" statement is used to execute a block of code when a certain expression (known as condition) evaluates to `true` known as the `then` block. The if statement can also execute another block if the condition is `false` known as the `else` block. The `else` block can be defined using the `"else"` keyword. An if statement has a shorthand for defining another if inside the `else` block, making nested if statements easier to write. This syntax uses the `"elif"` keyword and acts the same way as defining an `else` block with another if statement inside. Additionally, the if statement body may be defined in different ways depending on the body type. If the body consists of a single statement, a shorthand can be used to minimize the lines of code.

Caveats

- The condition must always be a boolean. Explicit casting is needed.

Examples

Simple if statement.

```
if condition {
    print "Condition is true"
}
```

If statement with an else block.

```
if condition {
    print "Condition is true"
} else {
    print "Condition is false"
}
```

If statement with multiple nested conditions.

```
if number == 0 {
    print "The number is 0"
} elif number == 1 {
    print "The number is 1"
} else {
    print "The number is not 0 nor 1"
}
```

If statement with the shorthand body.

```
if number == 0 => print "The number is 0"
elif number == 1 => print "The number is 1"
else => print "The number is not 0 nor 1"
```

2.5.7 While Statement

A while statement is used to repeat a block of code while a certain expression (known as condition) evaluates to `true`. The condition is evaluated every time the loop is about to begin. If the while block is executed the program counter jumps back to the condition to evaluate it again. When the condition is no longer true, the program counter skips the block and continues execution. The while statement also has a shorthand to define its body when it only consists of a single statement.

Caveats

- The condition must be a value of `bool` type. Explicit casting is needed.

Examples

Simple while statement.

```
while condition {
    print "Condition is true"
}
```

Using the shorthand for single statements.

```
while condition => print "Condition is true"
```

Real world while example

```
a: int = 0
while a < 10 => print a
```

2.5.8 For Statement

A for statement is very similar to a while statement but instead of working with a condition it works with an iterator. An iterator is a data type that supports indexation and therefore, can be iterated. Nuua iterators are:

Data type	Value Type	Index Type
`string`	`string` (a single character)	`int`
`[T]`	`T`	`int`
`{T}`	`T`	`string`

Table 2.3: Nuua iterators

Indexation is done with the *Access* expression. The for loop defines up to two variables to its block. One containing the current value of the indexed item and another optional variable containing the current index being used.

Caveats

- The value and the index are variables that are automatically declared with their respective types in the "for" block.
- The value and the index types are automatically inferred according to Table 2.3.

Examples

Simple for statement.

```
for char in "string" {
    print char
}
```

For statement with the index.

```
for letter, index in ["A", "B", "C"] {
    print index as string + ": " + letter
}
```

For statement with the shorthand.

```
for num in 0..10 => print num
```

2.5.9 Return Statement

A return statement is used inside the function to determine its execution should end, and optionally return a value as the result. Return statements are mandatory in functions that have a return type.

Caveats

- If the function has a return type, at least one return at the top level of the function block is required. Otherwise, an error is thrown prior to execution.
- Return expression type must match the function's return type.

Examples

A simple return statement.

```
fun a() {
    return
    print "Never executed"
}
```

A return statement returning a value.

```
fun b(): int {
    return 10
}
```

2.5.10 Print Statement

The print statement is used to write a register to the `stdout` file. This statement will be finally deleted alongside the keyword when a propper I/O is added into the language as mentioned in chapter 3.

Caveats

- Any data type can be printed with this statement. Even functions and objects.

Examples

A simple print statement.

```
print "Hello, World"
```

A print of a function

```
fun a(): int {
    print a
    return 10
}
```

2.6 Expressions

Expressions can always be reduced up to a value of a single data type. This section explains all the expressions that can be found in Nuua.

Some of the expressions may have already been mentioned briefly in section 2.1.2.

2.6.1 Integer Expression

The integer expressions can be written as an integer number directly in the source code. Integer expressions return a value with the `int` data type.

Caveats

- There are no prefix/postfix indicators to change the integer bit size or base (Like LL, 0x, etc.).

