UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA

DE

MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

WAXX - A TRANSPILER TO MODERNIZE EXISTING PROGRAMMING LANGUAGES

Absolvent

Irimia Ioan-David

Coordonator științific

Popa Alexandru

București, iulie 2020

Contents

Abstract

Today's programming languages tend to simplify their syntax and add "syntax sugar", which means ways to write long code constructs in a shorter, easier to understand manner. Syntax sugar can be as simple as the ability to drop semicolons at the end of statements or more complex, such as JavaScript's string interpolation. It is generally accepted that code should be readable. A challenge appears with older languages and ones that refuse to divert from their current syntax. In this synthesis-type paper, we present Waxx, a programming language/transpiler/tool inspired by Python to take on that challenge and modernize coding. It's objective is to offer a single syntax for multiple imperative, object-oriented programming languages and a way to write clean code that is on par with today's standards. This project's compiler takes syntactically correct Waxx code and converts it into source code in a chosen target language. It does so with the help of Finite State Machines and various concepts of theory of automata and compiler theory. Waxx programs can use classes, functions, and other objects defined in the source code of other project files (written even in other languages). Waxx code does not need to be semantically correct (e.g. it does not check variable types or even if they have been previously declared), so it allows Waxx integration in existing projects and writing arbitrary, "detached" code to be used elsewhere completely. Waxx can also function as a "pseudo-interpreted language", by compiling to JavaScript and running it immediately in the browser, so one can achieve basic tasks and test the language at leisure.

Keywords: Waxx, Python, JavaScript, syntax sugar

Abstract

Limbajele de programare din ziua de astăzi tind să își simplifice sintaxa și să adauge syntax sugar" (zahăr sintactic), referindu-se la modalități de a scrie construcții lungi de cod în maniere scurte și ușor de înțeles. Zahărul sintactic poate fi simplu, precum posibilitatea pentru programator de a nu pune punct și virgulă la sfârșitul intrucțiunilor, sau mai complex, cum ar fi interpolarea stringurilor în JavaScript. Este acceptat ca un adevăr că programele trebuie să fie lizibil. Limbajele mai vechi de programare și cele ce refuză să își adapteze sintaxa pe care o au în prezent pune o barieră pentru programatori. În această lucrare de tip sinteză prezentăm Waxx, un limbaj de programare inspirat din Python cu un transpiler care să ridice acea barieră și să modernizeze scrierea de software. Obiectivul său este să ofere o singură sintaxă pentru mai multe limbaje imperative și orientate pe obiect și o metodă de a scrie cod curat, la curent cu standardele contemporare. Compilatorul acestui proiect ia cod corect din punct de vedere sintactic (în limbajul Waxx) și îl transformă în cod sursă în limbajul țintă ales cu ajutorul automatelor finite deterministe și al diverselor concepte de limbaje formale și teorie a compilatoarelor. Programele Waxx pot folosi clase, funcții și alte obiecte definite în codul sursă al altor fișiere din proiect (scrise chiar în alte limbaje). Codul Waxx nu trebuie neapărat să fie corect și din punct de vedere semantic (de exemplu, el nu verifică tipurile variabilelor sau dacă măcar au fost declarate înainte). Prin urmare, Waxx se poate integra cu proiecte existente și oferă posibilitatea scrierii de cod detașat" ce poate fi folosit și în alte locuri. De asemenea, Waxx poate funcționa pe post de limbaj pseudo-interpretat" prin transpilarea sa în JavaScript și rularea sa imediată direct în browser, dând programatorului o modalitate de a testa limbajul și coda programe simple într-un timp foaret scurt.

Keywords: Waxx, Python, JavaScript, syntax sugar

I Introduction

Code obfuscation is one of the primary causes of faulty software and often results in "spaghetti code". Too often do developers not have the means and knowledge to write simple, readable code. Some programming languages don't provide the means (such as C++, notorious for its complicated syntax and, in older versions, lack of syntax sugar). They abide by conventional syntax constructs, such as semicolons at the end of statements, excessive parentheses, blank lines containing only brackets, etc. These constructs no longer bring any meaning in modern programming. While they do provide meaning for the compiler, they don't for the programmer. Moreover, many of today's compilers and interpreters for programming languages are smart enough to strip away those features.

