UNIVERSITAT POLITÈCNICA DE CATALUNYA

Escola Superior d'Enginyeria de Manresa

The Design of an Experimental Programming Language and its Translator

The Nuua Programming Language

June 7, 2019

Bachelor's thesis submitted by

Èrik Campobadal Forés

in partial fulfillment of the requirements for the

Degree of ICT Systems Engineering

Advisor: Sebastia Vila Marta

"When you want to know how things really work, study them when they're coming apart."

— William Gibson, *Zero History*

To my family and many friends that encouraged me and emotionally supported me during all those years. Especially to my *mother*.

Acknowledgements

I would like to thank all the professors who have participated in my education, dedicating their time to help me develop my skills as an engineer. Especially the professors from the department of mining, industrial and ICT engineering. Special thanks to Sebastia Vila Marta who guided me during the development of this thesis.

Abstract

This thesis introduces the main concepts of compiler theory and interpreters in order to design a programming language together with a compiler and an interpreter to be able to execute programs written on it. The system that is designed consists of a layered architecture that encompasses the compiler and the interpreter in different phases. Additionally, a standard library is proposed in a very basic way and is implemented as part of the thesis. As a conclusion, we can observe different evolutions and improvements that can be added to the language, the compiler, and the interpreter.

Resum

Aquesta tesi introdueix els conceptes principals de teoria de compiladors i intèrprets per poder dissenyar un llenguatge de programació juntament amb un compilador i un intèrpret per poder executar programes escrits en ell. El sistema que es dissenya consisteix en una arquitectura per capes que engloba el compilador i l'intèrpret en diferents fases. Addicionalment es proposa una llibreria estàndard de manera molt bàsica i s'implementa com a part de la tesi. Com a conclusions podem observar diferents evolucions i millores que es poden anar afegint al llenguatge, al compilador i a l'intèrpret.

Contents

List of Figures

List of Tables

List of Listings

Part I.

Memory

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 Taula 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 Figura 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 Figura 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

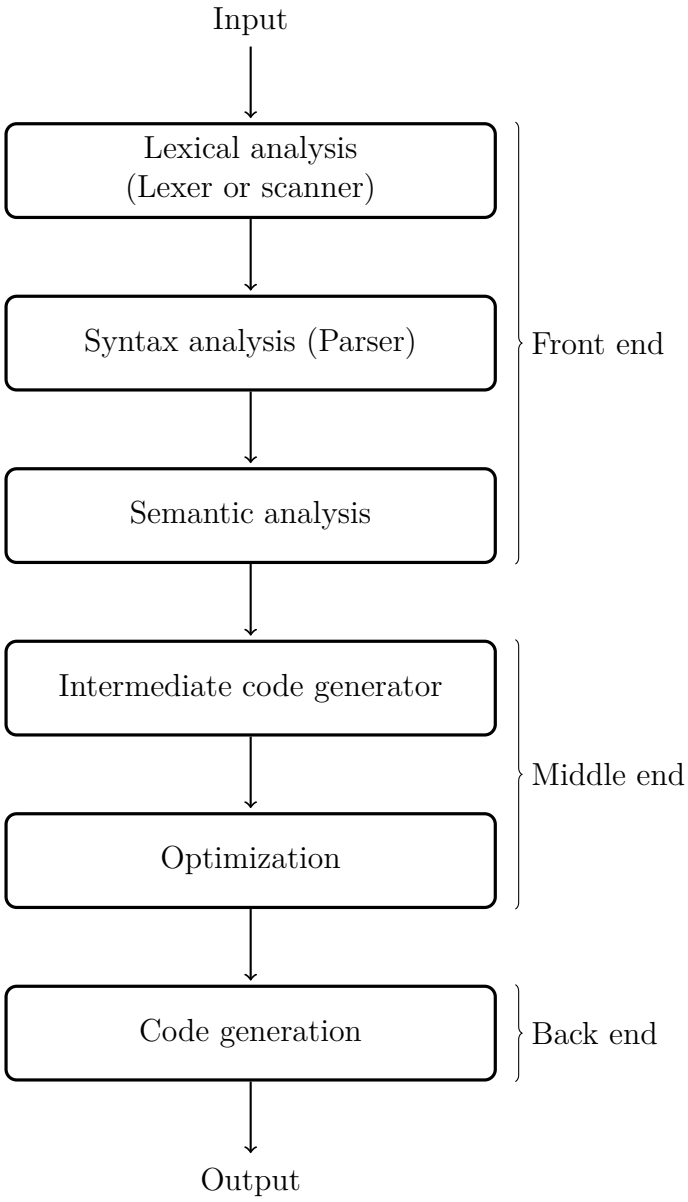


Figure 1.2.: Common compiler phases

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 Figura 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.

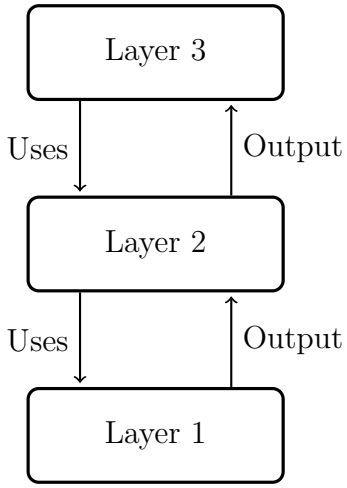


Figure 1.3.: Layered system

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.

Figura 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.

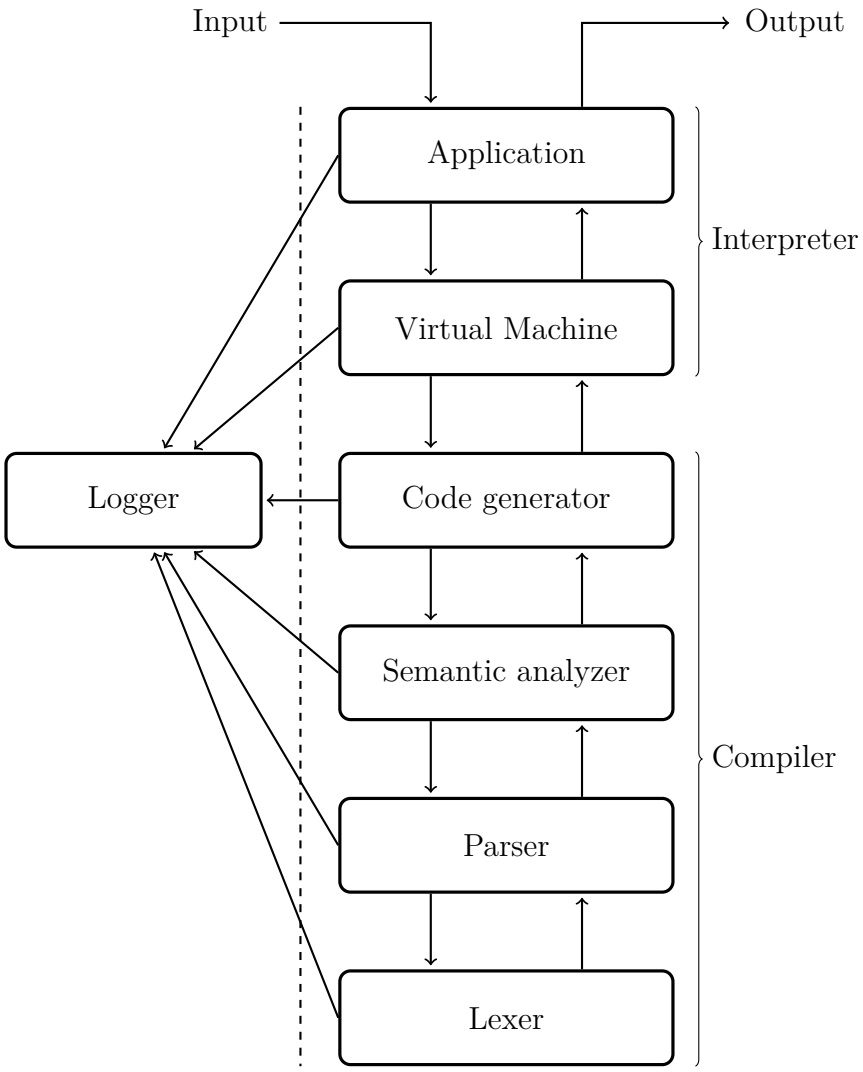


Figure 1.4.: Nuua's architecture diagram (Layered System)

2. Introduction To Nuua

This chapter introduces 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 discussed in Chapter 8.

Nuua's system architecture built in this thesis, as described in Secció 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.

This introduction dives directly to the usage of the language and provides example programs to understand it. This chapter does not deep dive into the syntax and the details of the language. The Nuua language specification can be found at Apèndix A

2.1. Hello, World

To introduce a programming language, the simple yet most common program that is written is a "Hello, World" program to get a first view the language syntax.

When writing a Nuua program, you first have to create a module. A module is a file that contains the logic of the program. Each module is a collection of top-level declarations. Top-level declarations can be functions, classes, exports, and use declarations. A special function called `main` must be included in the main module. This function is called the entry point and it's where the program will begin its execution. This function must also include a single argument of type `[string]` (list of strings) that includes the command line arguments given. The extension of the file is `.nu`. Therefore, a simple "Hello, World" program may be written as follows

```
fun main(argv: [string]) {
    print "Hello, World"
}
```

Listing 1: hello_world.nu

Since a module can have multiple functions we may add an additional function to it. The function can also have a return type, specified with a semicolon at the right of the

parenthesis followed by the type name. A modification on the `hello_world.nu` program can be made to support a function that returns the string `"Hello, World"`.

```
fun main(argv: [string]) {
    print hello()
}

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

Listing 2: hello_world2.nu

Notice how the order of the function declarations is not an issue. To make another final modification on the program, an additional parameter is added to the hello function.

```
fun main(argv: [string]) {
    print hello("World")
}

fun hello(name: string): string {
    return "Hello, " + name
}
```

Listing 3: hello_world3.nu

This parameter can be added to the `"Hello, "` string and it's then concatenated and returned. All this operations are discussed in Apèndix A.

2.2. Shorthands

Nuua comes with tons of shorthands to perform different actions. As a visual example, the Listing 3 can also be written as in Listing 4

```
fun main(argv: [string]) => print hello("World")

fun hello(name: string): string -> "Hello, " + name
```

Listing 4: shorthands.nu

Notice the difference between those two arrows. The `=>` is used to determine a single statement while the `->` is used to determine a single returned expression.

These shorthands also come in handy when defining a single statement while loop or any other control flow statement. As an example. The Listing 5 shows a program that prints "yes" or "no" depending on a condition.

```
fun main(argv: [string]) {
    number: int = 10
    if number == 10 => print "yes"
    else => print "no"
}
```

Listing 5: shorthands2.nu

We can also see how variable declarations are made by looking at line 2 on Listing 5. Variable declarations also have a shorthand to omit the type if an initial expression is provided.

```
fun main(argv: [string]) {
    number := 10
    while number > 0 {
        print "Higher than 10"
        number = number - 1
    }
}
```

Listing 6: shorthands3.nu

2.3. Modules

Nuua can also have different modules to encapsulate the logic. As an example, Listing 7 makes use of an exported function that can be found in Listing 8. If the function `add` was not exported, the compiler would complain.

```
use add from "module1"

fun main(argv: [string]) {
    print add(10, 20)
}
```

Listing 7: main1.nu

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

Listing 8: module1.nu

2.4. Loops and Ranges

Looping is a very easy in Nuua. Nuua supports the `while` and the `for ... in ...` statements. The while statement has already been seen in Listing 6. Listing 9 gives an overview to the `for ... in ...` statement and the `range` statement. The range statement is a way to create a list of numbers given an inclusive or an exclusive range.

```
fun main(argv: [string]) {
    for num in 0..10 {
        print num
    }
    for num, i in 0...10 {
        print i as string + ": " + num as string
    }
    for letter in "Erik" => print letter
}
```

Listing 9: loop.nu

Listing 9 also gives a hint about casting values to other data types.

2.5. Classes and Objects

Nuua can also perform limited object-oriented programming as can be seen on Listing 10

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

fun main(argv: [string]) {
    p := Person!{name: "Erik", age: 22}
    print p.name + " is " + p.age as string
}
```

Listing 10: oop.nu

Classes can also have methods bound to it, in fact, the code in Listing 10 can be written with a method as shown in Listing 11

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

fun main(argv: [string]) {
    p := Person!{name: "Erik", age: 22}
    print p.show()
}
```

Listing 11: oop.nu

3. Logger

This chapter starts the implementation of the Nuua system. Specifically, this chapter deals with the creation of the independent logger module found in the Nuua system diagram on Figura 1.4.

The logger is responsible for dealing with the error reporting of the system, so any layer can tell the logger to store messages (a log) and at the appropriate moment a layer could make the application crash (throw an error). Those crashes are controlled and must output a message to the user with information regarding the issue that caused the problem.

The errors that the application can throw must always include the source file, the line and the column where the error happened. However, there might be the possibility for a crash before any source code is analyzed, meaning that a crash without this information is still possible.

In case of an error, the reported error must be formatted accordingly and written in the standard error stream (stderr).

3.1. Error Design

Error reporting plays a very important job in a compiler. Recently, GCC released a new version that improves the diagnostics that the compiler outputs to have user-friendly errors similar to how rust does error reporting. Therefore, the error design must be taken into consideration when designing the Nuua logger.

The Figura 3.1 shows the concept for the error design.

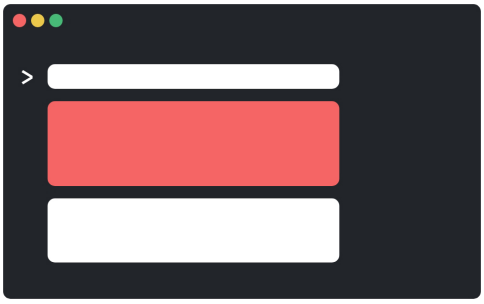


Figure 3.1.: Error logging concept

The first white line should include the *absoltute* path of the source file followed by the line and the column where the error happened.

The red paragraph should include an error description of the error that caused the crash. This paragraph must have a fixed width to ensure the error has a readable format.

The last white paragraph contains the line as seen in the source code of the program. The line must also include a reference to the column where the error happened so that a quick look at the line can determine the problem.

3.2. Logger Entities

The logger entity has all the properties that are mandatory fora log. This includes the absolute file path of the source program, the line of the file where the error happened and the column. Additionally, the error message that it has associated. A code representation is shown in the Listing 12.

```
// Represents a log entity.
class LoggerEntity {
    public:
        // Stores the file where the log comes from.
        const std::shared_ptr<const std::string> file;
        // Stores the line of the log.
        const line_t line;
        // Stores the column of the log.
        const column_t column;
        // Stores the message of the log.
        const std::string msg;
        // Creates a new log entity.
        LoggerEntity(
            const std::shared_ptr<const std::string> &file,
            const line_t line,
            const column_t column,
            const std::string &msg
        ) : file(file), line(line), column(column), msg(msg) {}
};
```

Listing 12: Logger entity class

3.3. Logger Class

Listing 13 shows a logger class implementation that satisfies all the previous statements. In short, it stores a list of all the log entities and provides methods to add and remove

logs. It also provides a crash method that outputs the logs and returns an exit code without calling `exit()`.

