The Design of an Experimental Programming Language and its Translator

The Nuua Programming Language

ÈRIK CAMPOBADAL FORÉS

A bachelor thesis submmited to fulfill the degree of
ICT Systems Engineer

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

Contents

List of Figures

1 Introduction

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 for the computer to execute it.

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

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 acording to the project needs.
- Design the Nuua Programming Language. This grammar must be simple, elegant 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 build the compiler with. Among other options, the languages that satisfy the previous statement are low-level programming languages like C, C++, D, Rust, Go, Crystal, Zig or Nim.
- Define a scalable software system to allow the compiler to grow efficiently. The software system that is chosen will be very important towards the scalability of the project.
- Build a compiler for the Nuua programming language and a very simple standard library.

1.2 Preliminary overview

This section details the most important aspects found in a compiler. 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 [2, 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 ammount of variants that exists. To clarify the EBNF syntax that this thesis is going to use, the following table will be used to specify the production rules. More information may be found in the EBNF grammar article [7].

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 in 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 language and translate it into another as showin in Figure 1.1. Often, the compiler term is 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. However, there are multiple types of compilers, and each one is used to implement different translation tasks. Nuua's system architecture built in this thesis, as described in section 1.4, is based on a interpreter architecture, wich consists of a compiler that translates the Nuua language into an intermediate representation known as *bytecode* and further interpreted by a virtual machine.

Input language
Compiler
Output language

Figure 1.1: Compiler overview

1.2.3 Phases of a compiler

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

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

- **Lexical analysis**: In this phase, the input source is transformed from a character string into a token list, this is also called tokenization. Tokens are known and have certain attributes. For example, some tokens might include integers, symbols (+, -, *, etc.) or identifiers among others. Lexemes are also evaluated as individual tokens, making a single token for identifiers matched as language keywords (like 'if', 'while', etc.). This phase is also called a lexer or scanner.
- **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[1]). 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 scanner-less 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 intermediate representation (IR) can be avoided but it's

[1]Abstract Syntax Tree (Wikipedia)