Some of those languages have proved that readable code can be enforced to the user not only through naming conventions but also through banishing noisy code patterns. The most notable example is Python, which is a primary source of inspiration for Waxx and whose syntax is further analyzed in this document.

Unfortunately, some languages enforce these noisy patterns through their very syntax. We can't write C++ code without them, so you have no choice but to risk writing fuzzy code. Yes, you can write the same programs in both C++ and Python, but the languages themselves don't work the same and don't have the same features. But while the language features do matter, but the syntax doesn't. This is where Waxx comes in.

Waxx is a tool through which I contribute to modernizing programming, consisting of a programming language syntax (just the syntax) and a compiler, which compiles Waxx code into another language (currently handling Haxe). Waxx aims to provide syntactic sugar features to languages lacking them or to enrich their already existing syntax by stripping away code noise and providing a set of guidelines for writing code. One syntax to rule them all.

I.1 Goals

The main goal of this thesis is to describe Waxx and its implementation, along with providing example code snippets and analyzing various forms of modern syntax, describing other notable programming languages and taking a deep dive into the design of Waxx compared to the other languages presented. This paper also contains a guide and documentation for the compiler and the Waxx syntax. The implementation of Waxx is open-source and fully accessible for readers online; the link can be found in the appendix [A].

An analysis by Ghazala Shafi Sheikh [23] creates several comparison criteria for programming languages: Simplicity (property of being close to natural language and the inexistence of unnecessary overhead), Writability (existence

abstraction and the inexistence of boilerplate code), Reliability (garbage collection, exception handling, and pointer safety), Appropriate Data Structures, Availability, Market Demand, Community Support, Machine or OS Limitations, Available Libraries and Coverage of programming concepts. Waxx aims to add to all of the above-mentioned criteria for any chosen language from the available ones for output, arguably except for Reliability in terms of adding to pre-existing mechanisms of the language, Market Demand, and Community Support. Waxx achieves these, respectively:

- Simplicity: Waxx is very close to natural language. For example, using the "my" keyword instead of "this." (explained later)
- Writability: Waxx does not have unnecessary parentheses and overhead. Any overhead or boilerplate code can be inserted from an external source by the "overhead" keyword/instruction
- Appropriate Data Structures and Coverage of programming concepts: Inspired from Kotlin, Waxx has "data classes", which are one-liner classes (explained later)
- Availability: Waxx is free and open-source
- Machine or OS Limitation: Given that Waxx compiles to multiple other languages, it improves the ability of a developer to write cross-platform code
- Availability of Libraries: Waxx has a small built-in library for code abstraction

I.2 Motivation

The two primary motivation factors which lead to the development of Waxx are:

1. My disliking of options and extensibility when it comes to language syntax sugar. Most developers want to code faster, simpler, and in a way that is easier to read. Legibility is arguably the most important aspect of the code in team projects; being able to understand someone else's code at a glance is mandatory. While comments do help, they are essentially noise with no meaning to the compiler. In Uncle Bob's book Clean Code: A Handbook of Agile Software Craftsmanship, he says that "The proper use of comments is to compensate for our failure to express ourselves in code". If we can manage to express in code what the code really means, comments can and should be stripped away. This is notoriously a long-running controversy in the programmer community and occasionally treated as a joke. Each joke, however, has a seed of truth at its core. Moreover, different languages can do different things. Python is not specifically meant for web development and even though Haxe is very cross-platform, it miserably fails to deliver reliable code that interacts with the browser's API the same way JavaScript does. Thus, I felt the need for a language to unify all of them.

Each language has its own philosophy, design patterns, capabilities, and so on. If we can build a bridge between these different potentials, it would be akin to the discovery of a theory of everything in the world of physics (hopefully, in the near future). My primary use of Waxx is to generate code for Haxe, one of my favorite languages which I actively use to develop apps and games as a hobby. Haxe is a powerful language, but I sought to make it even more powerful by enhancing it with syntax buffs.

2. My ambition to build a programming language from the ground up, go through all the processes involved, and finish a product without relying on external libraries. It is notable that I did not use Lex, YACC, ANTLR, or other such technologies in this project. I built my own compiler as cleanly as I could which other people who have never done such thing can inspect and easily understand the code behind it. I wish this thesis and project to be a valuable contribution not only as a tool that can be used to further enhance productivity while coding but also as a learning tool for those wishing to get into building a compiler. I wish I had such a document/project when I first started researching how to build a compiler. There now exists one, just the way I wanted it to be.