```
// Represents the logger used in the whole toolchain.
class Logger
{
    // Stores all the log entities.
    std::vector<LoggerEntity> entities;
    // Displays a specific log entity.
    void display_log(const uint16_t index, const bool red) const;
    public:
        // Stores the executable path.
        std::string executable_path;
        // Adds a new entity to the entity stack.
        void add_entity(
            const std::shared_ptr<const std::string> &file,
            const line_t line,
            const column_t column,
            const std::string &msg
        );
        // Pops an entity from the entity stack.
        void pop_entity();
        // Crashes the program by displaying the entity stack
        // and further returning an apropiate exit code.
        int crash() const;
};
```

Listing 13: Logger entity class

3.4. Cross-platform Caveat

Due to the fact that Nuua is supported in multiple platforms, there is an important feature mentioned that must be handled manually. The colored red output must be portable across all the major platforms. Windows is the platform that is most special and therefore, special treatment must be taken.

For windows platforms, the C++ header `windows.h` must be included. This header includes special functions to work with the windows terminal. The preprocessor macros `_WIN32` and `_WIN64` can be used to determine if the windows header is needed.

Figure Listing 14 shows the implementation of a red `printf` function working under all the major platforms.

```
#if defined(_WIN32) || defined(_WIN64)
    #include <windows.h>
#endif
#include <stdio.h>
#include <stdarg.h>
#include <cstdlib>

static int red_printf(const char *format, ...)
{
    va_list arg;
    int result;
    va_start(arg, format);
    #if defined(_WIN32) || defined(_WIN64)
        HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
        CONSOLE_SCREEN_BUFFER_INFO consoleInfo;
        WORD saved_attributes;
        GetConsoleScreenBufferInfo(hConsole, &consoleInfo);
        saved_attributes = consoleInfo.wAttributes;
        SetConsoleTextAttribute(hConsole, FOREGROUND_RED);
        result = vfprintf(stderr, format, arg);
        SetConsoleTextAttribute(hConsole, saved_attributes);
    #else
        // \x1b[31m\x1b[0m = (9 + '\0')
        char *fmt = malloc(sizeof(char) * (strlen(format) + 10));
        strcpy(fmt, "\x1b[31m");
        strcat(fmt, format);
        strcat(fmt, "\x1b[0m");
        result = vfprintf(stderr, fmt, arg);
        free(fmt);
    #endif
    va_end(arg);
    return result;
}
```

Listing 14: Red printf function

4. Lexer

The lexer is the lowest layer in the Nuua system as shown in Figura 1.4. The lexer job is to transform the program source code into a list of tokens. This step is also known as lexical analysis or tokenization.

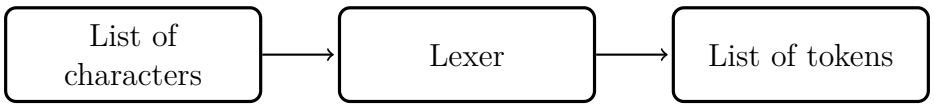


Figure 4.1.: Lexer overview

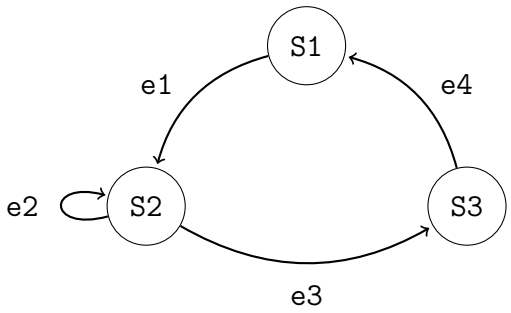
The lexer needs to provide an API to the layer above and one of the parameters of the scan function is the path of the source file to scan. The lexer is also responsible for opening the file and creating a `char` array with its contents. This can be done in the lexer class constructor seen in Listing 16. When the layer above wishes to scan the tokens the tokenization begins and therefore a list of tokens is built and returned.

4.1. Strategy

The strategy of the lexer is to use a state machine to determine the pattern of the tokens and be able to identify what token type it's being scanned. The state machine finds a pattern and creates a specific token depending on the state it landed. The state machine can also lead to no pattern. If there's a sequence of characters that don't match a pattern, an error is then thrown using the logger module as described in Chapter 3.

The implementation uses a C++ `switch` statement to match the first character of the pattern and further continues if needed depending on the patterns.

In case a `\n` is found, the lexer line is incremented and the column is reset to 0. Since the current line and column are properties in the lexer instance they only affect a single instance allowing multiple instances to scan different files at the same time. Figura 4.2 shows an example pattern found in the lexer.



(i) Diagram

State	Explanation
`S1`	Initial state of the state machine.
`S2`	Build string. `s += c`
`S3`	Create token

(ii) State table

Event	Condition	Action
`e1`	`c == '"'`	`c = next_char()`
`e2`	`c != '"'`	`c = next_char()`
`e3`	`c == '"'`	`-`
`e4`	`-`	`c = next_char()`

(iii) Event table

Figure 4.2.: Example finite state machine for strings

As shown in Figura 4.3, an efficient way to scan the file is using two pointers to some of the `chars` found in the source file. The first pointer points to the start of the token being scanned and the second one advances accordingly with the help of the state machine. When the state machine determines that a new token must be created the start pointer is used as the first character pointer in the token as explained in Secció 4.2 and the length of the token can be determined by using pointer arithmetic given:

$$\texttt{length} = \texttt{current\_ptr} - \texttt{start\_ptr} + 1$$

Since the string values can ignore the two `'"'` found at the start and at the end of the token, the length will be decremented by two and the start pointer is incremented by one to get only the string value. When a blank space is found in the initial state of the state machine the `start_ptr` pointer address is then set to the `current_ptr` address and the current parser column increases.

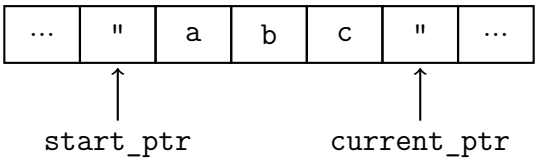


Figure 4.3.: Lexer scan technique

4.2. Tokens

Tokens represent specific lexeme patterns. The list of the patterns and the tokens associated with it can be seen on Taula 4.1, Taula 4.2 and Taula 4.3. The tokens must also take into consideration escaping characters. For example, a string representation might be willing to use the char `'"'` as part of the string value and not determine the end of the string. Therefore, there is a list of escaped chars that are taken into consideration as shown in Subsecció A.1.5.

Pattern	Token
\n	TOKEN_NEW_LINE
(	TOKEN_LEFT_PAREN
)	TOKEN_RIGHT_PAREN
{	TOKEN_LEFT_BRACE
}	TOKEN_RIGHT_BRACE
,	TOKEN_COMMA
.	TOKEN_DOT
..	TOKEN_DOUBLE_DOT
...	TOKEN_TRIPLE_DOT
-	TOKEN_MINUS

(i)

Pattern	Token
+	TOKEN_PLUS
/	TOKEN_SLASH
*	TOKEN_STAR
->	TOKEN_RIGHT_ARROW
!	TOKEN_BANG
!=	TOKEN_BANG_EQUAL
=	TOKEN_EQUAL
==	TOKEN_EQUAL_EQUAL
>	TOKEN_HIGHER
>=	TOKEN_HIGHER_EQUAL

(ii)

Table 4.1.: Nuua tokens (1)

Pattern	Token
=>	TOKEN_BIG_RIGHT_ARROW
:	TOKEN_COLON
return	TOKEN_RETURN
print	TOKEN_PRINT

(i)

Pattern	Token
use	TOKEN_USE
from	TOKE_FROM
elif	TOKEN_ELIF
in	TOKEN_IN
export	TOKEN_EXPORT

(ii)

Table 4.2.: Nuua tokens (2)

Pattern	Token	Pattern	Token
<	TOKEN_LOWER	fun	TOKEN_FUN
<=	TOKEN_LOWER_EQUAL	else	TOKEN_ELSE
<IDENTIFIER>	TOKEN_IDENTIFIER	true	TOKEN_TRUE
<STRING_EXPR>	TOKEN_STRING	false	TOKEN_FALSE
<INTEGER_EXPR>	TOKEN_INTEGER	while	TOKEN_WHILE
<FLOAT_EXPR>	TOKEN_FLOAT	for	TOKEN_FOR
as	TOKEN_AS	if	TOKEN_IF
or	TOKEN_OR	\0	TOKEN_EOF
and	TOKEN_AND	[	TOKEN_LEFT_SQUARE
class	TOKEN_CLASS	]	TOKEN_RIGHT_SQUARE
(i)		(ii)	

Table 4.3.: Nuua tokens (3)

To make an efficient token representation, instead of copying the `char` array (lexeme) that matches that token pattern a reference to the first character of the token is saved into the token instance and then the length is specified to know how many chars it does consist of. That way, no unnecessary memory is added to each token. The only needed thing is to keep the source code `char` array in memory and avoid reallocations till the token instance is not needed anymore. Figura 4.4 shows a visual representation. Listing 15 shows the token representation that is used in the Nuua compiler.

Among other information, the type of the token (as shown in Taula 4.1, Taula 4.2 and Taula 4.3) is also stored as part of the token. Additionally, the line and column where that token was found is also stored.

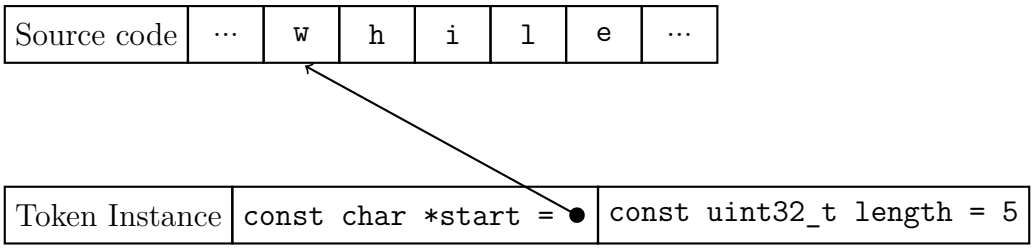


Figure 4.4.: Lexer token technique

```
class Token
{
    public:
        // The type of the token.
        const TokenType type;
        // A pointer to the first char.
        // (Not used as a null terminated char array)
        const char *start;
        // The length of the token.
        const uint32_t length;
        // The line where it is found.
        const line_t line;
        // The column where it is found.
        const column_t column;
        // String representation of the token names.
        static std::vector<std::string> token_names;
        // Pattern matching a token name.
        static std::vector<std::string> type_names;
        // Contains the escaped chars of the language.
        static const std::unordered_map<char, char> escaped_chars;
        // Create a new instance of a token given the
        // required parameters.
        Token(
            const TokenType type,
            const char *start,
            const uint32_t length,
            const line_t line,
            const column_t column
        ) : type(type), start(start), length(length),
            line(line), column(column) {}
        // Debug the token by printing it on the
        // screen with the correct format.
        void debug_token() const;
        // Convert the token to a string representation.
        std::string to_string() const;
        // Convert the token to its pattern string representation.
        std::string to_type_string() const;
};
```

Listing 15: Token class

4.3. Lexer Class

The lexer class represents a lexer instance to scan a file and return a list of tokens. Listing 16 shows the code used in the Nuua compiler to transform a `char` array corresponding to the source file to a list of token instances.

```
class Lexer
{
    // Stores the file where the tokens are being scanned.
    std::shared_ptr<const std::string> file;
    // Stores the start char of the token being scanned.
    const char *start;
    // Stores the current char of the token beeing scanned.
    const char *current;
    // Stores the current line in the source file.
    line_t line;
    // Stores the current column in the source file.
    column_t column;
    // Stores a list of reserved words for the identifiers.
    static const std::unordered_map<std::string, TokenType>
    ↪  reserved_words;
    // Generates a token error.
    const std::string token_error() const;
    // Build the token and set the start char to the current one.
    Token make_token(TokenType type);
    // Helper to build tokens.
    bool match(const char c);
    TokenType is_string(bool simple);
    TokenType is_number();
    TokenType is_identifier();
    // Reads a file and stores the contents in the current instance.
    void read_from_file(const std::shared_ptr<const std::string> &file);
    public:
        // Stores the source code of the file.
        std::unique_ptr<std::string> source;
        // Scans the source and stores the tokens.
        void scan(std::unique_ptr<std::vector<Token>> &tokens);
        // Initializes a lexer given a file name.
        Lexer(const std::shared_ptr<const std::string> &file);
};
```

Listing 16: Lexer class

5. Parser

The job of the parser is to transform the list of tokens returned by the Lexer into an abstract syntax tree.

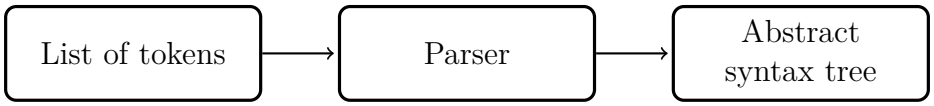


Figure 5.1.: Parser overview

There are a lot of different algorithms and strategies for parsing grammars. There are also a lot of parser generators out there to automatically build a parser given the grammar. However, this thesis implements a handwritten top-down recursive descend predictive parser for the following reasons:

1. To have as much control as needed over the parser. Especially for error reporting.
2. To enforce a zero-dependency policy.
3. Easy to build and fast enough for the job.
4. To learn how to build a recursive descent parser and learn how it works.

Recursive descent parsers are probably the simplest way to build a parser as mentioned in [Bob18, Section 6]. In fact, even if it's a simple parser it's used in very famous and large projects like `GCC` or `V8`.

As can be seen on [Joh08] the top down parers can be classified into the ones using backtracking and the ones using a predictive technique.

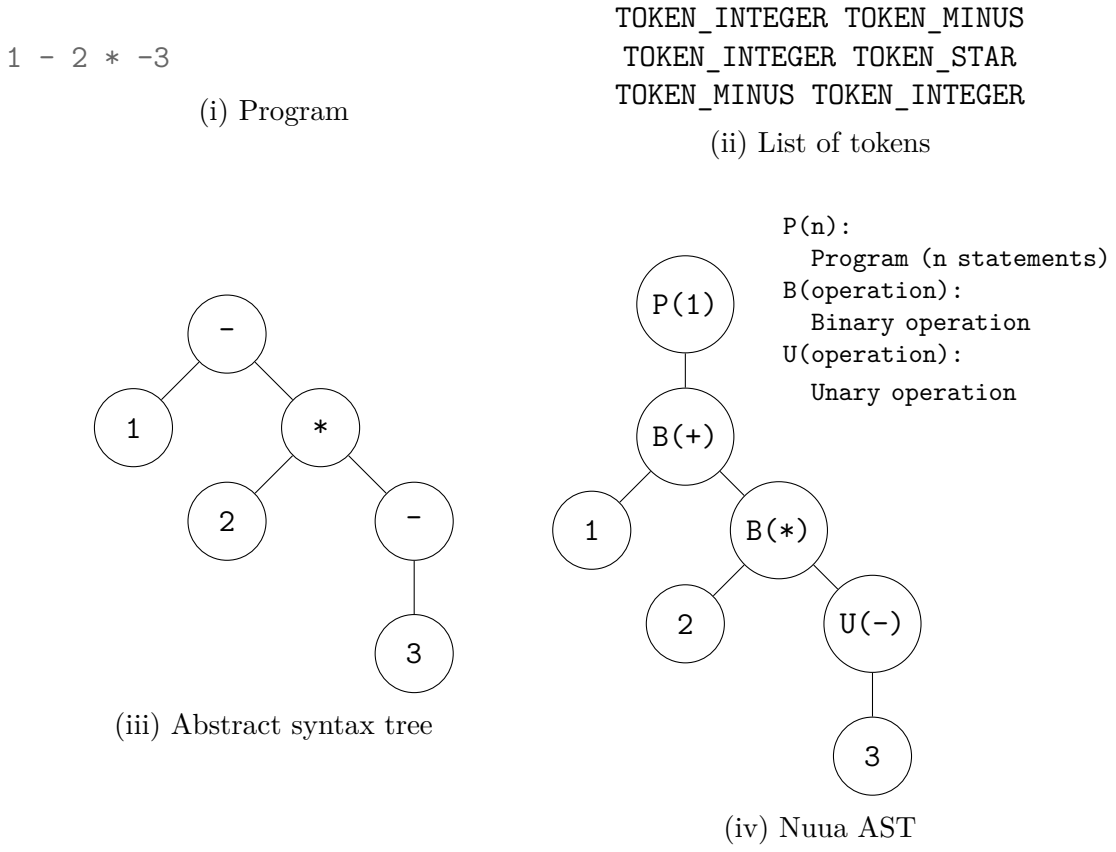
- *Backtracking parsers*: Backtracking parsers try to guess the correct production rule based on the current token. This guess can, however, lead to a dead-end and therefore a backtrack is needed to go back and make another decision. This is, however, very inefficient for programming languages due to the number of nonterminals found in them.
- *Predictive parsers*: Predictive parsers choose the production rule according to the current token and the next token found. The next token is called a lookahead token. Predictive parsers have a major issue with left-recursion and therefore,

the language grammar needs to be adapted to it. The Nuua grammar as seen on Secció A.1 is already adapted for left-recursion based on the solution mentioned on [Bob18, Section 6].

A recursive descent parser is a possible technique for implementing a top-down predictive parser that consists in the creation of a function for each nonterminal found in the grammar and then calling the functions depending on the current token and the lookahead.

5.1. Abstract Syntax Tree

The abstract syntax tree (AST) is the data structure used to represent the program in memory. This tree contains the representation of the program starting from an initial node `program` as seen in Secció A.1.2. Figura 6.1 shows an example AST.



(i) Program

(ii) List of tokens

(iii) Abstract syntax tree

(iv) Nuua AST

Figure 5.2.: Example abstract syntax tree

All the nodes found in the abstract syntax tree of Nuua can be found at Apèndix B. Each node is explained with the information it stores for further use in the compiler.

5.2. Value Data Types

The parser layer is responsible for creating the type class used in the nodes and in the blocks to determine the type of the values. Since some nodes will store information about their type and it will be added when the semantic analyzer runs. The blocks also need the type of the variables stores so it's convenient to define it here.

The data types supported in Nuua are already explained in Secció A.4, therefore Listing 17 shows the available data types supported with the addition of a value type that contains no type. This is used in call expression to functions that don't have a return type. This type is only used by the compiler and not the interpreter.

