The design of an experimental programming language and its translator

The Nuua Programming Language

Èrik Campobadal Forés

A bachelor thesis presented for the degree of
ICT Systems Engineer

EMIT- Department of Mining, Industrial and ICT Engineering
Polytechnic University of Catalonia
Catalonia, Spain
June 2019

Contents

List of Figures

Chapter 1

Introduction

Programming languages are used everyday by thousands of engineers and yet few of them understand how they work. A programming language is basically the syntax and grammar used as humans 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 and then a compiler needs to translate it into machine code for the computer to be able to execute it.

To design such a software system, it's vital to understand the theory behind it and learn about all the different steps involved to make a computer understand and execute a custom programming language.

1.1 Objectives

The objectives of this document are to define and implement a fully featured experimental programming language to learn and overcome the diferent challenges that are presented during the process. This document also explains and implement all the different aspects of the programming language ecosystem. This includes for instance the standard library, dependency management and a full featured webpage for the language. The experimental language is called **Nuua**.

To better summarize the objectives, this list should give a great overview to understand what this document achieves.

- Understand the theory behind all steps involved in common programming languages to be able to reproduce them and adapt it at the needs of the project.

- Define a non-ambiguous grammar for the Nuua Programming Language. This grammar is simple, elegan and yet following most common programming language's specifications to avoid confusion when learning it, allowing either mature programmers or newcomers to pick up the grammar quickly and efficiently.

- Choose an efficient programming language to code the language with. The language would need to be efficient, fast and yet able to code with it pretty fast to avoid wasting time. Among other options, the languages that satify the previous statement are low level programming languages like C, C++, D, Rust, Go, Crystal or Nim.

- Define a scalable project structure to allow the programming language to grow efficiently without creating a big ball of mud [1]. The software system that is chosen will be very important towards the scalability of the project.

- Create the programming language with a defined set of features found in most programming languages. Those include from basic datatypes to object oriented programming.

- Create a Standard Library with a common set of operations or protocols implemented (For example the HTTP protocol) and also common libraries to work with diferent data types in a higher level of abstraction (For example data collections).

- Create a cloud centralized dependency manager to allow the programs to require certain libraries with a certain version. Possible references are NPM (Node Package Manager), Composer (PHP package manager), Cargo (Rust package manager) or Go mods (Go modules).

- Develop the project's website to showcase the language, store the documentation and host the information for the dependency manager.

1.2 High level overview

To firstly understand all the complex system that this document implements, it's important to explain a few key concepts found in a typical compiler. As

[1]Big ball of mud (Wikipedia)

mentioned, the job of a compiler is to take an input source and translate it into another. Often, the compiler term is used to express a translation to a much diferent environment, that means that usually, the input is written in a high level language and further translated into a lower level of abstraction. As an example, GCC (GNU Compiler Collection) and it's implementation of the C programming language basically compiles the input code into assembly (and optimized if needed), then to an object file and finally linked to generate an executable. That said, the compiler job was to translate a high level programming language (C) into a low level programming language (Assembly). However, there are multiple types of compilers, and each one is used to implement different translation tasks.

- **Compiler**: Translates a high level language to a low level language in the same or diferent architecture. For example, GCC's C implementation
- **Cross-Compiler**: The translation is done to a specific (and probably diferent than the compiler's machine) architecture. For example, AVR-GCC compiles specifically for the AVR microcontroller architecture. Other closer examples may be found when compiling for 32 bits or 64 bits target architecture.
- **Bootstrap compiler**: A compiler that is built using the language it compiles to (The initial version is written in another language) For example, the official Go compiler is written in Go.
- **Decompiler**: Translates a low level language to a high level language in the same or diferent architecture. For example, the Boomerang decompiler.
- **Transpiler (Source-to-source compiler)**: Translates a language to a similar level of abstraction (usually between high level languages). For example, TypeScript (compiles to JavaScript).

Input language
Compiler
Output language

Figure 1.1: Compiler overview

1.3 Language runtime

A compiler job is just to translate, and then, it's up to the language implementation to run it. There are diferent ways to run or execute a programming language.

- **Compile to machine-language**: If the target language of the compiler is executable code, this can be run by the operating system and it will produce the nessesary outputs. Machine code have a big advantage and disadvantage. The executed code will be very fast but the executable is not portable among diferent architectured. This introduces platform-dependent builds that are very fast.