Examples

```
0
25
81237
-6378
-1
```

2.6.2 Float Expression

The float expressions can be written as any floating point number, using a `"."` as the decimal delimiter, directly in the source code. Float expressions return a value with the `float` data type.

Caveats

- An integer followed by a `"."` without any other number on the right-hand side, it's considered an error. An explicit number must be written in the right-hand side to create a float expression, even to indicate `.0`.

Examples

```
0.0
25.5
81237.11111
-6378.673
-1.9
```

2.6.3 Boolean Expression

The boolean expressions can be written as either `true` or `false` directly in the source code. Boolean expressions return a value with the `bool` data type.

Examples

```
true
false
```

2.6.4 String Expression

The string expressions can be written as any text enclosed between `'"'` directly in the source code. String expressions return a value with the `string` data type.

Caveats

- As for June 2, 2019, the strings use a bare-bones C++ `std::string` to represent the string, that means that the string is a list of single-byte characters (characters in the ASCII character table). A plan to support wider characters is mentioned in chapter 3.

Examples

```
"A string is represented like this"
```

2.6.5 List Expression

The list expressions can be written as list of expressions separated by comma enclosed between `"["` and `"]"` directly in the source code. A list inner type, say `T`, is determined by the type of the first expression of the list. List expressions return a value with the `[T]` data type.

Caveats

- List expression can't be empty due to an unknown type. Even when assigning them to a variable with a defined type. If there's the need for an empty list of a given type, declare a variable with the type and don't initialize it.
- Lists can only have a single type stored on it, therefore, if a list expression have more than one expression on it, the types must match. If a type does not match the first type of the list, an error is thrown prior to execution.

Examples

A list expression that return a value of type `[string]`.

```
["this", "is", "a", "valid", "list", "of", "strings"]
```

A list expression that return a value of type `[int]`.

```
[1]
```

2.6.6 Dictionary Expression

The dictionary expressions can be written as a list of comma-separated pairs of `key: expression` The key is an identifier representing the dictionary key willing to be used. directly in the source code. A dictionary inner type, say `T`, is determined by the type of the first expression of the list. Dictionary expressions return a value with the {`T`} data type.

Caveats

- Dictionary expression can't be empty due to an unknown type. Even when assigning them to a variable with a defined type. If there's the need for an empty dictionary of a given type, declare a variable with the type and don't initialize it.
- Dictionaries can only have a single type stored on it, therefore, if a dictionary expression have more than one expression on it, the types must match. If a type does not match the first type of the dictionary, an error is thrown prior to execution.

Examples

A dictionary expression that return a value of type {`string`}.

```
{name: "Erik", occupation: "Student", color: "#ff0000"}
```

A dictionary expression that return a value of type {`int`}.

```
{left: 10, right: 20, sum: 30}
```

2.6.7 Object Expression

The object expression is used to initialize an object of a given class. The object expression is used by writing the identifier of the class followed by a `!` and list of comma-separated pairs of `key: expression` (where the key is an identifier) enclosed between { and }. The keys in the argument list are the class properties willing to initialize and the expression is the value that the property is going to be assigned to.

Caveats

- The keys found in the arguments must exist in the class properties. If one of the keys does not correspond to an existing class property an error is thrown prior to execution.
- The expressions of the keys in the argument list must match the class property type they want to initialize. If there's a type mismatch, an error is thrown prior to execution.

Examples

An example class to provide the examples.

```
class Person {
    name: string
    born_at: int
}
```

An object of class Person without arguments.

```
Person!{}
```

An object of class Person with arguments.

```
Person!{name: "Erik", born_at: 1997}
```

2.6.8 Group Expression

The group expression is used to give certain operations priority over the default operator precedence. The group expression is an expression enclosed between `(` and `)`.

Examples

```
(1 + 2) * 3 // 9
(1 + 4 - 3) * (2 * (2 + 2)) // 16
```

2.6.9 Access Expression

The access expression is used to access an inner value of another expressions. In short, the access expression can be used in any Nuua iterator and the returned value is the inner value found on its index with the respective type as shown in Table 2.3.

Caveats

- Only nuua iterators can be accessed.

