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
Directed by Sebastia Vila Marta
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 must be simple, elegan and yet it needs to follow the 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 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. This

[1] Big ball of mud (Wikipedia)

section details the most important aspects found in a typical programming language and in a compiler infrastructure. This high level overview won't deal with details and only explains the basics to understand the whole system without deep understanding. However is good to note that the most import aspecs can be explained further on their respective chapters.

1.2.1 Language grammar

Most programming languages can be expressed using a notation called **Backus Naur form** or **BNF** for short. This notation is a form of formal grammar (Specifically, a context-free grammar) that is used to specify the rules, grammar and alphabet of a language. The BNF form is often used to write a language specification, so language implementators are able to implement and understand the grammar and rules of a certain language. This document uses a variant of the BNF notation called **Extended Backus Naur form** or **EBNF** for short with a modified syntax to write the Nuua language specification.

In short, in a formal language there's two diferent types of symbols. Terminals and non-terminals.

- **Terminals**: Symbols that are literals. In a programming language, literal symbols may refer to signs like '+', '*' or even numbers or identifiers. Those are often called tokens and it's part of the result of a lexing process.
- **Non-terminals**: Symbols that can be replaced or often reduced based on the grammar rules. It's replacement or reduction ends up beeing a terminal symbol.

The EBNF notation have the following constraints.

```
symbol = pattern;
```

In this code, **symbol** represents a non-terminal symbol and **pattern** represents the current non-terminal symbol grammar rule. In EBNF there are ways to represent diferent situations. This table releates the EBNF syntax symbols that can be used to define rules.

Symbol	Definition
=	Used to define a rule.
,	Used to concatenate patterns.
|	Used to define a union of two sets of patterns.
*	Used to define an exception.
-	Used to define a union of two sets of patterns.
[...]	Used to define an optional pattern.
{...}	Used to define an 0 or many pattern.
(...)	Used to group a pattern.
"..." or '...'	Used to define a terminal symbol.
(*...*)	Used to define a comment.
?...?	Used to define a special pattern.
; or .	Used to terminate a given grammar rule.

The following example defines a set of symbols to define how an integer may look like, allowing any integer to have an optional symbol prefix followed by as many digits as needed.

```
integer
     = ["-" | "+"], digit, {digit}
     ;
digit
     = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
     ;
```

However, this document will use a modified version of EBNF that deals with common regular expression (regex) patterns. The special symbols found in this modified version are:

Symbol	Definition
`:`	Used to define a rule.
space	Used to concatenate patterns (space separated).
`|`	Used to define a union of two sets of patterns.
`...+`	Used to define a one or more pattern.
`...*`	Used to define a zero or more pattern.
`...?`	Used to define an optional pattern.
`(...)`	Used to group a pattern.
`"..."` or `'...'`	Used to define a terminal symbol.
`;`	Used to terminate a given grammar rule.