- **Interpret**: Interpreters are often a very diferent overview of this problem. Interpreters do not compile to machine language, and instead, it often compiles to something called Bytecode. Bytecode is essentially a portable machine-like language that is used to execute the language. The advantages are that is platform-independent and it's designed specifically with the language needs but one big disadvantage is that it's much slower to run than real machine-code. Interpreters can also make language execution dynamic allowing untyped languages that plays around with diferent variable types.

- **Just-in-time compiler**: Just-in-time compilers (often called JIT compilers) are an intermidiate aproach between the other two that tries to speed up the execution of interpreters by compiling chunks of code at a specific moment while the program run to speed up portions of the code that is often executed (for example functions that are called frequently). Esentially, 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 successive calls.

1.4 Structure

Nuua is structured as a layered system. This system is known to be robust and scalable at very low performance cost. It's very easy to understand and yet very powerful for software systems like this one. This systems are also used for other complex software systems, such as operating systems or complex protocols like TCP/IP.

By using a layered system, you can make sure each layer's function is correctly executed without messing with the whole system itself, creating a very impresive way to scale-up or upgrade existing parts of the system without damaging the others. However, a consistent API should in fact be established from the ground up to avoid breaking changes. If the API is maintained, the individual layers may be upgraded independently without the need of extra work. A layered system consists of diferent independent ordered layers that need to meet certain requirements. The requirements are mostly simple, each layer can only use the API given by the layer that's below it and it can only give an API to the one above.

Since this structure is ideal for Nuua, the decision to build it upon it was inevitable. Therefore, the layers Nuua have in it's system are exposed in Figure 1.2. An independent module called Logger is found on the left side of the figure. This module is in fact a simple logger used by all layers to output messages if needed (like errors messages). However, since all modules require the use of it, it can be used but not modified.

There's also three distinct sections found in the system, the back end, the middle end and the front end. I'll talk about them in a future section since it's not the purpose of the current one. However it's important to keep in mind they exist.