```
// Determines the available native types in Nuua.
typedef enum : uint8_t {
    VALUE_INT, VALUE_FLOAT, VALUE_BOOL,
    VALUE_STRING, VALUE_LIST, VALUE_DICT, VALUE_FUN,
    VALUE_OBJECT, VALUE_NO_TYPE
} ValueType;
```

Listing 17: Nuua data types

The type class is very complex. In fact, it must store the `VALUE_TYPE` and additional information about the type, for example, the inner type for values like lists or dictionaries. For the function type, it must store the function parameter types and its return type. For class types, it must store the class name used. Listing 18 shows the type class used in the Nuua compiler. As can be seen, there are a lot of additional methods bound to it to work with types. Those include copy methods, reset methods, type comparisons and additionally, methods that help to deal with the long list of casts, unary and binary operations found in Secció A.6.

```
class Type
{
    // Stores the asociation between the types
    // as string and the value type.
    static const std::unordered_map<std::string, ValueType> value_types;
    // Determines the string representation
    // of the value types.
    static const std::vector<std::string> types_string;
    public:
        // Stores the type.
        ValueType type;
        // Stores the inner type if needed.
        // Used as return type for functions.
        std::shared_ptr<Type> inner_type;
        // Class name.
        std::string class_name;
        // Store the parameters of a function type.
        std::vector<std::shared_ptr<Type>> parameters;
        // Create a value type given the type.
        Type() : type(VALUE_NO_TYPE) {}
        Type(const ValueType type)
            : type(type) {}
        // Create a type given a value type and the inner type.
        Type(
            const ValueType type,
            const std::shared_ptr<Type> &inner_type
        ) : type(type), inner_type(inner_type) {}
        // Create a type given a string representation of it.
        Type(const std::string &name);
        // Create a type given an expression and a
        // list of code blocks to know the variable values.
        Type(
            const std::shared_ptr<Expression> &rule,
            const std::vector<std::shared_ptr<Block>> *blocks
        );
        // Create a function type given the function itself.
        Type(const std::shared_ptr<Function> &fun);
        // Additional methods used as helpers for casts,
        // unary and binary operations and for other operations
        // like copy or
        // ...
};
```

Listing 18: Type class

5.3. Block Scope and symbol table

The parser layer will also create the block representation explained in Secció A.2. The way the scope block is built must be explained here since it will be attached as part of some of the AST nodes. This acts as a symbol table for a specific scope.

A symbol table is used to store the name of the variables in the block and further verify if they are declared or get specific data of them. For example, the symbol table can verify if a variable is declared, the type of the variable, the register where it is placed, etc. Additionally, the block does also store the defined classes. This is done due to the fact that the block class is also used as the module scope internally.

5.3.1. Block Variable Type

The block variable type is the representation of a variable stored in the block. A variable must include the following fields:

- The variable type.
- The AST node where it appears (The declaration).
- A register where this variable is stored.
- If the variable is exported or not.
- The node where it was last used.

The variable name, however, is not part of this class, since to reference this variable class, a hashmap is used and the hashmap key is a variable name.

Listing 19 shows the class representation of a block variable type.

```
class BlockVariableType
{
    public:
        // Represents the variable type.
        std::shared_ptr<Type> type;
        // Stores the AST node where this variable is.
        std::shared_ptr<Node> node;
        // Represents the register where it's stored.
        register_t reg = 0;
        // Determines in the variable is exported.
        // (only applies to TLDs).
        bool exported = false;
        // Represents the last use of the variable (Variable life).
        std::shared_ptr<Node> last_use;
        // Constructors.
        BlockVariableType() {};
        BlockVariableType(
            const std::shared_ptr<Type> &type,
            const std::shared_ptr<Node> &node,
            const bool exported = false
        ) : type(type), node(node), exported(exported) {}
};
```

Listing 19: BlockVariableType class

5.3.2. Block Class Type

The block class type is similar to the block variable type but stores much fewer fields. Also, this information is only used in the module scope. The block class type stores the following information:

- The block scope of the class.
- If the class is exported or not.
- The node where it was last used.

Listing 20 shows the class representation of the block class type. It can be seen that a forward declaration to the block class must be used till its defined.

```
class Block; // Forward declaration.
class BlockClassType
{
    public:
        // Stores the block scope of the class.
        std::shared_ptr<Block> block;
        // Determines if the class is exported.
        bool exported = false;
        // Stores the AST node where this variable is.
        std::shared_ptr<Node> node;
        // Constructors.
        BlockClassType() {}
        BlockClassType(
            const std::shared_ptr<Block> &block,
            const std::shared_ptr<Node> &node,
            const bool exported = false
        ) : block(block), node(node), exported(exported) {}
};
```

Listing 20: BlockClassType class

5.3.3. Block Class

With the form of the variables and the classes that are stored in the block the final definition for a block can be given, therefore, the block must then be able to store variables, classes and additionally provide with methods to help perform different operations on them.

Listing 21 shows the block class used in the Nuua compiler. As can be seen, two hashmaps are used to store the relation between variable names and its class shown in Listing 19 and the same with the class name with its class shown in Listing 20.

```
class Block
{
    public:
        // Stores the variable name and the type of it.
        std::unordered_map<std::string, BlockVariableType> variables;
        // Stores the custom types of the block.
        std::unordered_map<std::string, BlockClassType> classes;
        // Gets a variable from the current block or returns nullptr.
        BlockVariableType *get_variable(const std::string &name);
        // Gets a class from the current block or returns nullptr.
        BlockClassType *get_class(const std::string &name);
        // Sets a variable.
        void set_variable(const std::string &name, const
        ↪  BlockVariableType &var);
        // Sets a class.
        void set_class(const std::string &name, const BlockClassType
        ↪  &c);
        // Determines if a variable is exported.
        bool is_exported(const std::string &name);
        // Determines if a class is exported.
        bool is_exported_class(const std::string &name);
        // Determines if the block have a variable.
        bool has(const std::string &name);
        // Determines if the block have a class.
        bool has_class(const std::string &name);
        // Debug the block by printing it to the screen.
        void debug() const;
        // Helper to get a single variable out of a list of blocks.
        // It iterates through it starting from the end till the
        ↪  front.
        static BlockVariableType *get_single_variable(const std::string
        ↪  &name, const std::vector<std::shared_ptr<Block>> *blocks);
};
```

Listing 21: Block class

5.4. Parser Class

The parser class is able to parse the program tree by using some recursive logic. When a use declaration is found when parsing, the contents of the imported module are parsed using a new instance of the parser class. That way, the first parser class contains the AST of all the program. It's also good to note that the parser class formats the given paths to absolute paths and checks for the file location to determine if the imported

module belongs to a relative file or the standard library. The parser class uses a similar technique as the Lexer when dealing with the current token. The parser has a pointer to the current token being parser and the lookahead token can be calculated by using pointer arithmetic. Listing 22 shows the implementation of the parser.

```
class Parser
{
    // Stores the current parsing file.
    std::shared_ptr<const std::string> file;
    // Stores a pointer to the current token beeing parsed.
    Token *current = nullptr;
    // Consumes a token and returns it for futher use.
    Token *consume(const TokenType type, const std::string &message);
    // Returns true if the token type matches the current token.
    bool match(const TokenType token);
    // Returns true if any of the given token types matches the current
    ↪  token.
    bool match_any(const std::vector<TokenType> &tokens);
    // Expressions
    std::shared_ptr<Expression> primary();
    std::shared_ptr<Expression> unary_postfix();
    // ... Other expression production rules.
    // Statements
    std::shared_ptr<Statement> fun_declaration();
    std::shared_ptr<Statement> use_declaration();
    // ... Other statement production rules.
    public:
        // Parses a given source code and returns the code.
        void parse(
            std::shared_ptr<std::vector<std::shared_ptr<Statement>>>
            ↪  &code
        );
        // Creates a new parser and formats the path.
        Parser(const char *file);
        // Creates a new parser with a given formatted and initialized
        ↪  path.
        Parser(std::shared_ptr<const std::string> &file)
            : file(file) {}
        // ... Other helpers and debugging methods.
};
```

Listing 22: Parser class

5.5. In-memory Module Cache

During the parsing of a module, this might require the use of other modules. When importing a module, the required module is first parsed. However, The module might have already been parsed by a previous import, so there's no need to parse that file again. This is prevented by having an in-memory cache to avoid re-parsing files more than once.

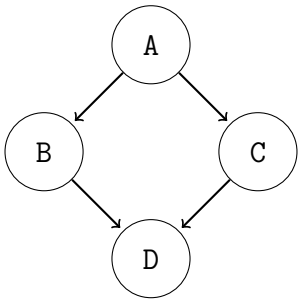


Figure 5.3.: A use case for the in-memory parser cache

This situation increases performance when a library is used by a lot of modules. Figura 5.3 illustrates this situation by having a module `D` imported more than once. The in-memory cache will prevent the second parse and will assign that use node the right AST parsed previously.

6. Semantic Analyzer

The semantic analyzer is run after the AST has been generated and therefore the program is all in the same data structure. The job of the semantic analyzer is not to transform the AST but to analyze it. There is the possibility for optimizations to be performed during the analysis to improve the AST. This includes for instance loop unrolling, constant folding or dead code elimination. However, those things are not part of this thesis and therefore those optimizations have been mentioned in Secció 11.1. However, this thesis deals with other types of optimizations. Specifically, bytecode optimization is performed to some extent.

In short, the job of the analyzer is to perform the following things:

- Create the symbol table for each scope and block found. This includes all module scopes and each block scope found.
- Get all the type information from the expressions.
- Perform the semantic checks on all the statements and expressions found in the AST. All the requirements for those nodes are explained in Secció A.5 and Secció A.6 respectively.
- Add specific node information. This information has been mentioned individually on each tree node on Apèndix B.

6.1. Technique

Before any analysis begins, a full module scope analysis needs to be performed. This is done to ensure that when functions are analyzed (and so do their statements) the module scope is already valid and contains the information needed to have a correct statement analysis. For instance, if a function has a statement that makes a call to another function on the module, that callee function must be known, so its signature needs to be analyzed first.

To solve this, a first analysis is performed only on the top level declarations (module scope) found in the module. If one of the top-level declarations is a use declaration, then that module is first analyzed completely before the current analysis can continue. The top-level analysis just gets information about the top-level declarations and does not perform further analysis. That means that it just gets the information of the functions,

classes and so on without analyzing the function bodies. Top level classes are also analyzed only on the top level of their bodies. Once all this phase is complete, the code analysis begins in an arbitrary order. This can be done as the top level is already analyzed and therefore all the symbol table for the module block is already built and the types are known.

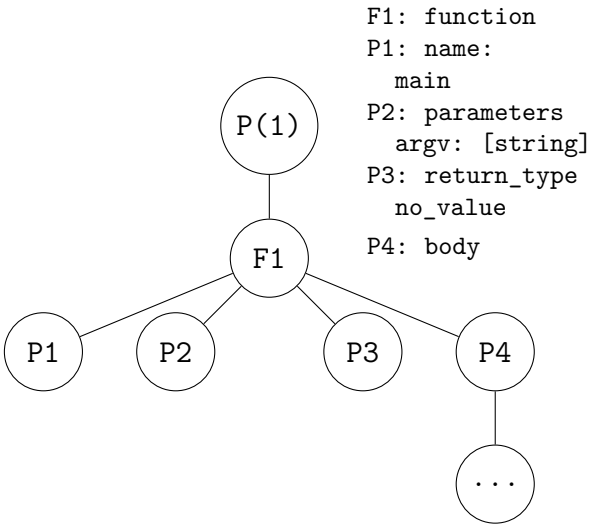
Once the code analysis begins the function bodies are analyzed and so do classes methods. The analysis run by recursively walking the AST and further performing all the checks mentioned in Secció A.5 and on Secció A.6 depending on the type of node being analyzed. This analysis is also recursive.

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

(i) Program