I.3 Thesis Structure

This thesis is separated into several sections, addressing language design, theory of automata and compilers, the implementation of the Waxx compiler and a set of guidelines for Waxx.

The first section talks about related projects and languages. It covers serveral programming languages tat served as inspiration for Waxx, including Haxe, Python, JavaScript, CoffeeScript, Kotlin, Groovy, Nim and others. It also explains antipatterns in language design and analyzes the Zen of Python and the controversial “camel-case versus snake-case” debate.

The following section describes preliminary knowledge required for a correct understanding of how Waxx works. It contains subsections such as the technology used, syntax sugar explained, the differences between transpilers, interpreters and compilers, the concepts of zero-cost abstraction and detached code. It then goes on to talk about the structured program theorem and various notions of formal language theory and compiler theory.

The implementation section goes through all the source code of this project. It explains the repository structure and illustrates how each and every source code file of Waxx works. It also takes into account other project-related topics, such as what did and did not work, whether Waxx is scalable and why Lex/YACC were not used.

The final section of this paper consists of a guide on how to download, install and get Waxx working. It explains the syntax of Waxx and describes each of its features in a separate subsection.

II Related Work

This section goes over some existing programming languages that have influenced the development of Waxx. The most essential languages presented here are Haxe, JavaScript, and Python. That is because Haxe is the main output language of Waxx, and Python is the pioneer for Waxx syntax and JavaScript is the language Waxx is developed in. They are each explained separately in their respective subsections.

We will also take a look at other programming languages that have affected the design of Waxx, aka related work. These include older as well as newer programming languages, such as CoffeeScript, Kotlin, Nim, and more.

The following sections briefly describe how other projects, namely CoffeeScript, Kotlin, and Nim are related to Waxx and how they solved or failed to solve certain problems.

They all have in common one thing: they aim to improve the syntax of their "predecessors': CoffeeScript for Javascript, Kotlin for Java, and Nim for Python. They do this with the addition of syntax sugar, described in each language's section individually.

II.1 Haxe - Primary Output

From haxe.org: "Haxe is an open-source high-level strictly-typed programming language with a fast optimizing cross-compiler." Haxe takes code written in Haxe (whose syntax is very similar to Java) and outputs code written in other languages, such as C++ or JavaScript. It was created by Nicolas Cannasse as a successor to ActionScript, but has evolved to become extremely cross-platform and is now primarily used for making video games and web applications.

Haxe tries to unify multiple programming languages into one, by providing a full suite of tools for programming: a language with a compiler, a standard library, a package manager, etc. However, you are bound by the limitation of its syntax, which made after Java. That means semicolons, brackets, and forced parentheses. Waxx does a similar thing but addresses only the syntax problems of other languages. When you write Waxx code and compile it to Haxe, you actually write Haxe code with Waxx syntax.

I develop games using Haxe in my spare time, but I am not satisfied with its syntax. I often feel it is too verbose and needs so much code to express so little logic.

Haxe is, though, very high level. Its standard API is remarkably simple and, fortunately, it has very few exceptions to its rules. It always uses the camel case (explained later). Functions (methods) always sit in classes. One expects all

class names to start with a capital letter and all method names with lower case letters and the list goes on.

A free, online Haxe compiler can be found at the link in the appendix [B].

Here is a simple example program in Haxe.

Code II.1

```
class Test {
    static function main() {
        trace("Hello World!");
    }
}
```

Haxe also has access to JavaScript-like anonymous objects.

Code II.2

```
var dog = {
    name: "Daisy",
    age: 14
}
```

This also means Haxe has native support for JSON processing; one can access a "Dynamic" object by property name with a dot (just like in JavaScript) without needing to use square brackets and a string, like in Python dictionaries and hash maps in other languages.