So the previous example grammar would now be written as:

```
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 aproach to write the language grammar.

1.2.2 Compilers

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 (Assem-

bly). 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).

Figure 1.1: Compiler overview

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.

- **Machine-language compiler**: Compilers whose target language is executable code (machine-language) can be run by the operating system. 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.

- **Interpreter**: Interpreters are often a very diferent overview of this problem. Interpreters do not compile to machine language and instead it often compiles to bytecode, a portable and reduced instruction set similar to machine instructions 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.

1.2.3 Phases of a compiler

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

Those phases are often found to be diferent 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 pahase, the input source is transformed from a character string into a token list, this is also called tokenization. Tokens are known and have certain attributes. For example, some tokens might include integers, symbols (+, -, *, etc.) or identifiers among others. Lexemes are also evalued as individual tokens, making a single token for identifiers matched as language keywords (like 'if', 'while', etc.). This phase is also called lexer or scanner.

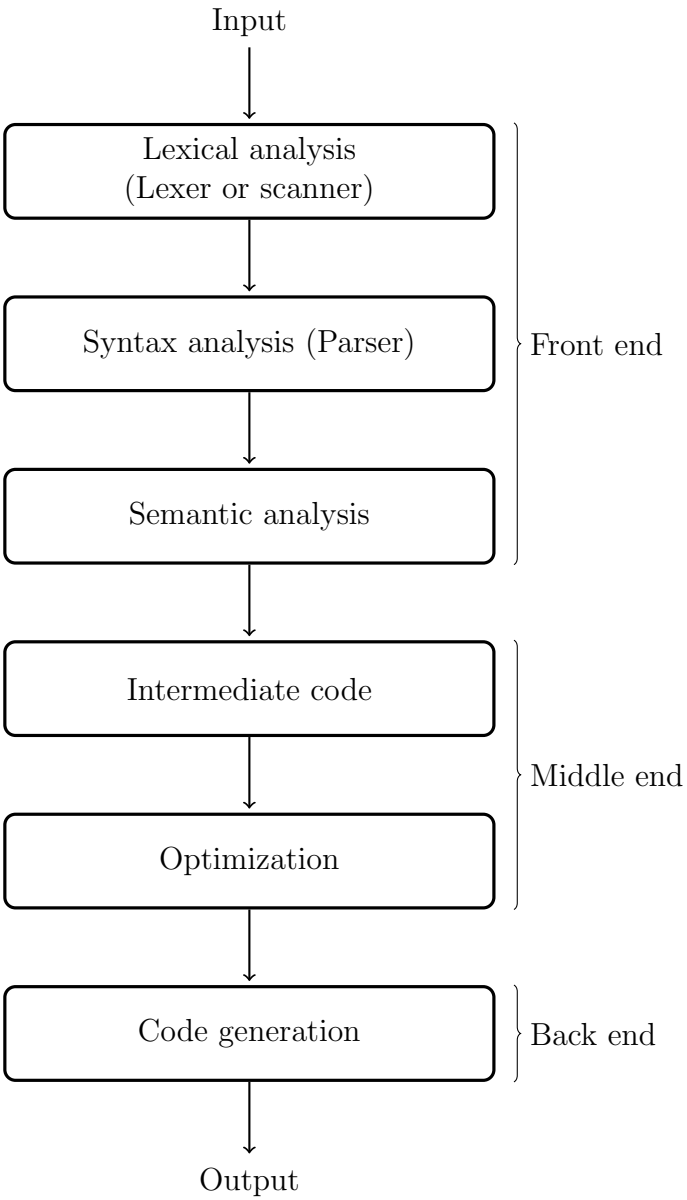

Figure 1.2: Common compiler phases

- **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 to the token list to parse the input and create an Abstract Syntax Tree (AST[2]). An AST is already a structural representation that represents your input source. This stage determines if it is a valid program based on the language grammar and the specified rules. There are also scannerless parsers that takes the lexical analysis and the syntax analysis into a single step. It is harder to understand and build but it often leads to faster parsing and less memory usage.

- **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.).

- **Intermidiate code**: An intermidiate representation (IR) can be avoided but it's often used to have a platform independent optimizer. Usually the code generation target a specific architecture and would require diferent optimizers depending on each architecture. However, by having a IR it's possible to have a single optimizer. A much used IR is Three Adress Code (TAC[3]) that can be organized in quadruples or triples.

- **Optimization**: This optimization is often performed on the IR and performs diferent 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 diferent language output. For example machine code. This phase often have to deal with instruction scheduling or register allocation while they have to output a fully working program.

[2]Abstract Syntax Tree (Wikipedia)

[3]Three Address Code (Wikipedia)

1.3 System architecture

To design the system architecture of Nuua, a 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 a 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**.