Examples

A string access.

```
"Hello"[1] // e
```

A list access.

```
["Hello", "World"][1] // "World"
```

A dictionary access.

```
{key1: "Hello", key2: "World"}["key1"] // "Hello"
```

2.6.10 Slice Expression

Slices acts the same way as python slices and this explanation can be found in [Prz15]. Basically it's a way to get a range of inner values in a Nuua iterator (Only those that can be index with an `int` type). The supported parameters are:

Parameter	Explanation
`start`	Starting index of the slice. Defaults to 0.
`end`	The last index of the slice or the number of items to get. Defaults to the length of the iterator
`step`	Optional. Extended slice syntax. Step value of the slice. Defaults to 1.

Table 2.4: Slice parameters

Caveats

- Only nuua iterators whose inner value can be accessed using an `int` type can be sliced.

Examples

```
"Hello"[1:3] // el
"Hello"[1:] // ello
"Hello"[:3] // Hel
"Hello"[::2] // Hlo
"Hello"[::-1] // olleH
```

2.6.11 Call Expression

The call expression is used to call a value. The value needs to be callable and Therefore the value must be a function. When a call expression is used, its return value is the value returned from the function. The call accepts the arguments that will be passed to the function as the function parameters. The call expression is the caller and the target function is the callee.

Caveats

- If the callee has no return value, then the caller is banned from beeing treated as an expression, only beeing able to be used where its value is not used.

Examples

An example function acting as a callee.

```
fun test(a: int): int -> a + 1
```

An example call.

```
test(10) // 11
```

2.6.12 Property Expression

The property expression is used to access a specific property in an object, meaning that the target expression can only be an object. The property name is the identifier used after the dot.

Caveats

- If the object class has no property named as the identifier, an error is thrown prior to execution.
- If the object is not initialized, a runtime segmentation fault error is thrown.

Examples

An example class to work with.

```
class Person {
    name: string
}
```

An example property expression.

```
Person!{name: "Erik"}.name // Erik
```

2.7 Comments

Comments in nuua can be written by using a double \\ followed by the comment text. The comment text lasts till a \n character is found. Therefore, multiline comments can be done by manually writting the double back slash on each differnt line.

Comment blocks are not part of the language grammar and therefore they are totally discarded from the AST. When the lexer finds the double back slash, it proceed to discard the whole line.

```
// Some comment here
fun test() {
    print "Hello"
    // Some other comment here
    print "Hello again"
}
```

3 Forthcoming Features

> *"Manufacturing is more than just putting parts together. It's coming up with ideas, testing principles and perfecting the engineering, as well as final assembly."*
>
> — James Dyson

Although Nuua is currently in a very advanced phase, a lot of features are still missing, features that are present in most mature programming languages. This chapter defines a set of features that are planned to be implemented into the language and the compiler.

- A C foreign function interface (FFI) to call C code using Nuua. This should allow the use of libc and other existing C code.
- A propper I/O interface to read and write to files, including stdin, stdout, stderr and removing the print statement.
- Function overloading.
- Extend and create a proper useful standard library.
- A website (nuua.io) to announce, showcase and write all the documentation.
- A centralized package/module manager to automate the process of using external modules at a certain version.
- Use a wider string representation for the compiler. Change from `std::string` to something wider like `std::u16string` or `std::wstring`.
- Support more binary operators such as `%, +=, -=, /= or *=`.

4 Bibliography

[Agg12] Aggie Johnson. "Three Address Code Examples". English. In: (2012). URL: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/240%20TAC%20Examples.pdf.

[Alf06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. English. 2nd Edition. Greg Tobin, 2006. ISBN: 978-0321486813.

[And13] Anders Schlichtkrull, Rasmus T. Tjalk-Bøggild. *Compiling Dynamic Languages*. English. Technical University of Denmark, 2013. URL: http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/6620/pdf/imm6620.pdf.

[And15] Andrea Bergia. "Stack Based Virtual Machines". English. In: (2015). URL: https://andreabergia.com/stack-based-virtual-machines/.

[Bar14] Bartosz Sypytkowski. "Simple virtual machine". English. In: (2014). URL: https://bartoszsypytkowski.com/simple-virtual-machine/.

[Bja13] Bjarne Stroustrup. *The C++ Programming Language*. English. 4th Edition. Addison-Wesley Professional, 2013. ISBN: 978-0321563842.