Another notable feature of Haxe is metaprogramming: changing the code itself at compile time through macros. Upon hearing the term "macro", most people frown at the idea of hiding code behind a find-and-replace-all which might not even be analyzed correctly by the compiler to signal errors. Truth be told, in C/C++ macros can leak out of scopes and namespaces and mess up other parts of the code. Macros are a powerful feature if used correctly and Haxe tries to solve this problem (and succeeds). In Haxe, macros work as callable functions which are interpreted at compile-time by a special engine, before the whole code gets compiled. Essentially, yes, macros do return code, but the engine behind it always makes sure that what comes out of is a self-contained expression which is also syntactically and semantically correct. So, the expression it returned must also fit in with the rest of the code, at the "left" and "right" of the returned expression. Furthermore, the code inside a macro function is also Haxe! You can use all the standard Haxe API at compile time inside a macro. And that's not all: macros can affect code outside of their scope (with what is known

as decorators in other languages). A simple, practical example would be a macro that annotates a class to make every field it has also static.

Waxx draws inspiration from Haxe macros [30]. One thing that it does is exactly the example given above: the ability to create static classes. Waxx does not use macros for this - the feature is directly built in the syntax!

With Haxe being heavily inspired by Java (same class/package structure, same keywords, etc), we disregard Java's syntax in this paper as its own section. Java is, however, mentioned throughout the paper, but Haxe examples are given in many places where Java ones fit as well.

A later section describes Kotlin, an improvement over Java syntax-wise; therefore, it can also be considered an improvement over Haxe.

Several examples of Waxx to Haxe are given in later sections.

II.2 Python - Syntax Pioneer

Python is a powerful and fast programming language that is easy to read, write, learn and integrate with other systems [37]; its standard library is also very simple ("batteries included"). As of 2019, Python is the second most popular programming language on GitHub (as stated by their official statistics).

In 2017, Python was the second most demanded skill on Angel List and the highest paid skill on average [11] and has the second most voted syntax on Slant [33].

Python is high-level, typeless (loosely typed, it does not care about types), object-oriented, and interpreted.

While this section does not concern itself with Python's features, it does with Python's syntax. Waxx follows the same syntax principles as Python and strongly agrees with "The Zen of Python" [17]:

- Use indentation to delimit (nest) blocks of code (curly braces are used to delimit dictionaries or JSON objects)
- Avoid using parentheses for readability purposes
- No semicolons

Printing ”Hello World” is as simple as typing:

Code II.3

```
print('Hello World')
```

II.3 JavaScript - Development and Instant Scripting

Waxx can output fast, modern JavaScript code. Due to its implementation, Waxx can quickly compile to JavaScript and apply an ”eval” function to the resulting source code, which can make Waxx behave like an interpreted language. This allows for in-browser testing and demos. Just write your code and hit go! It works out of the box. This is possible because Waxx transpiles fast at its core. At the most basic level, Waxx compilation to JavaScript should be much faster than, say, C++ or Java.

An important note is that Waxx compiles to the latest JavaScript version. It requires an updated browser and as of 18th of June 2020, it only works on the following browser versions and above [34]: - Chrome 80 - Edge 80 - Opera 67 - Firefox 74, but not guaranteed to work It should also work with the Node.js at least version 14.0.0.

There is currently no support for Safari and Samsung Internet and there will be no support for any version of Internet Explorer. Future versions of these browsers might assure the correct functioning of Waxx generated JavaScript code.

It is part of the Waxx creed to advance technology and drive software to evolve and push it to new levels!

Code II.4

```
console.log("Hello World")  // or alert("Hello World")
```

II.4 CoffeeScript - Old but Gold

CoffeeScript is a programming language that compiles to JavaScript [5]. Its syntax is inspired by some other languages, most notably Ruby, Python, and Haskell. It uses indentation to delimit code blocks, drops the semicolons, and also drops parentheses for function calls.

CoffeeScript is relevant to Waxx because they have the same goal: to add syntax sugar to an existing language. CoffeeScript is to JavaScript what Waxx is to multiple programming languages.

And because JavaScript is interpreted and not semantically checked at compilation, it allows users to write Detached

Code and to let the JavaScript interpreter worry about semantics.

In 2020, CoffeeScript is declining because modern JavaScript provides more than enough features. CoffeeScript also worked as a means to write modern code which compiled to non-modern code so it works in older browsers. What CoffeeScript did wrong (in my opinion) is it failed to adapt to the newer syntax of JavaScript and it stripped away so much syntax that it went in the other extreme.

Firstly, CoffeeScript targeting JavaScript is a problem, because JavaScript's syntax is updated very frequently and CoffeeScript features were added to JavaScript. Waxx, on the other hand, targets a different category of languages: ones that abide by their syntax and that make a conscious choice not to add that much syntax sugar to the language. Backward compatibility is a big reason for that. One can't strip away features from a language without risking to enrage one's users and divert them away from the language. These are languages like C++, C, and Haxe.