```
TOKEN_FUN TOKEN_IDENTIFIER
TOKEN_LEFT_PAREN
TOKEN_IDENTIFIER TOKEN_COLON
TOKEN_LEFT_SQUARE
TOKEN_IDENTIFIER
TOKEN_RIGHT_SQUARE
TOKEN_RIGHT_PAREN
TOKEN_LEFT_BRACE TOKEN_NEW_LINE
TOKEN_RIGHT_BRACE TOKEN_NEW_LINE
TOKEN_EOF
```

(ii) List of tokens



(iii) Abstract Syntax Tree

Variable	Type
main	([string])

(iv) Module scope symbol table

Figure 6.1.: Example program top-level analysis

Among other mentioned checks that are specific to certain nodes, some of the checks the analyzer performs are:

- Declared variables.
- Function return value match.
- Functions without return type usage.
- Argument type match.
- Assignment type match.
- List / String / Dictionary index type.
- Variable lifetime (`last_use`).
- Use only exported declarations.
- Iterator check on required nodes (must be list / string / dict).
- Type matching.
- `main([string])` required on the main module.

6.2. Type Inference

Type inference is done in Nuua by the `Type` class seen on Listing 18. However, it is used in this layer to get the expression types of the program. Type inference is the ability to get the type of expressions at compile time. Type inference is a step needed to perform type checks on the AST. Most nodes require a type check at some point and therefore, type inference can be used to get the type of the expressions on such nodes. The `Type` class has a constructor that accepts an expression and a list of blocks. This constructor can determine the expression type and if the expression contains variables, their type is obtained from the block list, starting from the last block going back till the first block of the list. Although errors are checked, the expression should always be analyzed prior to any type inference to avoid issues.

6.3. Module class

The module class is used to analyze a module. It automatically creates a new instance of it when it needs to analyze the AST of a use declaration.

Listing 23 shows the implementation of the module class used by the analyzer

```
class Module
{
    // Stores the file name of that module.
    std::shared_ptr<const std::string> file;
    // Stores the code of that module.
```

```
std::shared_ptr<std::vector<std::shared_ptr<Statement>>> code;
// Stores the blocks used while analyzing the module.
std::vector<std::shared_ptr<Block>> blocks;
// Return variable if needs to be checked. Only 1 can exist since
// it can only analyze 1 function at a time.
std::shared_ptr<Type> return_type;
// Analyzes the TLDs of the current module.
void analyze_tld();
// Analyzes the given top level declaration.
void analyze_tld(const std::shared_ptr<Statement> &tld, const bool
↪  set_exported = false);
// Analyzes the class top level declarations.
// void analyze_class_tld(const std::shared_ptr<Class> &c);
void analyze_class_tld(const std::shared_ptr<Statement> &tld, const
↪  std::shared_ptr<Block> &block);
// Analyzes the code.
void analyze_code();
// Analyzes the statement.
void analyze_code(const std::shared_ptr<Statement> &rule, bool
↪  no_declare = false);
// Analyzes the expression.
void analyze_code(const std::shared_ptr<Expression> &rule, const
↪  bool allowed_noreturn_call = false);
// Analyzes the block.
std::shared_ptr<Block> analyze_code(
    const std::vector<std::shared_ptr<Statement>> &code,
    const std::vector<std::shared_ptr<Declaration>> &initializers =
    ↪  std::vector<std::shared_ptr<Declaration>>(),
    const std::shared_ptr<Node> &initializer_node =
    ↪  std::shared_ptr<Node>()
);
// Declares a variable to the most top level block.
void declare(const std::shared_ptr<Declaration> &dec, const
↪  std::shared_ptr<Node> &node = std::shared_ptr<Node>());
// Check if the given module have all the classes defined.
bool check_classes(const std::vector<std::string> &classes, const
↪  std::shared_ptr<Node> &fail_at);
public:
    // Stores the main block of that module.
    std::shared_ptr<Block> main_block = std::make_shared<Block>();
    // Main module constructor
    Module(std::shared_ptr<const std::string> &file)
        : file(file) {}
```

```
        // Analyzes the module and adds it's entry to the modules
        ↪  symbol table.
        std::shared_ptr<Block>
        ↪  analyze(std::shared_ptr<std::vector<std::shared_ptr<Statement>>>
        ↪  &code, const bool require_main = false);
};
```

Listing 23: Module class

6.4. In-memory Module Cache

Since a module might be imported more than once as shown in Figura 5.3 the analyzer have an in-memory cache to know when a module has been analyzed to avoid redundant analyses. Once a module has been analyzed it is added into a list of analyzed modules and when a new module is required to be analyzed is first checked if it has been. If the module has already been analyzed it returns its module scope symbol table and otherwise, it gets analyzed.

7. Code generator

The code generation layer is responsible for generating the bytecode that will be interpreted by the virtual machine. Bytecode instructions are very similar to hardware instructions but at a much higher level (they are not as close to hardware). Figura 7.1 shows the job of the code generator.

This chapter does not define all the available opcodes found in Nuua. All those opcodes can be found on Apèndix C.

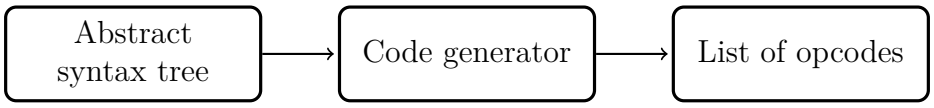


Figure 7.1.: Code generator overview

The goal of this layer is to be able to transform a data structure into a linear sequence of opcodes. The opcodes have been designed for the needs of the project. This transformation is performed in a similar way as the semantic analysis explained in Chapter 6. It first performs a top-level walk to assign each function a register (known as tld registration) and if a use declaration is found, it is also registered by after the current module. Therefore, it delays it's registration until the current one it's analyzed. This has no impact on the resulting code rather than having each module registered sequentially (easier to debug).

7.1. Call Stack and Frames

Each function call in Nuua introduces a new frame that is pushed into the call stack. The call stack is a stack that stores the active frames in the application. When the function ends the frame is popped from the call stack. The frame contains information about the function variables, specifically, it contains the registers where the function values live. Space is allocated when the frame is created and it's then deallocated when the function ends. This is done by the virtual machine as seen in Chapter 8.

The frame consists of an array containing the registers that the function use. Those registers are allocated when the frame is created (when a function is called). They must

also contain the amount of registers it has allocated and a return address that is used by the virtual machine to go back to the opcode where the function was called.

To allow passing parameters between frames (function parameters and return values) all frames share a common value stack. When a function is called, its arguments are pushed to the stack and once the function frame is set they are poped from it. When a function returns a value, it is pushed to the stack and the frame ends. When the last frame is restored, it then pops the value to a register for further use.

Therefore, Listing 24 shows the frame class that is used in the Nuua compiler and virtual machine.

```
class Frame
{
    public:
        // Stores the registers.
        std::unique_ptr<Value[]> registers;
        // Stores the registers size.
        registers_size_t registers_size = 0;
        // Stores the return address to get back to the original
        ↪  program counter.
        opcode_t *return_address = nullptr;
        // Allocates the space to store the registers.
        void allocate_registers(registers_size_t size);
        // Frees the allocated register space.
        void free_registers();
        // Setup the frame.
        void setup(registers_size_t size, opcode_t *return_address);
};
```

Listing 24: Frame class

7.2. Register Allocation

Compiler allocation takes place during code generation by the use of a `FrameInfo` class that helps to determine the right amount of registers needed to compile a function and also optimizes the registers used by using a very simple strategy. The frame information class is shown in Listing 25.

As it can be seen, the frame info is used as a register allocator, it basically consists of two lists of registers and a current register index.

- *Free Registers*: Stores the registers that were freed and can be used again.
- *Protected Registers*: Stores the registers that are protected from being freed (as they are used in variables and not anonymous expressions).

- *Current Register*: Stores the index of the register that will be given in case no free register is available.

```
class FrameInfo
{
    // Stores the registers that are free to use again.
    std::vector<register_t> free_registers;
    // Stores the registers that are protected to get free unless
    ↪  forçed.
    std::vector<register_t> protected_registers;
    public:
        // Stores the next register to give in case no free ones are
        ↪  available.
        register_t current_register = 0;
        // Method to return an available register.
        // It will try to get it from the free_registers
        // otherwise it will return a new register and
        // increment current_register by 1
        register_t get_register(bool protect = false);
        // Used to free a register.
        void free_register(register_t reg, bool force = false);
        // Resets the frame info.
        void reset();
};
```

Listing 25: FrameInfo class

When the function `get_register` is called, it first performs a check on the `free_registers` list. If that list is empty it means that there is no free register to be used, and therefore the `current_register` index is the register that will be given and it's incremented by 1. If the list is not empty it means that there are free registers available and then the last element of the list is poped and is the register that will be given. In any case, if the `protect` parameter is true the given register is also added to the `protected_registers` list before the function ends.

When the function `free_register` is called, it first performs a check to see if the register that is attempted to be freed is a protected register. If it is protected and the `force` parameter is false the function ends without any modifications. If it is protected and the `force` parameter is true, the element it's removed from the `protected_registers` and the register is added to the `free_registers` list. In case the register it's not in the `protected_registers` list, it is added to the `free_registers` list without any further checks.

The reason behind this strategy is because when the code generation is taking place each node in the AST gets compiled and simple expressions like `1 + 2` need to use 3 registers.

1 for the destination where the result of the operations is going to be stored (this calls the `get_register` function found in the register allocator) and 2 for the operands. However, when the operation is complete and the result is in the destination register, the other 2 registers are not used again. They are then considered garbage. However, that means that not only they are garbage but they are also using part of the memory that could be used by further registers.

The solution is to use the register allocator presented in Listing 25 to free the values of the 2 registers that can become garbage after the operation is complete. The reason there are protected registers is in case the operation was something like `a + 2`. Where the register where the variable `a` is stored should not be freed because the variable could be used afterward. This also means that when a variable is declared, the register that is bound to be a protected register and can only be freed when the variable reaches the last node where it is used (This is already known since the semantic analysis determines the variable lifetime). The variable can be freed by using the force parameter.

In the Figura 7.2 there can be seen that the registers where the same destination register is used in different operations. This is thanks to this register allocation technique. The number of required registers for the function is also much lower needing much less space to allocate in order to do all of its tasks.

7.3. Program Memory

The program memory represents the place where the program is going to be stored. Not only does include all the generated bytecode but also includes the constant pool where all the constants are stored and also a way to store the relation between opcodes and their file, line, and column where they are found in the program source file in case the virtual machine needs to throw an error. The strategy used is explained in Subsecció 7.3.1. Listing 26 shows the class used in Nuua to represent the program memory.

```
class Memory
{
    public:
        // This stores the opcodes and consant indexes of the code.
        std::vector<opcode_t> code;
        // Stores the value constants.
        std::vector<Value> constants;
        // Stores the lines corresponding to the opcodes.
        std::unordered_map<size_t, std::shared_ptr<const std::string>>
        ↪  files;
        // Stores the lines corresponding to the opcodes.
        std::unordered_map<size_t, line_t> lines;
        // Stores the lines corresponding to the opcodes.
        std::unordered_map<size_t, column_t> columns;
        // Dumps the memory.
        void dump();
};
```

Listing 26: Memory class

7.3.1. Bytecode and Source File Relationship

The number of opcodes generated can be pretty high but it's not an issue considering each opcode is just 64 bits (64 bits * operand). However, each opcode should be bound to a file pointer, a line, and a column. This includes so many extra bytes for no reason. This would, however, be quite easy to implement by creating 3 additional arrays with the same length of the opcodes + operands (again it uses a lot of memory). A proposed solution uses 3 hashmaps (for the files, lines, and columns) where the hashmap keys correspond to an arbitrary index in the bytecode array where the corresponding value has changed.

This is better explained with an example as shown in Figura 7.2. As the annotations on the right side of the bytecode show, there are only 3 line changes. Therefore, the hashmap that corresponds to the lines will have 3 entries where the keys are the indexes where this line change occur. Following the same code we can deduce:

- The first index is 0 with the value of 1.
- The second index 2 with the value of 2.
- The third index 19 with the value of 3.

This strategy allows deducing the line where the code failed by looking at the hashmap keys and getting the value of the highest key that is lower or equal to the current index. Let's assume the code fails at the annotation marked with a *. The fail index is 8 (the

opcode is on index 8). By looking at the hashmap keys previously stated, the highest key lower than 8 is the key 2 and therefore the value is line 2.

The same strategy applies to the file and the column as well.

In short then, one of the jobs of the code generator is to add the necessary keys and values to the hashmaps.

```
fun main(argv: [string]) {
    1 + 2 + 3
    4 + 5 + 6
}
```

(i) Program

```
POP R-00000 // Line 1
LOAD_C R-00001 C-00000 // Line 2
LOAD_C R-00002 C-00001
ADD_INT R-00003 R-00001 R-00002 // *
LOAD_C R-00002 C-00002
ADD_INT R-00001 R-00003 R-00002
LOAD_C R-00002 C-00003 // Line 3
LOAD_C R-00003 C-00004
ADD_INT R-00004 R-00002 R-00003
LOAD_C R-00003 C-00005
ADD_INT R-00002 R-00004 R-00003
RETURN
EXIT
```

(ii) Bytecode generated

Figure 7.2.: Example program bytecode

7.4. Program

With the program memory taking care of all the opcodes, constants and the bytecode relationship with the source file, a program can be represented by its memory and the main frame (where the global registers live). Listing 27 shows the program class used in Nuua.

```
class Program
{
    public:
        // Stores the main frame of the program.
        Frame main_frame;
        // Stores the program memory (The main code).
        std::unique_ptr<Memory> memory;
        // Constructor to allocate the memory of the
        // program memory. Can't be done here since
        // Memory is a forward declared class.
        Program();
};
```

Listing 27: Program class

7.5. Values

Values are very important for the virtual machine, but the code generator is the one that defines how values look like since the code generator already creates values (the constants) for the program. Therefore, the values need to be explained here. As Secció A.4 explains, the values may have any of those types but not more than one at the same time. The code generator defines the C++ data types for each value as shown in Listing 28.

```
typedef int64_t nint_t;
typedef double nfloat_t;
typedef bool nbool_t;
typedef std::string nstring_t;
typedef std::vector<Value> nlist_t;
typedef ValueDictionary ndict_t;
typedef ValueFunction nfun_t;
typedef ValueObject nobject_t;
```

Listing 28: Nuua data type definitions

A value must know its type and its current value. Due to the fact that the values are very different types but only a single one can exist at the same time, a C union can be used to avoid wasting memory. Modern C++ has a `std::variant` that is basically a C union with steroids since it calls the deconstructors when the data changes and also stores the type it owns. A value should be able to be retyped and changed without the need to allocate a new value and deallocate an old one. Reusing memory is something that will be needed since registers are re-used and the fewer memory allocations, the faster the execution becomes.