[Bob11] Bob Nystrom. "Pratt Parsers: Expression Parsing Made Easy". English. In: (2011). URL: https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/.

[Bob18] Bob Nystrom. *Crafting Interpreters*. English. 2018. URL: https://craftinginterpreters.com/.

[Bri88] Brian W. Kernighan; Dennis M. Ritchie. *C Programming Language, 2nd Edition*. English. 2nd Edition. Prentice Hall, 1988. ISBN: 978-0131103627.

[D L] D Language Foundation. *D Programming Language Specification*. English. [Online; accessed 1-June-2019]. URL: https://dlang.org/spec/spec.

[Edw09] Edward Barrett. *A JIT Compiler using LLVM*. English. Bournemouth University, 2009. URL: http://llvm.org/pubs/2009-05-21-Thesis-Barrett-3c.pdf.

[Fed17] Federico Tomassetti. "EBNF: How to Describe the Grammar of a Language ". English. In: (2017). URL: https://tomassetti.me/ebnf/.

[Fel00] Felix Bachmann, Len Bass, Jeromy Carriere, Paul C. Clements, David Garlan, James Ivers, Robert Nord, Reed Little. *Software Architecture Documentation in Practice: Documenting Architectural Layers*. English. Software Engineering Institute, 2000. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5019.

[Goo] Google. "Dart Programming Language Specification". English. In: (). [Online; accessed 1-June-2019]. URL: https://dart.dev/guides/language/spec.

[Kei] Keith Schwarz. *Three-Address Code IR*. English. URL: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/13/Slides13.pdf.

[Kei06] Kein-Hong Man, esq. "A No-Frills Introduction to Lua 5.1 VM Instructions". English. In: (2006). URL: http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VM
pdf.

[Kei11] Keith Cooper, Linda Torczon. *Engineering: A Compiler*. English. 2nd Edition. Morgan Kaufmann, 2011. ISBN: 978-0120884780.

[Mar15] Mark Richards. *Software Architecture Patterns*. English. O'Reilly Media, Inc., 2015. ISBN: 978-1491971437. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5019.

[Prz15] Jakub Przywóski. "Python Reference (The Right Way)". In: (2015). [Online; accessed 1-June-2019]. URL: https://python-reference.readthedocs.io/en/latest/.

[Pyt] Python Software Foundation. *Python 3.7.3 documentation*. English. [Online; accessed 1-June-2019]. URL: https://docs.python.org/3/.

[Ric79] Richard Bornat. *Understanding and Writing Compilers: A do-it-yourself guide*. English. 3rd Edition. Palgrave, 1979. ISBN: 978-0333217320.

[Rob05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes. "The Implementation of Lua 5.0". English. In: (2005). URL: https://www.lua.org/doc/jucs05.pdf.

[Rob13] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes. "Closures in Lua". English. In: (2013). URL: https://www.cs.tufts.edu/~nr/cs257/archive/roberto-ierusalimschy/closures-draft.pdf.

[Ter10] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)*. English. 1st Edition. Pragmatic Bookshelf, 2010. ISBN: 978-1934356456.

[Thea] The Go Authors. *The Go Programming Language Specification*. English. [Online; accessed 1-June-2019]. URL: https://golang.org/ref/spec.

[Theb] The Rust Project. *The Rust Programming Language*. English. [Online; accessed 1-June-2019]. URL: https://doc.rust-lang.org/book/.

[Tim10] Timo Lilja. "Just-in-time Compilation Techniques". English. In: (2010). URL: https://wiki.aalto.fi/download/attachments/40010375/intro-report.pdf.

[Wik19] Wikipedia contributors. *Scannerless parsing — Wikipedia, The Free Encyclopedia*. English. [Online; accessed 1-June-2019]. 2019. URL: https://en.wikipedia.org/w/index.php?title=Scannerless_parsing&oldid=898030040.

[Yun05] Yunhe Shi, David Gregg, Andrew Beatt. "Virtual Machine Showdown: Stack Versus Registers". English. In: (2005). URL: https://www.usenix.org/legacy/events/vee05/full_papers/p153-yunhe.pdf.