Secondly, CoffeeScript has removed so many keywords and operators that it becomes hard to tell what the code actually does. No "function", "var" (or "let') keywords, optional "return" (when actually returning a value), etc. Moreover, CoffeeScript changed the way the user would write normal flow control code, such as for loops. ¡insert example¿. The point of simplifying syntax is to make the code more readable, which is the opposite of what CoffeeScript does.

There should be a balance between how much syntax and how little syntax is allowed.

Nevertheless, CoffeeScript had its time and brought to life numerous web applications, such as the code editor Atom, the task management website Trello, the mobile version of Airbnb, and many more. Even though it is becoming obsolete, it was certainly a pillar in web development and a needed cog in the gears of applications. (cite here!)

II.5 Kotlin - New and Improved

Kotlin is a clean, open-source, general-purpose language intended as a spiritual successor to Java. It is designed for the JVM (Java Virtual Machine) and can interchange its libraries with Java ones to offer interoperability between the two, thus being backward-compatible with Java programs.

Among others, Kotlin adds numerous syntactic sugar sprinkles over Java [8], such as:

- No semicolons
- No "new" keyword
- Type inference with "val" and "fun"

- The “it” keyword
- One-line classes or data classes
- Delegation

This paper only concerns itself with the syntax of Kotlin and how it affected Waxx’s design.

II.6 Groovy

Groovy is a niche, interpreted, and compiled programming language. Groovy is a better alternative to Java [9], as it is not only easier to read/write, but also faster to run since it has the option to be interpreted. It is additionally compatible with Java libraries, so it is interoperable with Java.

The idea of interoperability extended to Waxx. Groovy is an example of language whose design in terms of architecture and “language feel” is an ideal for Waxx.

II.7 Nim!

Nim is the newest programming language analyzed in this thesis (except for Waxx). It is a modern, object-oriented, typed, and general-purpose language with syntax heavily inspired by Python (such as delimiting scopes with whitespace). Nim is based on three core principles: efficiency, expressiveness, and elegance. Similarly to Haxe, Nim has support for metaprogramming, aka smart macros, which can change the way the language works without actually changing the syntax.

Aside from the modern approach to programming language design, an interesting feature of Nim is the ability to create Domain Specific Languages with macros, making scope-able constructs. This has inspired Waxx’s native support for YAML (explained later). Other features of Nim were taken into consideration for this project, such as having multiple types for accepted arguments and providing a model for writing clean documentation [18].

II.8 Antipatterns in Programming Languages

An antipattern is something that was considered useful or mandatory originally but afterward proving to be more harmful than advantageous. A symptom of an antipattern is spending too much time trying to understand what the

code does instead of actually fixing the problem or getting to work [4]. One single antipattern in syntax is not such a great deal, but together they add up and add unnecessary noise to code.

I consider the following things antipatterns in programming syntax:

II.8.1 Semicolons

A semicolon's purpose used to be to delimit one code instruction from another. In common language, its purpose is to connect two independent clauses. This defeats the purpose of semicolons at the end of statements in languages like C/C++, Java, Haxe, and others. The only acceptable place would be in a classic "for loop", to separate the three instructions (initialization, condition, incrementation). The classic "for loop" with three instructions can often be avoided. Waxx has opted out of semicolons in all of the above scenarios as it is preferable to put two different statements on two different lines.

II.8.2 Parentheses and Curly Braces

Even though parentheses help write correct code, they are often completely optional and don't bring much in terms of readability/writability/correctness. As is the case with Lisp, it's pretty much agreed on in the programmer communities that Lisp is hard to read because of its excessive parentheses. It is not a beginner-friendly/human-friendly language, the opposite of how Waxx is.

That is why Waxx gives up parentheses for flow control blocks (if, switch, while, etc) and adds the Pipe ("—") operator to get rid of even more parentheses (explained later), as inspired by Bash and Haskell's (the infix operator/function application operator) [28].

I also find scoping by indentation a nicer way to delimit code blocks; it's cleaner this way. Moreover, in Waxx it is optional to put a colon after a statement delimiting a child block of code, unlike Python.