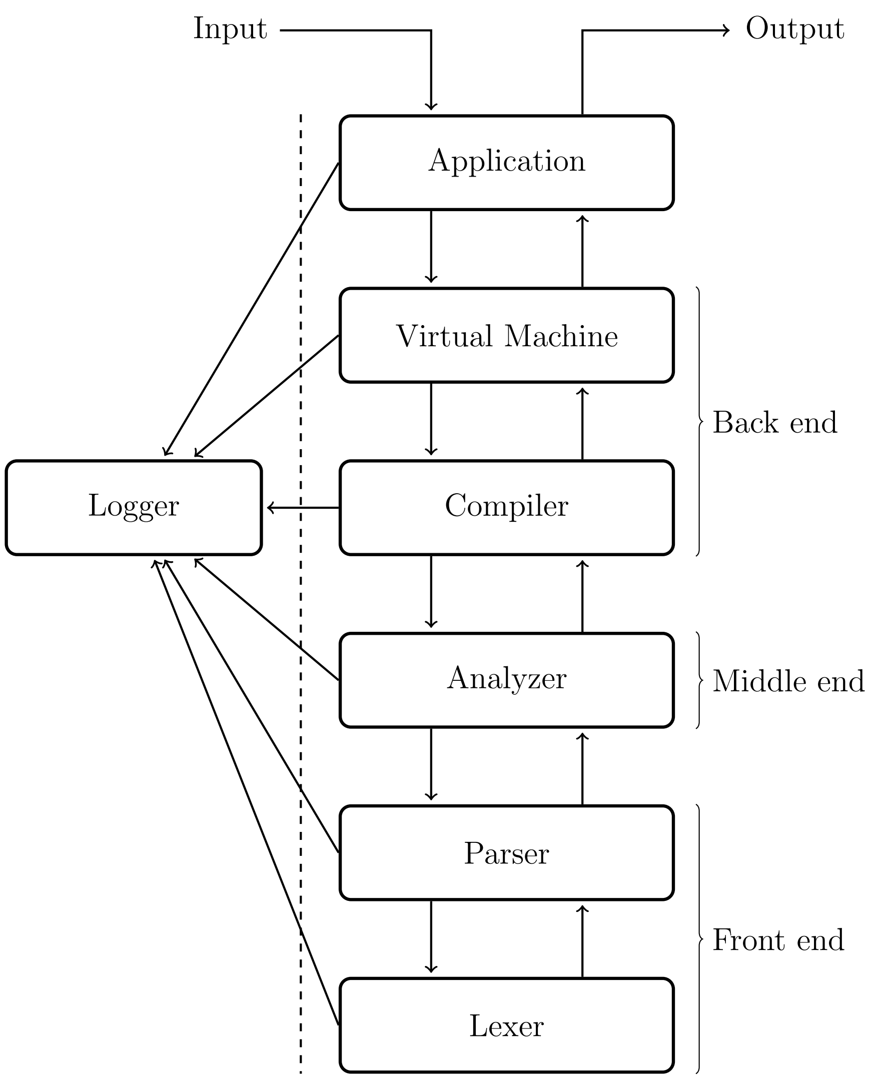

Figure 1.2: Nuua's Layered System

Chapter 2

The Nuua Language

This chapter will define all the Nuua Language. It provides enough information to get started with it and it provides the knowledge to understand how it is suposed to work.

When defining the Nuua Language, I had to look for an unambiguous grammar that could provide simple, elegant and yet non-ambiguous syntax to code a nuua program. The final result was cleaner than expected and yet very functional.

2.1 Data Types

Data types represent all the native types that Nuua supports for it's values. By default Nuua gives the programmer some common data types to work with. In addition to those types Nuua already have a casting mechanism to allow casting between diferent types, allowing operations between values of diferent types to be performed without issues (for example, making a simple addition between an integer and a float).

2.1.1 Integer

Integers are named as `int` and it represents subset of numbers represented by $\mathbb{Z}$. It includes 0 and the integers are represented using 64 bits. Meaning it's range for a given integer x is:

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

2.1.2 Unsigned Integer

Unsigned integers are named as `uint` and they are simply nonnegative integers (positive integers) $\mathbb{Z}^* = 0 \cup \mathbb{Z}^+$. It's value is also represend using 64 bits, meaning it's range for a given unsigned x is:

$$0 < x < 2^{64} - 1$$

2.1.3 Float

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

2.1.4 Boolean

Booleans are named as `bool` and they are simple booleans, they can be either `true` or `false`.

```
true / false
```

2.1.5 String

Strings are named as `string` and they are used to manipulate arrays of chars. It's implementation uses a C++ `std::string`. It can store any text that's surrounded by `'"'`.

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

2.1.6 List

Lists are named as `list[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. Lists are defined as `'['` followed by any number of items separated by a comma, and finishing with a `']'`. To access elements in the list you can reference it by using it's integer key.

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

2.1.7 Dictionary

Dictionaries are named as `dict[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 as the key. That means that to access an element, you need to provide it's string key instead of the key index. Dictionaries are defined as `'{'` followed by an identifier (not a string!) without spaces, then with a `':'` to mark the end of the string key and finally followed by it's value. You can then add more elements following the same pattern and using a comma as a separator. When all elements are added you then need to finish the dictionary with a `'}'`. Dictionaries, as lists, can only store 1 type of values. So each key can only store the same type.

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

2.1.8 Function

Nuua functions are a bit special, it's name in Nuua is `fun` they are always represented as what's commontly known as annoymous functions. That means that functions are values, and therefore they need to be assigned if you want to save it. This allows quick and easy callback mechanisms that will be further explained. However, as this concerns, functions share a similar syntax as what's also commontly known as arrow functions. However, nuua functions can be defined in 3 diferent ways, each resulting in diferent function bodies.

The base syntax starts with a `'('` followed by the declarations of the arguments separated by a comma (declarations are explained in the following sections). Then when the arguments are added, you need to close the parethensis with a `')'`. Since now, the sintax feels it's the same as a simple math grouping. However, after the closing parenthesis you'll need to specify the return type of the function. So you need to add `':'` followed by the return type. There are however, 3 types of functions to define, they depend on the following characters you add after the return type.

- Single expression functions
- Single statement functions
- Normal Functions with multiple statements

Single expression functions are defininind by adding an arrow `'->'` after the function closing parenthesis followed by a single expression. This expression will in fact be evaluated and futher returned as the function result.

```
(a: int, b: int): int -> a + b
```

Single statement functions are defininind by adding a big arrow `'=>'` after the function closing parenthesis followed by a single statement. This statement will be the only thing evaluated when the function is run. Nothing (none) is returned if the provided argument is not a return statement.

```
(a: int, b: int): int => return a + b
```

Normal Functions with multiple statements are defininind by adding a new block `'{'` after the function closing parenthesis followed by a a new line and the block statements. When finished, you'll need to provide a closing bracket for the block `'}'` followed by a new line.

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

2.2 Operations

Operations can be performed into each value, to modify the outcome. For example, you might want to perform an addition of two integers $1 + 2$. This can perfectly be done and in fact there are more native operations that can be performed into values. Automatic casting will be performed if the values do not match in type. If no reasonable casting can be done it will fail at runetime with an error message.