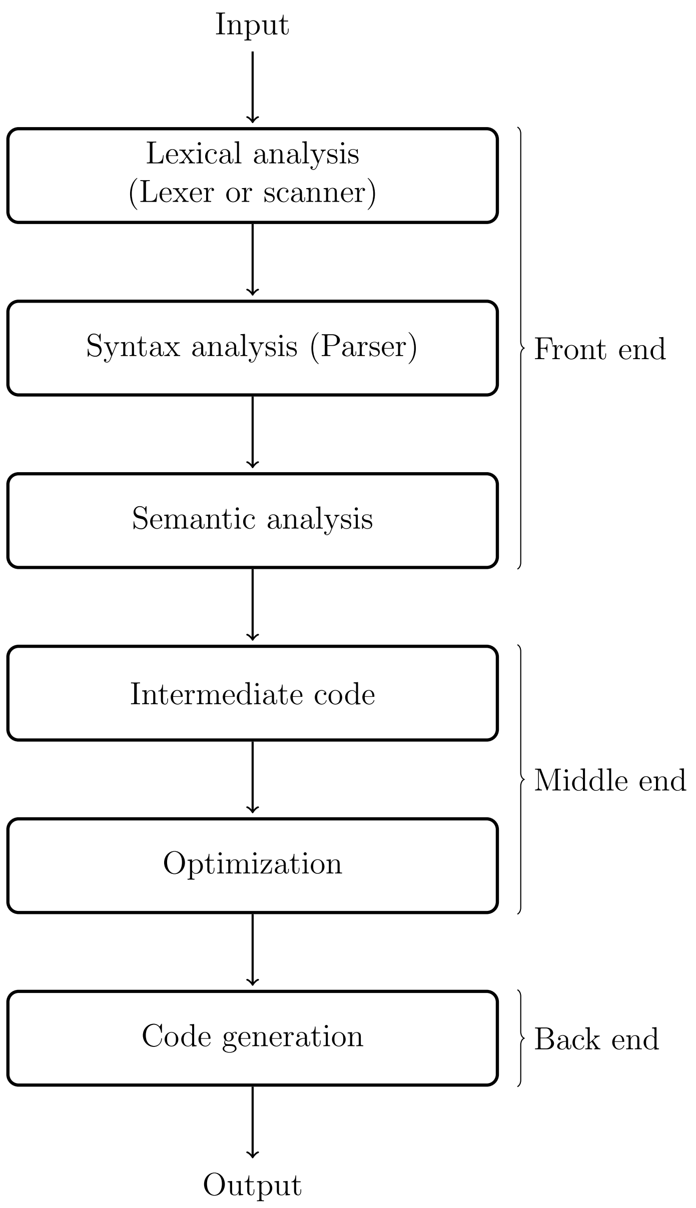

Figure 1.2: Common compiler phases

often used to have a platform independent optimizer. Usually the code 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 Adress Code (TAC[2]) that can be organized in quadruples or triples.

- **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.

1.3 Interpreters

There are different ways to interpret a language. This thesis implements a bytecode virtual machine (register-based) where the bytecode is the intermidiate representation generated by the Nuua compiler. Among the most common interpreters built for programming languages, we may find the following types of interpreters:

- **Bytecode interpreter**: The programming language is first compiled into an intermediate bytecode representation and further executed by this virtual machine. A very famous example of a programming language interpreted by virtual machine is the Lua virtual machine [14]
- **Abstract syntax tree interpreter**: This kind of interpreters just need the AST to work with, so no extra compilation to an intermediate representation is needed and therefore, easier to implement. However, due it's nature, they are much slower to execute and debug due the recursiveness of working with tree structures.
- **Just-in-time compiler**: 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 a specific moments while the program run to speed up portions of the code that is beeing 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.

[2]Three Address Code (Wikipedia)

1.4 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* [1].

A layered system has specific requirements regarding code communication. Specifically, a layered system consists of different layers arranged vertically. Those layers have a specific criterion that needs to be met. As a matter of fact, each layer can only use the layer below and gives an API for the layer above (if any) to use its functions. For example, the Figure 1.3 contains 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.

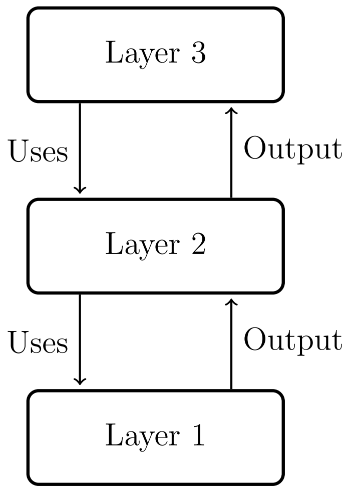

Figure 1.3: Layered system

This system is known to be robust and scalable at a very low-performance cost. It's very easy to understand and yet very powerful for software systems like this one. These 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 completely isolated and works independently by just using the layer below, creating a very impressive 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 for 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, the purpose is to analyze the command line arguments and fire the application accordingly.
- **Virtual Machine**: The virtual machine is the interpreter that runs Nuua. It's a register-based virtual machine that acts as the Nuua runtime environment.
- **Code generator**: Is responsible for the translation of the AST to the virtual machine bytecode. This acts as the code generation part of a compiler architecture.
- **Semantic analyzer**: Does all the semantic analysis of the compiler and optimizes the AST for faster and smaller output.
- **Parser**: Acts as the syntax analyzer, it uses a list of tokens and generates a fully valid AST.
- **Lexer**: Scans the source code and translates the characters to tokens, creating a list of tokens representing the original source code.