A layered system have specific constraints regarding to the whole communication. Specifically, a layered system consists of diferent layers arrenged vertically. Those layers have a specific criteria that needs to be met. As a matter of fact, each layer can only use the layer below and and gives an API for the layer above (if any) to use it's functions. For example, the Figure 1.3 contains a simple 3-tier layered system. The Layer 3 can only use Layer 2, and the output comes from Layer 2. Layer 3 can not 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.

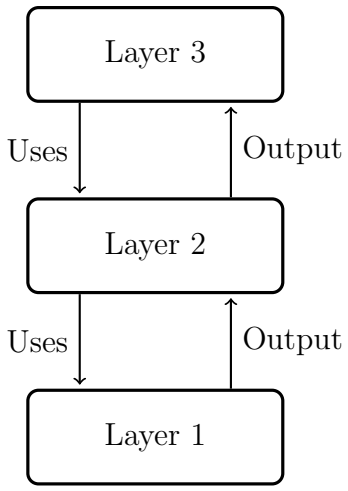

Figure 1.3: Layered system

This system is known to be robust 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 each layer gets completly isolated and works independently by just using the layer below, 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.

The Nuua architecture is shown at Figure 1.4. 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, it's purpose is to analyze the command line arguments and fire the application acordingly.
- **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.
- **Compiler**: Is responsible of the translation of the AST to the virtual machine bytecode. This acts as the code generation part of a compiler architecture.
- **Analyzer**: Does all the semantic analysis of the compiler and optimizes the AST for a 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 a tokens, creating a list of tokens representing the original source code.

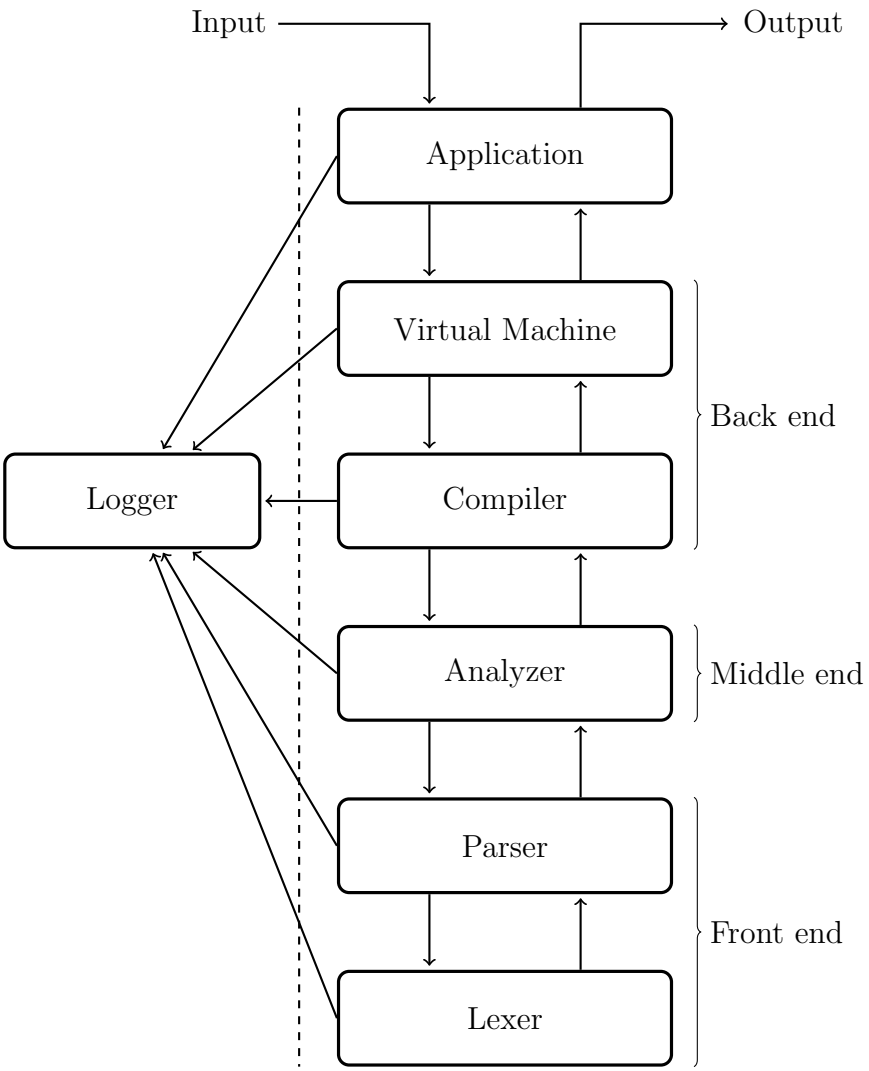

Figure 1.4: Nuua's architecture (Layered System)

Chapter 2

Bibliography

[1] 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. [Online]. Available: `https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5019`.

[2] 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.

[3] 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.

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

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

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

[7] Federico Tomassetti, "EBNF: How to Describe the Grammar of a Language ", English, 2017. [Online]. Available: https://tomassetti.me/ebnf/.

[8] Stanford, *Three-Address Code IR*, English. [Online]. Available: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/lectures/13/Slides13.pdf.

[9] Bartosz Sypytkowski, "Simple virtual machine", English, 2014. [Online]. Available: https://bartoszsypytkowski.com/simple-virtual-machine/.

[10] Andrea Bergia, "Stack Based Virtual Machines", English, 2015. [Online]. Available: https://andreabergia.com/stack-based-virtual-machines/.

[11] Bob Nystrom, *Crafting Interpreters*, English. 2018. [Online]. Available: https://craftinginterpreters.com/.

[12] ——, "Pratt Parsers: Expression Parsing Made Easy", English, 2011. [Online]. Available: https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/.

[13] Kein-Hong Man, esq, "A No-Frills Introduction to Lua 5.1 VM Instructions", English, 2006. [Online]. Available: http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf.

[14] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes, "The Implementation of Lua 5.0", English, 2005. [Online]. Available: https://www.lua.org/doc/jucs05.pdf.

[15] ——, "Closures in Lua", English, 2013. [Online]. Available: https://www.cs.tufts.edu/~nr/cs257/archive/roberto-ierusalimschy/closures-draft.pdf.