II.8.3 Verbosity and Boilerplate

There are numerous cases when we want to write a block of code with simple logic, but to do so we need to add auxiliary code, which we call "boilerplate code". This extra code does not bring meaning to our program but is just a required addition for our block to work. This is especially prevalent when writing code in separate files; In the case of Java, writing a class also requires specifying the package, importing (possibly too many) libraries and defining getters

and setters. Such boilerplate is only noise when trying to read code; it distracts the programmer from the real focus of their work.

There are, however, ways to minimize boilerplate. As an example, project Lombok is a Java library that helps reduce Java boilerplate code. Java's getter and setter pattern is a controversial topic. Thankfully, Lombok [20] cleanly solves this. Two useful things it adds are the "@getter" and "@setter" annotations, which automatically generate getters and setters in the background.

To reduce boilerplate, Waxx uses the "overhead" feature, as well as other structures for writing only logic with things automatically being implemented in the background, such as "data classes".

II.9 Zen of Python

The Zen of Python [17] is a poem and a set of guidelines for writing clean and correct code. Waxx also adheres to this code of conduit for Python and adds to its own set of "rules" for writing good code.

Waxx specifically targets certain lines of this poem in its design, such as "Beautiful is better than ugly" in that keywords are more beautiful than symbols and "Readability counts", for the same reason [2].

II.10 Camel-Case vs Snake-Case (and Pascal Case)

Camel-Case and Snake-Case are the two most used coding style standards. This refers to identifier names (variable, functions, etc): do you separate the words in a variable name with underscores, or do you shift the case of the next word's letter?

Code II.5

```
mySpecialFunction()     // This is Camel-Case
my_special_function()   // This is Snake-Case
```

Haxe, Java, and Nim use Camel-Case, while Python, Ruby, and Rust are examples of languages that use Snake-Case.

Even though a study [3] shows Snake-Case is easier to read than Camel-Case (especially for novice programmers), Waxx has opted for the Snake-Case mostly because it looks beautiful and compact, but also because some text editors (Visual Studio Code, for example) tend to hide underscores in certain contexts (integrated console), so it becomes harder to tell apart two identifiers.

III Preliminary Knowledge

This section describes programming and compiler architecture notions and explains various concepts about how Waxx works.

III.1 Structure

This subsection describes the technologies used in developing Waxx and then describes terminology. It clarifies what Syntax Sugar is, overviews the differences between interpreters, compilers, and transpilers, defines zero-cost abstraction, and my concept of detached code.

It then goes on to illustrate the Structured Program Theorem and concepts of formal language theory, like languages, alphabets, regular expressions, syntax, and all types of grammars.

Finally, this section covers the theory of automata and computation along with its terminology, counting lexical analysis, syntax analysis, semantic analysis, and last but not least, finite state machines and (abstract) syntax trees.

III.2 Technology

Waxx is developed in JavaScript. The front-end (demo website) was built in plain HTML5, no frameworks. The back-end (also JavaScript) consists of a simple command-line interface built with Node.js. The localhost version of the demo website uses the "serve-static" node module, which sets up all the web files automatically.

The in-website text editor is a JavaScript library called CodeMirror with a custom created theme.

For Waxx to work, the user should have the latest version of their browser as described in the "JavaScript - Development and Instant Scripting" section of this paper.

The thesis was written in LaTeXusing the online editor Overleaf.

III.3 What is Syntax Sugar?

Syntax Sugar, as described by Peter Landin in the '60s [16], originally referred to substituting mathematical expressions and notations in algorithms for English words (for example, replacing the λ sign with the "where" keyword).

In today's world, it means a different, more pleasant way to express code so that it is easier to read and write, without actually improving the capabilities of the language.

The most basic example of syntax sugar is the "+=" operator:

Code III.1

```
a += 10 // Same as a = a + 10
```

Syntax sugar enhances the legibility and writability of a language, reduces the maintenance cost of projects (in terms of time invested in coding/understanding code/fixing bugs), increases productivity and, last but not least, helps make syntax from a specific syntactic sugar sprinkle into a norm in the programming culture [7].

Take the addition of "for each" loops to languages like C++. The original way of iterating through an array would be to increment a variable going from 0 to the index of the last element in the array.