```
class Value
{
    void build_from_type(const Type *type);
    public:
        // The type of the value.
        Type type;
        // Using a variant to avoid unessesary memory.
        std::variant<
            // Stores the reporesentation of the VALUE_INT.
            nint_t,
            // Stores the representation of the VALUE_FLOAT.
            nfloat_t,
            // Stores the representation of the VALUE_BOOL.
            nbool_t,
            // Stores the representation of the VALUE_STRING.
            nstring_t,
            // Stores the representation of the VALUE_LIST.
            std::shared_ptr<nlist_t>,
            // Stores the representation of the VALUE_DICT.
            std::shared_ptr<ndict_t>,
            // Stores the representation of the VALUE_FUN.
            nfun_t,
            // Stores the representation of the VALUE_OBJECT.
            std::shared_ptr<nobject_t>
        > value;
        // Constructors
        // ...
        // Retypes the value.
        void retype(ValueType new_type, const std::shared_ptr<Type>
        ↪  &new_inner_type = std::shared_ptr<Type>());
        // Copies the current value to the destnation.
        void copy_to(Value *dest) const;
        // Gets a string representation of the value.
        std::string to_string() const;
};
```

Listing 29: Value class

As seen on Listing 30, the value of a function consists of the opcode index where the execution begins and a number that corresponds to the number of registers needed to allocate in order for the function to work. By using the register allocator, this number may be determined by the `current_register` that stores the max number of registers the frame needed when performing the code generation in the specific frame.

```
class ValueFunction
{
    public:
        // Stores the function index where it's code begin.
        size_t index;
        // Stores the ammount of registers needed
        registers_size_t registers;
        // Basic constructor for the function value.
        ValueFunction(size_t index, registers_size_t registers)
            : index(index), registers(registers) {}
};
```

Listing 30: ValueFunction class

Listing 31 shows the dictionary value used in Nuua. The dictionary needs the hashmap to store the values and key order.

```
class ValueDictionary
{
    public:
        // Represents the hashmap of the dictionary.
        std::unordered_map<std::string, Value> values;
        // Represent the current key order of the hashmap.
        // Even with an ordered map, the key order is still
        // important.
        std::vector<std::string> key_order;
        // Adds a new element to the dictionary.
        void insert(const std::string &key, const Value &value);
        // The basic constructor of the dictionary.
        ValueDictionary(const std::unordered_map<std::string, Value>
        ↪  &values, const std::vector<std::string> &key_order)
            : values(values), key_order(key_order) {}
};
```

Listing 31: ValueDictionary class

The last complex type of data is the object. An object, as seen on Listing 32 only needs the set of registers that correspond to its properties. Therefore, the length of the registers array is determined by the number of properties of the class it represents.

```
class ValueObject
{
    public:
        // Stores the registers containing the properties values.
        std::unique_ptr<Value[]> registers;
        // Create a new object value and allocates the registers given
        ↪  the size.
        ValueObject(registers_size_t size);
};
```

Listing 32: ValueObject class

Since the values of `nlist_t`, `ndict_t`, `nobject_t` are allocated in the heap and used as pointers then the `copy_to` method copies the pointer and not the value, making them pass-by-reference values. Thanks to the C++ smart pointers, the pointer value is only deallocated when no value is using it.

7.6. Compiler Class

The compiler class is used as the code generator for the Nuua compiler. The Nuua compiler class is responsible for allocating a new program and it performs all the register allocations as mentioned in this chapter. It also performs basic optimizations to avoid extra loadings by the use of a suggested register.

The suggested register is used instead of the need to move a register without reason (due to the nature of the tree compilation). By using a suggested register, some operations can avoid this extra step or avoid requesting a new register to the register allocator and using the suggested one.

If complex structures (like lists and dictionaries) are found to be constant (it means that all of the elements of the list or dictionary are constants and therefore, their values can be created at compile time) the whole list or dictionary value is created at compile time avoiding the need to push each individual values at run time.

Listing 33 shows the compiler class that Nuua use to generate the bytecode.

```
class Compiler
{
    // Stores the program itself where everything is beeing compiled
    ↪  to.
    std::shared_ptr<Program> program;
    // Stores the current block.
    std::vector<std::shared_ptr<Block>> blocks;
    // Stores the global frame information.
```

```
    FrameInfo global;
    // Sotres the current local information (it resets automatically).
    FrameInfo local;
    // Compiles a module.
    void compile_module(const
     ↪  std::shared_ptr<std::vector<std::shared_ptr<Statement>>> &code,
     ↪  const std::shared_ptr<Block> &block);
    // Registers the top level declarations by assigning a register to
     ↪  them.
    // it registers all TLD of all modules in the same main frame
     ↪  info.
    void register_tld(const
     ↪  std::shared_ptr<std::vector<std::shared_ptr<Statement>>> &code,
     ↪  const std::shared_ptr<Block> &block);
    // Compiles a list of statements
    // Defines a basic compilation for a Statement.
    void compile(const std::shared_ptr<Statement> &rule);
    // Defines a basic compilation for an Expression. (Returns the
     ↪  register with the result).
    // If const_opcode is true and the expression is a constant
     ↪  expression, it will move it
    // to a new register. Otherwise, it will just add the constant and
     ↪  the constant index.
    // suggested_register will use that register as the result.
    register_t compile(
        // The expression to compile
        const std::shared_ptr<Expression> &rule,
        // If the load constant opcode shall be included
        const bool load_constant = true,
        // If a register is suggested as the output instead of a new
         ↪  one.
        const register_t *suggested_register = nullptr,
        // If the access is an assignment, the value of it is passed
         ↪  here.
        const std::shared_ptr<Expression> &access_assignment_value =
         ↪  std::shared_ptr<Expression>()
    );
    // Adds an opcode to the program.
    void add_opcodes(const std::vector<opcode_t> &opcodes);
    // Adds a constant to the constant pool of the current frame
    // and return it's position.
    size_t add_constant(const Value &value);
    // Creates a constant list or dictionary from the given list
     ↪  expression.
```

```
    // Take into consideration that the list MUST be checked if
    // it's constant using is_constant() function.
    void constant_list(const std::shared_ptr<List> &list, Value &dest);
    void constant_dict(const std::shared_ptr<Dictionary> &dict, Value
    ↪  &dest);
    // Determines if an expression is constant (is in the constant
    ↪  pool).
    bool is_constant(const std::shared_ptr<Expression> &expression);
    // Sets a file flag at the current code location.
    std::shared_ptr<const std::string> current_file;
    void set_file(const std::shared_ptr<const std::string> &file);
    // Sets a line flag at the current code location.
    line_t current_line = 0;
    void set_line(const line_t line);
    // Sets a column flag at the current code location.
    column_t current_column = 0;
    void set_column(const column_t column);
    // Get a variable from the block stack and
    // return a pair containing the variable and a boolean
    // to indicate if it's global or not.
    std::pair<BlockVariableType *, bool> get_variable(const std::string
    ↪  &name);
    BlockClassType *get_class(const std::string &name);
    public:
        // Compile an input source and returns the main global
        ↪  register.
        register_t compile(const char *file);
        Compiler(const std::shared_ptr<Program> &program)
            : program(program) {}
};
```

Listing 33: Compiler class

8. Virtual Machine

The virtual machine is responsible for the bytecode execution. This layer must be carefully built since it may introduce significant bottlenecks to the program execution speed. If the virtual machine is carefully built and the bytecode is significantly improved, the resulting speed can be benefited. This thesis implements a register machine to interpret the code. A stack machine could also be used but it's less efficient as described in [Yun05].

To design such a system, consideration needs to be taken at instruction dispatching. There are different ways to dispatch instructions in a virtual machine as seen in [Yun05]. The most well known and used are:

- *Switch dispatch*: The switch dispatch uses a (C) switch statement to decide what instruction to execute.
- *Threaded dispatch*: The instruction dispatch is done by taking advantage of first-class labels. This is more efficient than a switch dispatch, but it's not part of ANSI C. There is still the possibility of using assembly to write part of the interpreter.

This thesis is designed with a C switch to be compliant with ANSI C. However, it's good to note that is not the most efficient way to dispatch instructions.

The virtual machine is responsible for storing the program. The virtual machine makes use of a program counter to decide the instruction to execute (the program counter is implemented as a pointer to the opcode). The virtual machine must also take care of the call stack and the shared value stack discussed in Secció 7.1.

The same way as the program counter points to the current instruction, two pointers are used to point to the current call frame and the top of the value stack.

8.1. Virtual Machine Class

The implementation of the virtual machine class is rather easy. The complicated stuff (memory, program structure, and frames) are already done by the code generator, and therefore the only thing left is to implement the opcodes.

The implementation can be seen on Listing 34

```
#define MAX_FRAMES 1024
#define STACK_SIZE 1024
class VirtualMachine
{
    // Stores the program to be executed.
    std::shared_ptr<Program> program = std::make_shared<Program>();
    // Virtual machine program counter.
    opcode_t *program_counter;
    // Stores the virtual machine stack (used to push function
    ↪  arguments / get return values)
    Value stack[STACK_SIZE];
    // Indicates the top of the stack.
    Value *top_stack = this->stack;
    // Stores the current frame stack.
    Frame frames[MAX_FRAMES];
    // Indicates the top level frame.
    Frame *active_frame = this->frames - 1; // It performs a
    ↪  pre-increment when a call is done.
    // Runs the virtual machine.
    void run();
    public:
        // It runs the virtual machine given a source input.
        void interpret(const char *file, const std::vector<std::string>
        ↪  &argv);
        // Resets the virtual machine program memories.
        void reset();
};
```

Listing 34: Virtual Machine Class

The two defines find on the top of Listing 34 sets the size of both stacks. They can be changed.

8.2. Opcode Operands Evaluation

When the virtual machine decides to execute an opcode, this normally comes with operands. Special care needs to be taken when getting the value of the operands. If you increment the program counter by one at a time it's fine if you do it by order but this normally requires setting some variables to store the values. It's also good to note that C++ have unspecified order of argument evaluation and that means that things like `a = 0; add(++a, ++a)` is not guaranteed to be `a = 0; add(1, 2)` and the compiler could chose to evaluate them in another order, for instance `a = 0; add(2, 1)`.

If using increments + variables, the number of additions performed in an arbitrary opcode with 3 operands are:

- 1 to go from the opcode to the 1st operand (program counter += 1).
- 1 to go from the 1st operand to the 2nd (program counter += 1).
- 1 to go from the 2nd operand to the 3rd (program counter += 1).
- 1 to fetch the next instruction (program counter += 1).

A solution can be provided by using the same amount of additions and solving problematic issues with increments and avoiding the temporary variables.

The solution requires the program counter to always point to the opcode. and further, perform additions to get the operands without modifying the program counter.

If using this method, the number of additions performed in an arbitrary opcode with 3 operands are:

- 1 to fetch the 1st operand (program counter + 1).
- 1 to fetch the 2nd operand (program counter + 2).
- 1 to fetch the 3rd operand (program counter + 3).
- 1 to increment the program counter to the next opcode (program counter += 4)

This method also reduces the number of assignments done, since only a single assignment is performed when the program counter needs to fetch the next instruction.

9. Application

The application layer is responsible for firing the application. In short, the job of this layer is to parse the command line arguments and run the virtual machine.

This layer exists so that the main file is not populated and the fact that this layer could evolve by providing different ways to run the program.

```
typedef enum : uint8_t {
    APPLICATION_FILE
} ApplicationType;
```

Listing 35: Application types

Listing 35 shows the supported application types. Additional types might be included in here, such as a prompt type or a string type. At the moment, the supported application is a file application, meaning the program must come from a file path. The application layer decides based on the command line arguments, the type of the application and performs the necessary actions accordingly.

This layer is also responsible for creating the `argv` parameter of the virtual machine. This parameter is a list of strings, so the C++ main arguments `argc` and `argv` are transformed into a vector of strings and further passed to the virtual machine so it can inject it into the main function call.

9.1. Application class

The application class is the implementation of the application layer, the implementation is very simple and can be seen in Listing 36

```
class Application
{
    // Defines the application type described above.
    ApplicationType application_type;
    // Stores the vritual machine used by the application.
    VirtualMachine virtual_machine;
    // Stores the file name if the application type requires it.
    std::string file_name;
    // Stores the command line arguments.
    std::vector<std::string> argv;
    // Run the application based on an input string.
    void string(const std::string &string);
    public:
        // The constructor determines the type of the application
        // based on the command line arguments.
        Application(int argc, char *argv[]);
        // Starts (runs) the application.
        int start();
};
```

Listing 36: Application class

10. Nuua Standard Library

This chapter deals with a very basic implementation of the Nuua standard library. This chapter is used as a working example of the path system mentioned in Secció 5.4.

Although this is just a minimal working example, it is enough to see how it works and be ready to get expanded as time goes on.

10.1. Utility module

The `utility` module as shown in Listing 37 implements basic utilities to work with different data types.

```
export fun int_min(a: int, b: int): int {
    if a < b => return a
    return b
}
export fun float_min(a: float, b: float): float {
    if a < b => return a
    return b
}
export fun int_max(a: int, b: int): int {
    if a > b => return a
    return b
}
export fun float_max(a: float, b: float): float {
    if a > b => return a
    return b
}
```

Listing 37: Utility module

10.2. List module

The `list` module as shown in Listing 38 implements basic functions to work with list types.

```
export fun list_int_sum(l: [int]): int {
    sum := 0
    for num in l => sum = sum + num
    return sum
}
export fun list_int_map(l: [int], f: (int -> int)) {
    for num, index in l => l[index] = f(num)
}
export fun list_float_sum(l: [float]): float {
    sum := 0.0
    for num in l => sum = sum + num
    return sum
}
export fun list_float_map(l: [float], f: (float -> float)) {
    for num, index in l => l[index] = f(num)
}
```

Listing 38: List module

10.3. Dict module

The `dict` module as shown in Listing 39 implements basic functions to work with dict types.

```
export fun dict_int_sum(e: {int}): int {
    sum := 0
    for num in e => sum = sum + num
    return sum
}
export fun dict_int_map(e: {int}, f: (int -> int)) {
    for num, index in e => e[index] = f(num)
}
export fun dict_float_sum(e: {float}): float {
    sum := 0.0
    for num in e => sum = sum + num
    return sum
}
export fun dict_float_map(e: {float}, f: (float -> float)) {
    for num, index in e => e[index] = f(num)
}
```

Listing 39: Dict module

11. Conclusions

> *"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

As a result of the hard work involved in the development of this thesis, the final result satisfies all the primary objectives set at Secció 1.1. The final result gets close to other programming languages in terms of project structure but also in terms of performance according to some initial tests done. However, benchmarking these systems is a very complex job and therefore it's not part of this thesis.

11.1. Further Evolution

Although all the objects are complete there are still many things that could be implemented into the language, the compiler, and the interpreter. Some of these things are very important and they are found in most mature programming languages. There are a set of features that are planned to be implemented into the language, the compiler, and the interpreter.

Some of this features are not simple changes or additions, in fact, some of the features provided here can probably take months to implement, therefore, they can't be included as part of this thesis and are just listed here for future reference.

- 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.
- Generic programming
- 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 `*=`.
- Add more optimizations to the compiler.

11.2. Code Repository

All the code involved in the development of this thesis can be found at the GitHub repository found in `https://github.com/nuua-io/Nuua`. Additionally, the website `https://nuua.io` may be used.

This repository includes a build system written in CMake to quickly build the project. More information about CMake can be found at their official website on `https://cmake.org/`.

As a reference, the main CMake script used to build the project is shown in Listing 40. Each layer in the system architecture have its own build script (2-3 lines each) that can be found on the repository.