2.2.1 Negation

Negation is an unary operation that simply negates the current value. It uses the operator `'!'`. This means it's boolean representation will be negated. For example, negating a `true` value, will result into a `false`. This operation can only be performed into boolean values, but automatic casting will be performed to achieve the result if it's not a boolean. The value would then be casted into a boolean and further negated.

2.2.2 Negative

Negative is an unary operation that takes the float representation (or integer) and applies it's negative result. It uses the `'-'` operand as an unary, meaning it just requires the right hand side operand. This makes the value be negative (if it was positive) or positive (if it was negative). The operation is done by multiplying -1 to the int or float value. The result will be an int or a float, depending on the input values.

2.2.3 Addition

Addition can be performed using the `'+'` operator. This is a binary operation, meaning it requires two operands, the left and the right handside of the operation.

A special case can be found when trying to add two strings `"abc" + "def"`, this will in fact result in the addition of both strings into a single resulting string `"abcdef"`. Otherwise, the value is transformed into a float and further used to add both values, returing a float.

2.2.4 Substraction

Substraction can be performed using the `'-'` operator. This is a binary operation, meaning it requires two operands, the left and the right handside of the operation. The value is transformed into a float and further used to substract both values, returing a float.

2.2.5 Multiplication

Multiplication can be performed using the `'*'` operator. This is a binary operation, meaning it requires two operands, the left and the right handside of the operation.

A special case can be found when one of the two operators is a string. This will result into the subsequent repetition of the string x times, beeing x the other operand. For example, `"abc" * 2` will result into `"abcabc"`. The same result will be achieved if the operands were inverted. Otherwise, the values are converted into a float and a normal multiplication is performed, returning a float.

2.2.6 Division

Division can be performed using the `'/'` operator. This is a binary operation, meaning it requires two operands, the left and the right handside of the operation. The value is transformed into a float and further used to divide the left hand side given the right hand side divisor, returing a float.

If the right hand side of the operator is 0, it will throw an error because a division by 0 cannot be performed.

2.2.7 Equality

The equality operation can be done using the `'=='` operator. This operation is binary, so it requires the left and the right hand side operators to work. It will then determine with a boolean value if the two values are the same. Bear in mind, this operation is not strict, beeing able to say `true` when for example, the following operation is given `10 == 10.00` or perhaps, a bit more clear to understand what that means by given diferent values that can still give `true`, for example, `0 == ""` or `false == 0`.

There is also the inverse of this operation. It uses the `'!='` operator. It works the same way but it gives true when the values are diferent.

2.2.8 Higher Than (or equal to)

The higher than opreation is a binary operation that uses the `'>'` operator to determine if the left hand side value is higher than the right hand side. It uses the float representation of the values to perform such an operation.

There is also a the same operation with the included equality test on it, you may use it with the `'>='` operator.

2.2.9 Lower Than (or equal to)

The lower than opreation is a binary operation that uses the `'<'` operator to determine if the left hand side value is lower than the right hand side. It uses the float representation of the values to perform such an operation.

There is also a the same operation with the included equality test on it, you may use it with the `'<='` operator.

2.3 Variable Declarations

Variables can be used to store data in an identifier you might recognize. However, you need to specify the variable type to know what kind of data type you're willing to store on it. This is called a variable declaration, you simply declare you want to use a variable named `variable_name` with a value type of `my_type`. The syntax is as follows:

```
variable_name: my_type
```

The list of types was specified in the last section. The variable name can be anything starting with a character followed by anything, including `'_'`. No spaces are allowed.

You may also specify an initializer to set the variable to an initial value. If no initializer is specified like the example above, the value will be initialized but into a 0 state. Meaning it will be usable. To declare a variable with an initializer, you just need to add a `'='` after the type.

```
variable_name: my_type = initializer
```

So for instance, if you want to declare a variable called `sum` that it's the addition of two integers 10 and 20, you may do such like follows:

```
sum: int = 10 + 20
```

2.4 Using variables

To use the value stored in a variable, you first need to declare the variable and then you need to load it by using the variable name as a value. For example, given the following declaration:

```
sum: int = 10 + 20
```

You may then load the variable by using sum:

```
sum: int = 10 + 20
res: int = sum * 2
```

In this example, the variable `res` will have the value of 60.

2.5 Control Flow - Conditionals

...

2.6 Control Flow - Loops

...