Code III.2

```
for (int i = 0; i < n_elements; i++)
    // code
```

The "for each" construct not only significantly boosts readability and the speed of writing, but has also become mandatory for iterating through other collections without having to manually type "begin()" and "end()" every time:

Code III.3

```
std::map<int, MyClass>::iterator i;
for (i = myMap.begin(); i != myMap.end(); i++)
    // code
```

Compare the code above to the one below:

Code III.4

```
for (auto &i : myMap)
    // code
```

That's better, isn't it?

Furthermore, sometimes syntactic sugar can become the core of the language and will arguably not be considered sugar

anymore, just syntax. In python, there is no such thing as a C-like for loop (as exemplified in figure TODO). There is only for-each and it is standard. Therefore, it is no longer syntactic sugar, but the syntax of the construct itself.

Code III.5

```
for element in my_list:
    print(element)
```

Other languages opted for this, and it's mostly newer languages that do this, such as Haxe. The limit of older languages is that they have to be backward compatible. Therefore, syntactic sugar will always remain sugar sprinkled on top and not become the core of the language, and the more syntax there is, the less readable the language is. There are too many calories in the dough combined with the sugar [12].

III.3.1 Examples of Modern Syntax Sugar

A discussion I held on reddit (on the "ProgrammingLanguages" subforum) which received over 170 replies yielded plenty of interesting ideas and examples of modern syntax sugar in other languages (the link to the discussion can be found in the appendix [C]):

The C# null coalescing operator ("??"), the null conditional operator ("?") and null coalescing assignment operator ("??=") [39]:

Code III.6

```
x ?? y      // Same as: x ? x : y
x?.prop     // Same as: x ? x.prop : null
x ??= y     // Same as: if (x) x = y
```

Haskell's infix application operator ("$") [28] and Bash's pipeline operator:

Code III.7

```
f \$ g x    -- Same as: f (g x)
cat myfile.txt | grep 'Hello'
```

Matlab [38] and R ranges:

Code III.8

```
1:10    # Same as: [1,2,3,4,5,6,7,8,9,10]
```

String interpolation [32] and multi-line strings in Haxe and JavaScript:

Code III.9

```
'I am \$name! Hello!'   // Same as: 'I am ' + name + '! Hello!'
```

Haxe, Python, Elixir map (dictionary) constructor syntax:

Code III.10

```
{
    'name' : 'Daisy',
    'age' : 14
}
```

Note: this is not the same as anonymous object constructor syntax [31]:

Code III.11

```
var anonymousObject = {
    name : 'Daisy',
    age : 14
};
var myMap = [
    'name' => 'Daisy',
    'age'  => 14
];
```

Haxe, Haskell and Prolog pattern matching:

Code III.12

```
switch (animal) {
    case Dog('Daisy'):
        trace('Hello!');
    // Other cases
}
```

Chained comparisons (seriously, it's 2020, why doesn't every language have this yet?):

Code III.13

```
if 10 < x < 20:    # Same as: if 10 < x and x < 20
```

Dart and Wurstscript method cascading [36]:

Code III.14

```
duck..quack()
    ..fly()
```

List comprehension in Python, Haxe and others:

Code III.15

```
[x*2 for x in numbers if x % 10 == 0] # Python

# Same as the JavaScript equivalent of:
numbers.filter(x => x % 10 == 0).map(x => x*2)
```

And if we're at it, lambda functions in JavaScript:

Code III.16

```
x => x * 2
// Same as (mostly)
function(x) {
    return x * 2
}
```

III.4 Interpreter vs Compiler

A compiler takes code and, after heavily processing and optimizing it, outputs code in a different language, often in OS-specific machine code or bytecode. This leads to significantly better run-time performance but has the drawback of slow compilation time.

An interpreter is analogous to a virtual machine, such as the Java Virtual Machine or Google Chrome's V8, which runs code as it reads it, without needing to compile it first. The obvious advantage is the ability to run code without a compilation time, as well as being able to invoke functions directly from the command line, as is the case with Python, JavaScript, and many more [1].

Another perk of interpreted languages is that (usually) function names are kept inside the memory of the program. This allows for features such as adding or accessing properties dynamically by their names:

Code III.17

```
dog['name']      // Same as: dog.name
```

Waxx is a compiled language, but...

III.5 Transpiler

Transpilers, also called source-to-source compilers, are a subset of compilers with the specific task of translating source code from a language to another language [22]. Examples of transpilers are CoffeeScript, TypeScript, and Nim. Waxx is also a transpiler since its goal is to produce code in a targeted language.