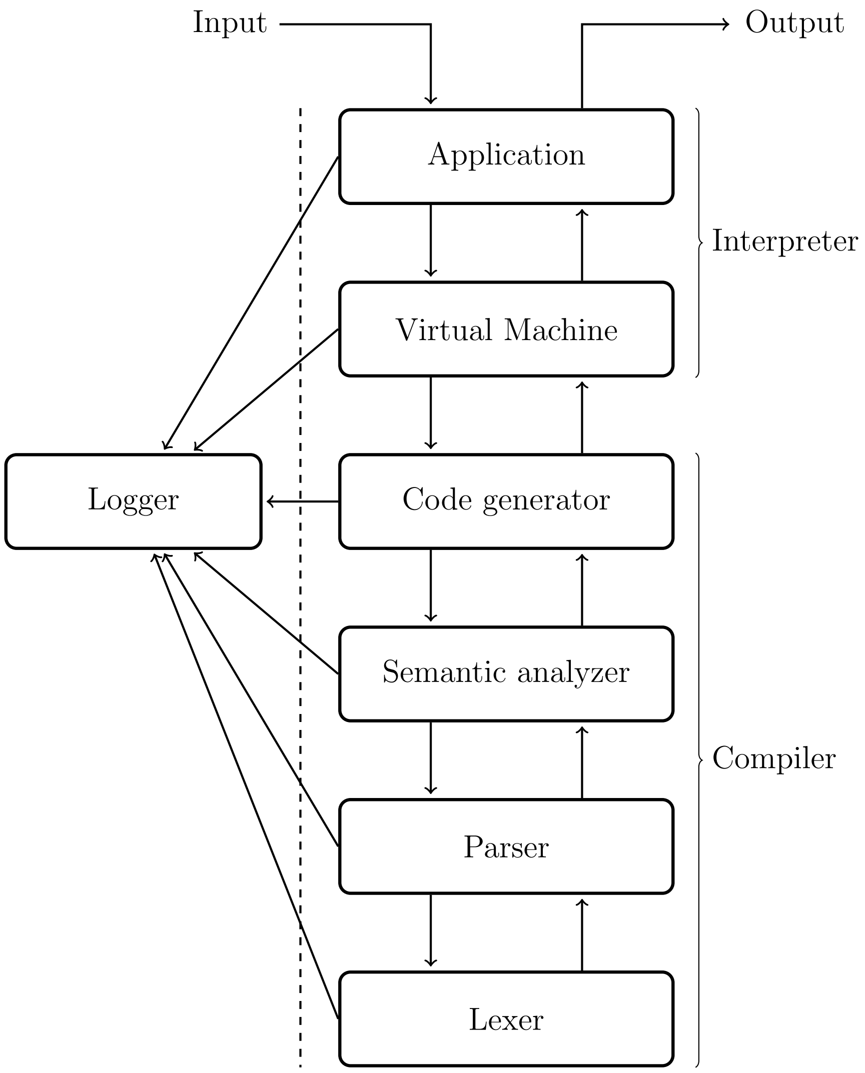

Figure 1.4: Nuua's architecture (Layered System)

2 The Nuua Language

This chapter defines the Nuua programming language. The language is set to be of a general purpose usage, with an inperative paradigm and a statically typed system. The official Nuua compiler is written in C++ with a zero-dependency policy and it's implemented as an interpreter using a register-based virtual machine.

2.1 Nuua 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 specially from Python, Rust and Go.

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

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

2.1.1 Lexical Grammar

The 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 [11, Appendix I] is that the syntax grammar is a context free grammar and the lexical grammar is a regular grammar.

Nuua's lexical rules are as follors:

```
DIGIT
    :  '0' ...  '9'
    ;
```

Digits are any character from '0' to '9', given that the ASCII table [1] uses a sequential

[1]ASCII table (Wikipedia)

enumeration for them, it's very easy to determine what characters are between '0' and '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
    :  DIGIT+
    ;
```

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

```
FLOAT
    :  DIGIT+ '.'  DIGIT+
    ;
```

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

```
BOOL
    :  'true'
    | 'false'
    ;
```

Bools are either 'true' or 'false', that are reserved words.

```
STRING
    :  '"' # '"' # '"'
    ;
```

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

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

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

2.1.2 Syntax Grammar

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'
    | export_declaration '\n'
    | class_declaration '\n'
    | fun_declaration '\n'
    ;
```

A top level delcaration 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.

```
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.

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

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. Otherwise, Nuua imports the specified targets from the modules.

Expressions

```
LIST
    :  "[" expression (',' expression)* "]"
    ;
```

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

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

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

2.1.3 Operator precedence

3 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`.