```
# Required project information.
cmake_minimum_required (VERSION 3.9)
project (Nuua)

# IPO support
include (CheckIPOSupported)

# Set the output directory
set (OUTPUT_DIR ${CMAKE_BINARY_DIR}/bin)
set (LIBRARY_DIR ${CMAKE_BINARY_DIR}/lib)
set (STANDARD_LIRBARY_DIR ${CMAKE_SOURCE_DIR}/Lib)

# Set the default build mode to release if not set.
if (NOT CMAKE_BUILD_TYPE)
  set (CMAKE_BUILD_TYPE Release)
endif ()

# Set the C++ standard to use.
set (CMAKE_CXX_STANDARD 20)
set (CMAKE_CXX_STANDARD_REQUIRED ON)

# Enable interprocedural optimizations
check_ipo_supported (RESULT ipo_supported)
```

```
if (ipo_supported)
    set (CMAKE_INTERPROCEDURAL_OPTIMIZATION true)
endif ()

# Set the C++ general flags and flags for debug and release.
if (CMAKE_COMPILER_IS_GNUCXX)
    set (CMAKE_CXX_FLAGS "-Wall -Wextra -pipe")
    set (CMAKE_CXX_FLAGS_DEBUG "-g -DDEBUG=true")
    set (CMAKE_CXX_FLAGS_RELEASE "-Ofast -s -flto -frename-registers
    ↪  -fopenmp -D_GLIBCXX_PARALLEL -DDEBUG=false")
endif (CMAKE_COMPILER_IS_GNUCXX)

# Set the output directories.
set (CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${LIBRARY_DIR}/$<0:>)
set (CMAKE_LIBRARY_OUTPUT_DIRECTORY ${LIBRARY_DIR}/$<0:>)
set (CMAKE_RUNTIME_OUTPUT_DIRECTORY ${OUTPUT_DIR}/$<0:>)

# Add the subdirectories where to look.
add_subdirectory (Logger)
add_subdirectory (Lexer)
add_subdirectory (Parser)
add_subdirectory (Analyzer)
add_subdirectory (Compiler)
add_subdirectory (Virtual-Machine)
add_subdirectory (Application)

# Setup the final executable.
add_executable (nuua nuua.cpp resources/icon.rc)
target_link_libraries (nuua Application)

# Copy the standard library to the exetuable path.
add_custom_command (TARGET nuua POST_BUILD COMMAND ${CMAKE_COMMAND} -E
↪  copy_directory ${STANDARD_LIRBARY_DIR} ${OUTPUT_DIR}/Lib)
```

Listing 40: Main CMake script

Part II.

Appendices

A. Nuua Language Specification

> *"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*

The Nuua language specification includes all the information about the language grammar, the scopes, the entry point, comments, the data types it has and additionally, the supported statements and expressions.

A.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 `"{" ... "}"`.

A.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.

A.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"
    | delete_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 Taula A.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.

```
delete_statement
    : "delete" expression
    ;
```

A `delete` statement uses the `delete` keyword and an expression willing to delete.

```
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.

A.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 A.1.: Nuua operator precedence from highest to lowest with the associativity

A.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 Subsecció A.5.11.

A.1.5. Escaped Characters

There are some characters that can be escaped in Nuua, for example when the value `'"'` wants to be used inside a string where the delimiters are also `'"'`. To escape a character, the prefix \ needs to be used followed by the character willing to escape. The following list gives a view of all the available characters that allow beeing escaped found in Nuua.

(i) \
(ii) `'`
(iii) `"`
(iv) `n`
(v) `t`
(vi) `r`
(vii) `b`
(viii) `f`
(ix) `v`
(x) `0`

A.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 Taula A.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 A.2.: Nuua statements with scope blocks

A.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 exist 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]) {
    // ...
}
```

A.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.

A.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:

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

A.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.

A.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
```

A.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 Secció 11.1. It can store any text that's surrounded by `'"'`.

A.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.

A.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.

A.4.7. Functions

The only way to define a function is by using the "fun" keyword as noted in Secció A.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 Subsecció A.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)
```

A.4.8. Objects

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

A.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 Secció A.1.2.

A.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"
```

A.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
```

A.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 as explained in Subsecció A.6.7. 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
    }
}
```

A.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
}
```

A.5.5. Variable Declaration

Variable declarations are scoped to the block where they are declared as mentioned in Secció A.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
```

A.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, shorthands 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"
```

A.5.7. While Statement

A while statement is used to repeat a block of code while a certain expression (known as a 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
    a = a + 1
}
```

A.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 A.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 Taula A.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
```

A.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
}
```

A.5.10. Delete Statement

The delete statement is used to delete a specific index of a target. The target must be a Nuua iterator as shown in Taula A.3. The rule must be an access expression to indicate the exact element to delete.

Caveats

- The target must be a Nuua iterator and the rule must be an access.

Examples

A simple delete statement.

```
a: [int] = [1, 2, 3]
delete a[1]
// a is now [1, 3]
```

A.5.11. 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 Secció 11.1.

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
}
```

A.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 Secció A.1.2.

A.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
```

A.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
```

A.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
```

A.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 7, 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 Secció 11.1.

Examples

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

A.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]
```

A.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}
```

A.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}
```

A.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
```

A.6.9. Access Expression

The access expression is used to access an inner value of another expression. 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 Taula A.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"
```

A.6.10. Slice Expression

Slices act 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 A.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"
```

A.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 being treated as an expression, only being 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
```

A.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"
```

A.6.13. Unary Expression

The unary expression is an expression that has an operation attached to it. Prefix notation is used to specify the operation to the expression. The list of available unary operations is found in Taula A.5.

Operation symbol	Operation
`-`	Unary negation of the expression on the right. The right expression must be an `int`, a `float` or a `bool`
`+`	Positive version of the `-`. Can be used to cast a `bool` to an `int`. The right expression must be an `int`, a `float` or a `bool`
`!`	Logical negation, the right expression must be a `bool`.

Table A.5.: Unary operations

Caveats

- A positive sign can be used to cast a `bool` to an int, although conventional casting can be used.

Examples

Example unary operations

```
-10 // -10
-(10 + 2) // -12
!false // true
+true // 1
```

A.6.14. Cast Expression

The cast expression is used to perform type conversion on values. The casts are checked prior to execution, and the supported list of casts is show in Taula A.6.

Input type	Cast type	Notes
`int`	`float`	Possible data loss.
`int`	`bool`	`false` when the value is `0`, otherwise `true`.
`int`	`string`	Conversion to a string representation.
`float`	`int`	Decimals are truncated. Possible data loss.
`float`	`bool`	`false` when the value is `0.0`, otherwise `true`.
`float`	`string`	Conversion to a string representation.
`bool`	`int`	`0` when the value is `false`, otherwise `1`.
`bool`	`float`	`0.0` when the value is `false`, otherwise `1.0`.
`bool`	`string`	Conversion to a string representation.
`string`	`bool`	`false` when the string length is `0`, otherwise `true`
`string`	`int`	Returns the string length.
`[T]`	`string`	Conversion to a string representation.
`[T]`	`bool`	`false` when the list length is `0`, otherwise `true`
`[T]`	`int`	Returns the list length.
`{T}`	`string`	Conversion to a string representation.
`{T}`	`bool`	`false` when the dictionary length is `0`, otherwise `true`
`{T}`	`int`	Returns the dictionary length.

Table A.6.: Nuua casts

Examples

Example casting operations.

```
10 as float // 10.0
25.8 as int // 25
false as string // "false"
[1, 2, 3] as string // "[1, 2, 3]"
[20, 30] as int // 2
```

A.6.15. Binary Expression

The binary expression is used to perform an operation that require two values to be done with the exception of the logical expression that is treated differently. The binary operations that can be performed are shown in Taula A.7, Taula A.8, Taula A.9 and Taula A.10.

Left type	Operation	Right type	Result type	Notes
`int`	`+`	`int`	`int`	`int` addition.
`float`	`+`	`float`	`float`	`float` addition.
`string`	`+`	`string`	`string`	`string` concatenation.
`bool`	`+`	`bool`	`int`	`bool` addition.
`[T]`	`+`	`[T]`	`[T]`	`[T]` concatenation.
`{T}`	`+`	`{T}`	`{T}`	`{T}` concatenation.

(i) Addition

Left type	Operation	Right type	Result type	Notes
`int`	`-`	`int`	`int`	`int` subtraction.
`float`	`-`	`float`	`float`	`float` subtraction.
`bool`	`-`	`bool`	`int`	`bool` subtraction.

(ii) Substraction

Table A.7.: Nuua additive binary operations

Left type	Operation	Right type	Result type	Notes
`int`	`*`	`int`	`int`	`int` multiplication.
`float`	`*`	`float`	`float`	`float` multiplication.
`bool`	`*`	`bool`	`int`	`bool` multiplication.
`int`	`*`	`string`	`string`	`string` repetition.
`string`	`*`	`int`	`string`	`string` repetition.
`int`	`*`	`[T]`	`[T]`	`[T]` repetition.
`[T]`	`*`	`int`	`[T]`	`[T]` repetition.

(i) Multiplication

Left type	Operation	Right type	Result type	Notes
`int`	`/`	`int`	`float`	`int` division.
`float`	`/`	`float`	`float`	`float` division.
`string`	`/`	`int`	`[string]`	`string` division.
`[T]`	`/`	`int`	`[[T]]`	`[T]` division.

(ii) Division

Table A.8.: Nuua multiplicative binary operations

Left type	Operation	Right type	Result type	Notes
`int`	`==`	`int`	`bool`	`int` equality.
`float`	`==`	`float`	`bool`	`float` equality.
`string`	`==`	`string`	`bool`	`string` equality.
`bool`	`==`	`bool`	`bool`	`bool` equality.
`[T]`	`==`	`[T]`	`bool`	`[T]` equality.
`{T}`	`==`	`{T}`	`bool`	`{T}` equality.

(i) Equality

Left type	Operation	Right type	Result type	Notes
`int`	`!=`	`int`	`bool`	`int` inequality.
`float`	`!=`	`float`	`bool`	`float` inequality.
`string`	`!=`	`string`	`bool`	`string` inequality.
`bool`	`!=`	`bool`	`bool`	`bool` inequality.
`[T]`	`!=`	`[T]`	`bool`	`[T]` inequality.
`{T}`	`!=`	`{T}`	`bool`	`{T}` inequality.

(ii) Inequality

Table A.9.: Nuua relational equality binary operations

Left type	Operation	Right type	Result type	Notes
`int`	`>`	`int`	`bool`	`int` higher than.
`float`	`>`	`float`	`bool`	`float` higher than.
`string`	`>`	`string`	`bool`	`string` higher than.
`bool`	`>`	`bool`	`bool`	`bool` higher than.

(i) Higher than

Left type	Operation	Right type	Result type	Notes
`int`	`>=`	`int`	`bool`	`int` higher than or equal.
`float`	`>=`	`float`	`bool`	`float` higher than or equal.
`string`	`>=`	`string`	`bool`	`string` higher than or equal.
`bool`	`>=`	`bool`	`bool`	`bool` higher than or equal.

(ii) Higher than or equal

Left type	Operation	Right type	Result type	Notes
`int`	`<`	`int`	`bool`	`int` lower than.
`float`	`<`	`float`	`bool`	`float` lower than.
`string`	`<`	`string`	`bool`	`string` lower than.
`bool`	`<`	`bool`	`bool`	`bool` lower than.

(iii) Lower than

Left type	Operation	Right type	Result type	Notes
`int`	`<=`	`int`	`bool`	`int` lower than or equal.
`float`	`<=`	`float`	`bool`	`float` lower than or equal.
`string`	`<=`	`string`	`bool`	`string` lower than or equal.
`bool`	`<=`	`bool`	`bool`	`bool` lower than or equal.

(iv) Lower than or equal

Table A.10.: Nuua relational ordering binary operations

Examples

Example binary operations.

```
10 + 20 // 30
20 < 50 // true
"sample" == "sample"
"abc" * 2 // "abcabc"
"erik" / 4 // ["e", "r", "i", "k"]
```

A.6.16. Logical Expression

The logical expression is used to perform a logical operation on two values. The logical operations supported are the `and` and the `or` operations that perform a logical conjunction and a logical disjunction respectively.

Caveats

- The values on both sides of the logical expression are required to be of type `bool`. Explicit casting is needed.
- If one or both values are not of type bool, an error is thrown prior to execution.

Examples

Example logical `and`.

```
true and false // false
true and true // true
```

Example logical `or`.

```
true or false // true
true or true // true
```

A.6.17. Range Expression

The range expression is used to create a list ranging from a `start` index to an `end` index. This expression allows for quick lists to be created and ready to worth with a single line. The range can be inclusive or exclusive depending on the number of dots found in the expression.

Caveats

- The `start` and the `end` index must be of type `int`. If not, an error is thrown prior to execution.
- Inclusive ranges use three dots, while an exclusive range uses two dots.

Examples

Example ranges.

```
0..3 // [0, 1, 2]
-2...2 // [-2, -1, 0, -1, -2]
```

A.6.18. Assignment Expression

The assignment expression is used to assign a value to a target. Target may be a variable, an access expression or a property expression. The left side (target) is then assigned the expression on the right side (value). The assignment expression returns the assignment value.

Caveats

- The types of the target and the value must match to perform an assignment. If not, an error is thrown prior to execution.

Examples

Example variable assignment.

```
test: int
test = 20 // 20
```

Example access assignment.

```
l := [1, 2, 3]
l[1] = 4 // 4
// The value of l will now be [1, 4, 3]
d := {"a": 10, "b": 20}
d["a"] = 5 // 5
// The value of d will now be {"a": 5, "b": 20}
```

Example property assignment.

```
class Person {
    name: string
}
p := Person!{name: "Erik"}
p.name = "User" // "User"
// The value of p will now be Person!{name: "User"}
```

A.7. Comments

Comments in Nuua can be written by using a double backslash (\\) followed by the comment text. The comment text lasts till a `\n` character is found. Therefore, multi-line comments can be done by manually writing the double backslash on each different 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 backslash, it proceeds to discard the whole line.

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

B. Nuua Abstract Syntax Tree Nodes

A node in the AST is represented by the class found Listing 41

```
class Node
{
    public:
        // Stores the real rule of the node.
        const Rule rule;
        // Stores the file where it's found.
        std::shared_ptr<const std::string> file;
        // Stores the line where it's found.
        line_t line;
        // Stores the column where it's found.
        column_t column;
        // Constructor.
        Node(
            const Rule r,
            const std::shared_ptr<const std::string> &f,
            const line_t l, const column_t c
        ) : rule(r), file(f), line(l), column(c) {};
};
```

Listing 41: Node class

However, nodes are categorized into two sections. Nodes can be either statements or expressions. Therefore, two classes are used as nodes instead as shown in Listing 42.

```
class Expression : public Node
{
    public:
        explicit Expression(const Node &node)
            : Node(node) {};
};
class Statement : public Node
{
    public:
        explicit Statement(const Node &node)
            : Node(node) {};
};
```

Listing 42: Expression and Statement classes

These two classes extend the node class so they inherit all the attributes from it.

To work with all the nodes in a more effective way, the definition in Listing 43 is added.

```
#define NODE_PROPS std::shared_ptr<const std::string> &file, const
↪  line_t line, const column_t column
```

Listing 43: Node properties definition

B.1. Integer Node

The integer node only needs the value that corresponds to it. In this case a C++ `int64_t` type is used as shown in Listing 44.

```
class Integer : public Expression
{
    public:
        int64_t value;
        Integer(NODE_PROPS, const int64_t v)
            : Expression({ RULE_INTEGER, file, line, column }),
              value(v) {};
};
```

Listing 44: Integer Node

B.2. Float Node

The float node only needs the value that corresponds to it. In this case a C++ `double` type is used as shown in Listing 45.

```
class Float : public Expression
{
    public:
        double value;
        Float(NODE_PROPS, const double v)
            : Expression({ RULE_FLOAT, file, line, column }),
              value(v) {};
};
```

Listing 45: Float Node

B.3. String Node

The string node only needs the value that corresponds to it. In this case a C++ `std::string` type is used as shown in Listing 46.

```
class String : public Expression
{
    public:
        std::string value;
        String(NODE_PROPS, const std::string &v)
            : Expression({ RULE_STRING, file, line, column }),
              value(v) {};
};
```

Listing 46: String Node

B.4. Boolean Node

The boolean node only needs the value that corresponds to it. In this case a C++ `bool` type is used as shown in Listing 47.

```
class Boolean : public Expression
{
    public:
        bool value;
        Boolean(NODE_PROPS, const bool v)
            : Expression({ RULE_BOOLEAN, file, line, column }),
              value(v) {};
};
```

Listing 47: Boolean Node

B.5. List Node

The list node needs the value that corresponds to it and a type parameter that's filled in by the semantic analyzer. In this case a C++ `std::vector<std::shared_ptr<Expression>>` type is used as shown in Listing 48. The vector contains other expressions found inside the list.

```
class List : public Expression
{
    public:
        std::vector<std::shared_ptr<Expression>> value;
        // Stores the list type since it's complex to analyze later.
        std::shared_ptr<Type> type;
        List(
            NODE_PROPS,
            const std::vector<std::shared_ptr<Expression>> &v
        ) : Expression({ RULE_LIST, file, line, column }),
            value(v) {};
};
```

Listing 48: List Node

B.6. Dictionary Node

The dictionary node needs the key-value pairs of strings and expressions and the key order. In this case a C++ `std::unordered_map<std::string, std::shared_ptr<Expression>>` type is used as shown in Listing 49. The type is also added and used by the semantic analyzer.

```
class Dictionary : public Expression
{
    public:
        std::unordered_map<std::string, std::shared_ptr<Expression>>
        ↪  value;
        std::vector<std::string> key_order;
        // Stores the dict type since it's complex to analyze later.
        std::shared_ptr<Type> type;
        Dictionary(
            NODE_PROPS,
            const std::unordered_map<std::string,
            ↪  std::shared_ptr<Expression>> &v,
            const std::vector<std::string> &ko
        ) : Expression({ RULE_DICTIONARY, file, line, column }),
            value(std::move(v)),
            key_order(std::move(ko)) {};
};
```

Listing 49: Dictionary Node

B.7. Group Node

The group node consists of a single expression as seen in Listing 50.

```
class Group : public Expression
{
    public:
        std::shared_ptr<Expression> expression;
        Group(NODE_PROPS, const std::shared_ptr<Expression> &v)
            : Expression({ RULE_GROUP, file, line, column }),
              expression(std::move(v)) {};
};
```

Listing 50: Group Node

B.8. Unary Node

The unary node consists of the token operation to do and the right expression. It also stores the type of unary operation as seen in Subsecció A.6.13. Listing 51 shows the representation of the unary node.

```
class Unary : public Expression
{
    public:
        Token op;
        std::shared_ptr<Expression> right;
        // Determines what type of unary operation
        // will be performed, no need to store a Type.
        UnaryType type = (UnaryType) NULL;
        Unary(NODE_PROPS, const Token &o, const
        ↪  std::shared_ptr<Expression> &r)
            : Expression({ RULE_UNARY, file, line, column }),
              op(o),
              right(std::move(r)) {};
};
```

Listing 51: Unary Node

B.9. Binary Node

The binary node consists of the token operation to do and the left and right expression. It also stores the type of binary operation as seen in Subsecció A.6.15. Listing 52 shows the representation of the binary node.

```
class Binary : public Expression
{
    public:
        std::shared_ptr<Expression> left;
        Token op;
        std::shared_ptr<Expression> right;
        // Determines what type of binary operation will be performed.
        BinaryType type = (BinaryType) NULL;
        Binary(
            NODE_PROPS,
            const std::shared_ptr<Expression> &l,
            const Token &o,
            const std::shared_ptr<Expression> &r
        ) : Expression({ RULE_BINARY, file, line, column }),
            left(std::move(l)),
            op(o),
            right(std::move(r)) {};
};
```

Listing 52: Binary Node

B.10. Variable Node

The variable node consists of the name of the variable. Listing 53 shows the representation of the variable node.

```
class Variable : public Expression
{
    public:
        std::string name;
        Variable(NODE_PROPS, const std::string &n)
            : Expression({ RULE_VARIABLE, file, line, column }),
              name(n) {};
};
```

Listing 53: Variable Node

B.11. Assign Node

The assign node consists of the target (the expression to be assigned) and the value of the assignment. Listing 54 shows the representation of the assign node. It also stores a boolean that determines if the assignment is done to an access target (an inner value of a Nuua iterator). This boolean is used by the semantic analyzer.

```
class Assign : public Expression
{
    public:
        std::shared_ptr<Expression> target;
        std::shared_ptr<Expression> value;
        bool is_access = false;
        Assign(
            NODE_PROPS,
            const std::shared_ptr<Expression> &t,
            const std::shared_ptr<Expression> &v
        ) : Expression({ RULE_ASSIGN, file, line, column }),
            target(std::move(t)),
            value(std::move(v)) {};
};
```

Listing 54: Assign Node

B.12. Logical Node

The logical node consists the logical operation (`or` or `and`) and the left and right expressions. Listing 55 shows the representation of the logical node.

```
class Logical : public Expression
{
    public:
        std::shared_ptr<Expression> left;
        Token op;
        std::shared_ptr<Expression> right;
        Logical(
            NODE_PROPS,
            const std::shared_ptr<Expression> &l,
            const Token &o,
            const std::shared_ptr<Expression> &r
        ) : Expression({ RULE_LOGICAL, file, line, column }),
            left(std::move(l)),
            op(o),
            right(std::move(r)) {};
};
```

Listing 55: Logical Node

B.13. Call Node

The call node consists of the callee target and of the arguments provided to perform the call. Listing 56 shows the representation of the call node. It also stores if the call has a return depending on the callee. This helps the semantic analyzer determine where and when this call can be used since this expression may have no value.

```
class Call : public Expression
{
    public:
        std::shared_ptr<Expression> target;
        std::vector<std::shared_ptr<Expression>> arguments;
        // Determines if the call target returns a value or not.
        bool has_return = false;
        Call(
            NODE_PROPS,
            const std::shared_ptr<Expression> &t,
            const std::vector<std::shared_ptr<Expression>> &a
        ) : Expression({ RULE_CALL, file, line, column }),
            target(std::move(t)),
            arguments(a) {};
};
```

Listing 56: Call Node

B.14. Access Node

The access node consists of the target to access and the index for it. Listing 57 shows the representation of the access node. The type represents the type of access (the type of iterator to access).

```
class Access : public Expression
{
    public:
        std::shared_ptr<Expression> target;
        std::shared_ptr<Expression> index;
        AccessType type = (AccessType) NULL;
        Access(
            NODE_PROPS,
            const std::shared_ptr<Expression> &t,
            const std::shared_ptr<Expression> &i
        ) : Expression({ RULE_ACCESS, file, line, column }),
            target(std::move(t)),
            index(std::move(i)) {};
};
```

Listing 57: Access Node

B.15. Cast Node

The cast node consists of the expression to cast and the type to cast it to. Listing 58 shows the representation of the access node. The type of cast as shown in Subsecció A.6.14 is also stored and further used by the analyzer.

```
class Cast : public Expression
{
    public:
        std::shared_ptr<Expression> expression;
        std::shared_ptr<Type> type;
        CastType cast_type = (CastType) NULL;
        Cast(
            NODE_PROPS,
            const std::shared_ptr<Expression> &e,
            std::shared_ptr<Type> &t
        ) : Expression({ RULE_CAST, file, line, column }),
            expression(std::move(e)),
            type(std::move(t)) {}
};
```

Listing 58: Cast Node

B.16. Slice Node

The slice node consists of the expressions of the target and the expressions of the start, end and step indexes. Listing 59 shows the representation of the access node. The additional `is_list` represents if the slice is done to a list or a string and it's used by the semantic analyzer.

```
class Slice : public Expression
{
    public:
        std::shared_ptr<Expression> target;
        std::shared_ptr<Expression> start;
        std::shared_ptr<Expression> end;
        std::shared_ptr<Expression> step;
        // Determines if it's a list or a string, used by Analyzer.
        bool is_list = false;
        Slice(
            NODE_PROPS,
            const std::shared_ptr<Expression> &t,
            const std::shared_ptr<Expression> &s,
            const std::shared_ptr<Expression> &e,
            const std::shared_ptr<Expression> &st
        ) : Expression({ RULE_SLICE, file, line, column }),
            target(std::move(t)),
            start(std::move(s)),
            end(std::move(e)),
            step(std::move(st)) {}
};
```

Listing 59: Slice Node

B.17. Range Node

The range node consists of the expressions of the start and end indexes. Listing 60 shows the representation of the access node. The inclusive fields indicate if the range is inclusive or exclusive.

```
class Range : public Expression
{
    public:
        std::shared_ptr<Expression> start;
        std::shared_ptr<Expression> end;
        bool inclusive;
        Range(
            NODE_PROPS,
            const std::shared_ptr<Expression> &s,
            const std::shared_ptr<Expression> &e,
            const bool i
        ) : Expression({ RULE_RANGE, file, line, column }),
            start(std::move(s)),
            end(std::move(e)),
            inclusive(i) {}
};
```

Listing 60: Range Node

B.18. Delete Node

The delete node consists of the target expression to delete. Listing 60 shows the representation of the access node.

```
class Delete : public Expression
{
    public:
        std::shared_ptr<Expression> target;
        Delete(
            NODE_PROPS,
            const std::shared_ptr<Expression> &t
        ) : Expression({ RULE_DELETE, file, line, column }),
            target(std::move(t)) {}
};
```

Listing 61: Delete Node

B.19. Function Value Node

The function value node represents a value to build a function. The node must know the function name, the parameters of it, the return type and the body. Additionally,

the scope block is also stored for further use. Listing 62 shows the representation of the FunctionValue node.

```
class FunctionValue : public Expression
{
    public:
        std::string name;
        std::vector<std::shared_ptr<Declaration>> parameters;
        std::shared_ptr<Type> return_type;
        std::vector<std::shared_ptr<Statement>> body;
        std::shared_ptr<Block> block;
        FunctionValue(
            NODE_PROPS,
            const std::string &n,
            const std::vector<std::shared_ptr<Declaration>> &p,
            std::shared_ptr<Type> &rt,
            const std::vector<std::shared_ptr<Statement>> &b
        ) : Expression({ RULE_FUNCTION, file, line, column }),
            name(n), parameters(p),
            return_type(std::move(rt)),
            body(b) {}
};
```

Listing 62: FunctionValue Node

B.20. Object Node

The object node represents an object expression. The required pieces of information are the class name and the arguments to initialize the instance (as a key-value pair). Additionally, the scope block is also stored for further use. Listing 63 shows the representation of the Object node.

```
class Object : public Expression
{
    public:
        std::string name;
        std::unordered_map<std::string, std::shared_ptr<Expression>>
        ↪  arguments;
        Object(
            NODE_PROPS,
            const std::string &n,
            const std::unordered_map<std::string,
            ↪  std::shared_ptr<Expression>> &a
        ) : Expression({ RULE_OBJECT, file, line, column }),
            name(n),
            arguments(a) {}
};
```

Listing 63: Object Node

B.21. Property Node

The property node represents an access to an object property. The information needed is the expression to access and the name of the property. Listing 64 shows the representation of the Property node.

```
class Property : public Expression
{
    public:
        std::shared_ptr<Expression> object;
        std::string name;
        Property(
            NODE_PROPS,
            const std::shared_ptr<Expression> &o,
            const std::string &n
        ) : Expression({ RULE_PROPERTY, file, line, column }),
            object(o),
            name(n) {}
};
```

Listing 64: Property Node

B.22. Print Node

The print node only requires the expression to print. Listing 65 shows the representation of the Print node.

```
class Print : public Statement
{
    public:
        std::shared_ptr<Expression> expression;
        Print(
            NODE_PROPS,
            const std::shared_ptr<Expression> &e
        ) : Statement({ RULE_PRINT, file, line, column }),
            expression(std::move(e)) {}
};
```

Listing 65: Print Node

B.23. Expression Statement Node

The expression statement node is used as a wrapper to treat an expression as a valid statement. Listing 66 shows the representation of the ExpressionStatement node.

```
class ExpressionStatement : public Statement
{
    public:
        std::shared_ptr<Expression> expression;
        ExpressionStatement(
            NODE_PROPS,
            const std::shared_ptr<Expression> &e
        ) : Statement({ RULE_EXPRESSION_STATEMENT, file, line, column
        ↪ }),
            expression(std::move(e)) {}
};
```

Listing 66: ExpressionStatement Node

B.24. Declaration Node

The declaration node requires the name of the variable to declare and then a combination of the type and the initializer. The type may be empty if the initializer is set, and the

initializer may be empty if the type is set. Both of them may also be set but not empty. Listing 67 shows the representation of the Declaration node.

```
class Declaration : public Statement
{
    public:
        std::string name;
        std::shared_ptr<Type> type;
        std::shared_ptr<Expression> initializer;
        Declaration(
            NODE_PROPS,
            const std::string &n, std::shared_ptr<Type> &t,
            const std::shared_ptr<Expression> &i
        ) : Statement({ RULE_DECLARATION, file, line, column }),
            name(n),
            type(std::move(t)),
            initializer(std::move(i)) {};
};
```

Listing 67: Declaration Node

B.25. Return Node

The return node does not need an expression to be valid, but one may be provided. Listing 68 shows the representation of the Return node.

```
class Return : public Statement
{
    public:
        std::shared_ptr<Expression> value;
        Return(
            NODE_PROPS,
            const std::shared_ptr<Expression> &v =
            ↪  std::shared_ptr<Expression>()
        ) : Statement({ RULE_RETURN, file, line, column }),
            value(std::move(v)) {}
};
```

Listing 68: Return Node

B.26. If Node

The if node does need the condition expression and the then branch followed by the then block. Additionally, it may have an else branch and an else scope block. Listing 69 shows the representation of the If node.

```
class If : public Statement
{
    public:
        std::shared_ptr<Expression> condition;
        std::vector<std::shared_ptr<Statement>> then_branch;
        std::vector<std::shared_ptr<Statement>> else_branch;
        std::shared_ptr<Block> then_block, else_block;
        If(
            NODE_PROPS,
            const std::shared_ptr<Expression> &c,
            const std::vector<std::shared_ptr<Statement>> &tb,
            const std::vector<std::shared_ptr<Statement>> &eb
        ) : Statement({ RULE_IF, file, line, column }),
            condition(std::move(c)),
            then_branch(tb),
            else_branch(eb) {};
};
```

Listing 69: If Node

B.27. While Node

The while node does need the condition expression and the body of it. It also stores the scope block for further use. Listing 70 shows the representation of the While node.

```
class While : public Statement
{
    public:
        std::shared_ptr<Expression> condition;
        std::vector<std::shared_ptr<Statement>> body;
        std::shared_ptr<Block> block;
        While(
            NODE_PROPS,
            const std::shared_ptr<Expression> &c,
            const std::vector<std::shared_ptr<Statement>> &b
        ) : Statement({ RULE_WHILE, file, line, column }),
            condition(std::move(c)),
            body(b) {};
};
```

Listing 70: While Node

B.28. For Node

The for node does require the iterator and the variable to store the loop value. Additionally, it may have the index in case it's needed as part of the loop. It also stores the scope block for further use. Listing 71 shows the representation of the For node.

```
class For : public Statement
{
    public:
        std::string variable;
        std::string index;
        std::shared_ptr<Expression> iterator;
        std::vector<std::shared_ptr<Statement>> body;
        std::shared_ptr<Block> block;
        // Stores the type of the iterator.
        std::shared_ptr<Type> type;
        For(
            NODE_PROPS,
            const std::string &v,
            const std::string &i,
            const std::shared_ptr<Expression> &it,
            const std::vector<std::shared_ptr<Statement>> &b
        ) : Statement({ RULE_FOR, file, line, column }),
            variable(v),
            index(i),
            iterator(std::move(it)), body(b) {}
};
```

Listing 71: For Node

B.29. Function Node

The function node wraps the `FunctionValue` node as a statement. Listing 72 shows the representation of the Function node.

```
class Function : public Statement
{
    public:
        std::shared_ptr<FunctionValue> value;
        Function(const std::shared_ptr<FunctionValue> &v)
            : Statement({ RULE_FUNCTION, v->file, v->line, v->column }),
              value(v) {}
};
```

Listing 72: Function Node

B.30. Use Node

The use node requires the module name to import and an optional target list. Targets can be empty and all exported targets will be imported. It also requires a property to store the code of the module (the AST of that particular module) and the module scope block of it. Listing 73 shows the representation of the Use node.

```
class Use : public Statement
{
    public:
        std::vector<std::string> targets;
        std::shared_ptr<const std::string> module;
        std::shared_ptr<std::vector<std::shared_ptr<Statement>>> code;
        std::shared_ptr<Block> block;
        Use(
            NODE_PROPS,
            const std::vector<std::string> &t,
            const std::shared_ptr<const std::string> &m
        ) : Statement({ RULE_USE, file, line, column }),
            targets(t),
            module(std::move(m)) {};
};
```

Listing 73: Use Node

B.31. Export Node

The export node just requires the statement that is exporting. Listing 74 shows the representation of the Export node.

```
class Export : public Statement
{
    public:
        std::shared_ptr<Statement> statement;
        Export(NODE_PROPS, const std::shared_ptr<Statement> &s)
            : Statement({ RULE_EXPORT, file, line, column }),
              statement(std::move(s)) {}
};
```

Listing 74: Export Node

B.32. Class Node

The class node needs the class name, the body of the class and it saves the scope block of the class for further use. Listing 75 shows the representation of the Class node.

```
class Class : public Statement
{
    public:
        std::string name;
        std::vector<std::shared_ptr<Statement>> body;
        std::shared_ptr<Block> block;
        Class(
            NODE_PROPS,
            const std::string &n,
            const std::vector<std::shared_ptr<Statement>> &b
        ) : Statement({ RULE_CLASS, file, line, column }),
            name(n),
            body(b) {}
};
```

Listing 75: Class Node

C. Nuua Virtual Machine Opcodes

All the Nuua virtual machine opcodes can be found in this list with a small comment on the right-hand side with the required operands of each opcode. If the opcode has a result or destination, this is always the *first* operand found in the opcode. For example the opcode `OP_MOVE RX RY` moves the value of `RY` to `RX`. As another example, the opcode `OP_ADD_INT RX RY RZ` performs the addition of values `RY + RZ` and stores the result at `RX`. The following table explains the different operand types found.

- *RX, RY, RX, R1, R2, R3*: Represents the value of a register (they are found in the top frame).
- *PX*: Represent the value of a register (they are found in the object)
- *A*: Represents a literal value (normaly used for jumping offsets)
- *G1*: Represents a global value (they are found in the global registers)
- *C1*: Represents a constant value (they are found in the constant pool)

```
typedef enum : uint8_t {
    // Others
    OP_EXIT, // EXIT - - -

    /* Register manipulation */
    // Moves (copy) the value from RY to RX.
    OP_MOVE, // MOVE RX RY
    // Load a constant to the register.
    OP_LOAD_C, // OP_LOAD_C RX C1
    // Load a global register to a local register.
    OP_LOAD_G, // OP_LOAD_G RX G1
    // Set a global value.
    OP_SET_G, // OP_SET_G G1 RX

    /* Stack manipulation */
    // Push a value to the shared stack.
    OP_PUSH, // PUSH RX
    // Push a constant to the shared stack.
    OP_PUSH_C, // PUSH C1
    // Pops a value from the shared stack.
```

```
    OP_POP, // POP RX

    /* String releated */
    // Get the char from a string given its index.
    OP_SGET, // SGET RX RY RZ
    // Set a char of the string given its index.
    OP_SSET, // SSET RX RY RZ
    // Delete a given char in a string given its index.
    OP_SDELETE, // SDELETE RX RY

    /* List releated */
    // Push a value to a list.
    OP_LPUSH, // LPUSH RX RY
    // Push a constant to a list.
    OP_LPUSH_C, // LPUSH RX C1
    // Pop a value from a list.
    OP_LPOP, // LPOP RX
    // Get the value of a list given it's index.
    OP_LGET, // LGET RX RY RZ
    // Set the value of the list in the given index.
    OP_LSET, // LSET RX RY RZ
    // Delete a value of the list given the index.
    OP_LDELETE, // LDELETE RX RY

    /* Dictionary releated */
    // Get dictionary key given the integer index and the key order.
    OP_DKEY, // DKEY RX RY RZ
    // Get a value from the dictionary given the key.
    OP_DGET, // DGET RX RY RZ
    // Set the value of the given index key.
    OP_DSET, // DSET RX RY RZ
    // Delete a given key-value pair.
    OP_DDELETE, // DDELETE RX RY

    /* Function releated */
    // Call a function.
    OP_CALL, // CALL RX
    // Return to previous frame.
    OP_RETURN, // RETURN

    /* Object releated */
    // Get object property.
    OP_LPROP, // PROP RX PX
    // Set object property.
```

```
    OP_SPROP, // PROP PX RX

    /* Casting Operations */

    // Value casting
    OP_CAST_INT_FLOAT, // CAST_INT_FLOAT RX RY
    OP_CAST_INT_BOOL, // CAST_INT_BOOL RX RY
    OP_CAST_INT_STRING, // CAST_INT_STRING RX RY
    OP_CAST_FLOAT_INT, // CAST_FLOAT_INT RX RY
    OP_CAST_FLOAT_BOOL, // CAST_FLOAT_BOOL RX RY
    OP_CAST_FLOAT_STRING, // CAST_FLOAT_STRING RX RY
    OP_CAST_BOOL_INT, // CAST_BOOL_INT RX RY
    OP_CAST_BOOL_FLOAT, // CAST_BOOL_FLOAT RX RY
    OP_CAST_BOOL_STRING, // CAST_BOOL_STRING RX RY
    OP_CAST_LIST_STRING, // CAST_LIST_STRING RX RY
    OP_CAST_LIST_BOOL, // CAST_LIST_BOOL RX RY
    OP_CAST_LIST_INT, // CAST_LIST_INT RX RY
    OP_CAST_DICT_STRING, // CAST_DICT_STRING RX RY
    OP_CAST_DICT_BOOL, // CAST_DICT_BOOL RX RY
    OP_CAST_DICT_INT, // CAST_DICT_INT RX RY
    OP_CAST_STRING_BOOL, // CAST_STRING_BOOL RX RY
    OP_CAST_STRING_INT, // CAST_STRING_INT RX RY

    /* Unary Operations */

    // Negation
    OP_NEG_BOOL, // NEG_BOOL RX RY
    // Minus operations
    OP_MINUS_INT, // MINUS_INT RX RY
    OP_MINUS_FLOAT, // MINUS_FLOAT RX RY
    OP_MINUS_BOOL, // MINUS_BOOL RX RY
    // Plus operations
    OP_PLUS_INT, // PLUS_INT RX RY
    OP_PLUS_FLOAT, // PLUS_FLOAT RX RY
    OP_PLUS_BOOL, // PLUS_BOOL RX RY

    /* Extra binary but unary */
    // Increment register integer
    OP_IINC, // INC RX
    // Decrement register integer
    OP_IDEC, // DEC RX

    /* Binary Operations */
```

```
    // Addition
    OP_ADD_INT, // ADD_INT RX RY RZ
    OP_ADD_FLOAT, // ADD_FLOAT RX RY RZ
    OP_ADD_STRING, // ADD_STRING RX RY RZ
    OP_ADD_BOOL, // ADD_BOOL RX RY RZ
    OP_ADD_LIST, // ADD_LIST RX RY RZ
    OP_ADD_DICT, // ADD_DICT RX RY RZ
    // Substraction
    OP_SUB_INT, // SUB_INT RX RY RZ
    OP_SUB_FLOAT, // SUB_FLOAT RX RY RZ
    OP_SUB_BOOL, // SUB_BOOL RX RY RZ
    // Multiplication
    OP_MUL_INT, // MUL_INT RX RY RZ
    OP_MUL_FLOAT, // MUL_FLOAT RX RY RZ
    OP_MUL_BOOL, // MUL_BOOL RX RY RZ
    OP_MUL_INT_STRING, // MUL_INT_STRING RX RY RZ
    OP_MUL_STRING_INT, // MUL_STRING_INT RX RY RZ
    OP_MUL_INT_LIST, // MUL_INT_LIST RX RY RZ
    OP_MUL_LIST_INT, // MUL_LIST_INT RX RY RZ
    // Division
    OP_DIV_INT, // DIV_INT RX RY RZ
    OP_DIV_FLOAT, // DIV_FLOAT RX RY RZ
    OP_DIV_STRING_INT, // DIV_STRING_INT RX RY RZ
    OP_DIV_LIST_INT, // DIV_LIST_INT RX RY RZ
    // Equality
    OP_EQ_INT, // EQ_INT RX RY RZ
    OP_EQ_FLOAT, // EQ_FLOAT RX RY RZ
    OP_EQ_STRING, // EQ_STRING RX RY RZ
    OP_EQ_BOOL, // EQ_BOOL RX RY RZ
    OP_EQ_LIST, // EQ_LIST RX RY RZ
    OP_EQ_DICT, // EQ_DICT RX RY RZ
    // Not Equality
    OP_NEQ_INT, // NEQ_INT RX RY RZ
    OP_NEQ_FLOAT, // NEQ_FLOAT RX RY RZ
    OP_NEQ_STRING, // NEQ_STRING RX RY RZ
    OP_NEQ_BOOL, // NEQ_BOOL RX RY RZ
    OP_NEQ_LIST, // NEQ_LIST RX RY RZ
    OP_NEQ_DICT, // NEQ_DICT RX RY RZ
    // Higher than
    OP_HT_INT, // HT_INT RX RY RZ
    OP_HT_FLOAT, // HT_FLOAT RX RY RZ
    OP_HT_STRING, // HT_STRING RX RY RZ
    OP_HT_BOOL, // HT_BOOL RX RY RZ
    // Higher than or equal to
```

```
OP_HTE_INT, // HTE_INT RX RY RZ
OP_HTE_FLOAT, // HTE_FLOAT RX RY RZ
OP_HTE_STRING, // HTE_STRING RX RY RZ
OP_HTE_BOOL, // HTE_BOOL RX RY RZ
// Lower than
OP_LT_INT, // LT_INT RX RY RZ
OP_LT_FLOAT, // LT_FLOAT RX RY RZ
OP_LT_STRING, // LT_STRING RX RY RZ
OP_LT_BOOL, // LT_BOOL RX RY RZ
// Lower than or equal to
OP_LTE_INT, // LTE_INT RX RY RZ
OP_LTE_FLOAT, // LTE_FLOAT RX RY RZ
OP_LTE_STRING, // LTE_STRING RX RY RZ
OP_LTE_BOOL, // LTE_BOOL RX RY RZ

/* Logical operations */
// Logical or
OP_OR, // OR RX RY RZ
// Logical and
OP_AND, // AND RX RY RZ

/* Control flow (All relative jumps) */
// Forward jump
OP_FJUMP, // FJUMP A
// Backward jump
OP_BJUMP, // BJUMP A
// Conditional forward jump
OP_CFJUMP, // FJUMP A RX
// Conditional backward jump
OP_CBJUMP, // BJUMP A RX
// Conditional forward negative jump
OP_CFNJUMP, // FNJUMP A RX
// Conditional backward negative jump
OP_CBNJUMP, // BNJUMP A RX

/* Slice and range */
// String slice.
OP_SSLICE, // SSLICE RX RY R1 R2 R3 (dest, target, start, end,
↪  step)
// String slice to the end.
OP_SSLICEE, // SSLICEE RX RY R1 R2 (dest, target, start, step)
// List slice.
OP_LSLICE, // LSLICE RX RY R1 R2 R3 (dest, target, start, end,
↪  step)
```

```
    // List slice to the end.
    OP_LSLICEE, // LSLICEE RX RY R1 R2 (dest, target, start, step)
    // Exclusive range.
    OP_RANGEE, // RANGEE RX R1 R2 (dest, start, end)
    // Inclusive range.
    OP_RANGEI, // RANGEI RX R1 R2 (dest, start, end)

    /* Utilities */
    // Print a register
    OP_PRINT, // PRINT RX
    // Print a constant
    OP_PRINT_C, // PRINT C1
} OpCode;
```

D. 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.

[Joh08] Maggie Johnson. "Top-Down Parsing". In: (2008). [Online; accessed 3-June-2019]. URL: https://suif.stanford.edu/dragonbook/lecture-notes/Stanford-CS143/07-Top-Down-Parsing.pdf.

[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.