III.6 Zero Cost Abstraction

Zero Cost Abstraction is a design pattern used to simplify code without affecting the run-time speed or memory cost of the program [26]. This is typically used more often in low-level languages like Rust and C++ (where it's called "zero-overhead abstraction") but is not exclusive to them (Haxe has them).

A simple example of Zero Cost Abstraction is the inline keyword in C++ and Haxe:

Code III.18

```
inline int sum(int a, int b) {
    return a + b;
}
sum(1,2);
// Same as: 1 + 2 in the background
// No function is actually called
```

This is important because speed and abstraction are important, and they go hand in hand together.

III.7 Detached Code

I coined this term concerning the name "Context-Independent Grammar", where a Nonterminal is replaced by a Terminal, regardless of what is around it.

Even though Haxe can compile to JavaScript, you generally can't take a piece of generated JS code, put it in an already existing JS project, and expect it to work properly. With other languages (like Java) the generated libraries have to be linked with an existing project since it's not exactly source code.

Transpilers avoid the hustle of code compatibility by generating code directly insertable into an existing repository and also using what code already exists from inside the source code before generation! Thus, the programmer can switch from a language to Waxx without any additional steps.

So, Detached Code is code written in a higher-level language (Waxx) which can be inserted in an already existing codebase without linking libraries or any other dependency.

III.8 Structured Program Theorem

The Structured Program Theorem (found by the name of Böhm–Jacopini theorem) [19] states that simple flowcharts can compute any program by applying the following control structures:

1. Sequencing (concatenation) - the ability to run a code block after another code block (sequenced functions)
2. Selection - the ability to run a code block only if a certain condition/boolean was met
3. Iteration (repetition) - the ability to repeat the running of a code block.

This is somewhat similar to the capacity of a computer program to be run by a Turing machine. The Structured Program Theorem is brought in this paper to express that is any program can essentially consist only of those structures, then the only thing that matters is how we tell the computer to organize them. The closer we can get to a language closer to humans, the better it can be.

Many people hold the belief that it's impossible to code in plain English because of how imprecise our natural language is. My argument is that the way we speak in real life is highly abstracted: a person can usually understand what another

means with a statement in plain English, mostly because other details don't matter. If you tell a person to water a plant and:

1. He or she knows what to do, there are be no hidden variables in this person's instructions and everything is clear. Natural language succeeds.

2. He or she doesn't know everything and asks the instructor, then this is comparable to warnings or "missing argument exceptions" in programming languages. Natural language succeeds again.

3. He or she doesn't know everything and proceeds with the task. In this case, it is not the fault of the language itself, but misuse of abstraction. The instructions were not clear enough and that person perhaps resorted to assigning default values to certain variables, doing a search which takes time to find or calculate those variables or eventually throwing an error in case the task failed to finalize successfully. If this happens, the instructor can further give more instructions until the problem is solved.

Humans don't necessarily obey and don't necessarily have a good memory. So, even in natural language, computers can better understand humans.

A study [35] shows that 80% of the code base of a program can be expressed in plain English: most of it (42% of the total) is comprised of condition statements (if's) while the rest of that remaining bit of 80% are other simple instructions which can be easily commanded to the computer in English. The last 20% of the total is made up of initializations, file reading and writing, name generation, and others, but importantly mathematical equations only accounted for less than 2% of the codebase!

That is why Waxx takes an English-like approach to programming: it's meant to be understood at first glance without concern for hidden variables or abstracted instructions. An argument against this is that the speed of the program can decrease at run time (such is the case with certain Java syntactic sugar constructs). This is not always the case, however, because:

1. As exemplified by Python and JavaScript map, reduce and filter operations which are highly optimized for speed at run-time more so than for loops

2. There exists zero-cost abstraction, which makes use of compile-time increases to produce the same code as if there were no syntax sugar (such is the case with Haxe macros). Zero cost abstraction is described in another section. (cite here)

III.9 Formal Language Theory

This subsection defines various notions required to understand the bits and pieces of Waxx's implementation. This includes details about formal languages and theory of automata/machines, syntax, grammars, and semantics along with the theory of computation counting lexers, parsers, abstract syntax trees, and other.

III.9.1 Languages

In our common lives, we use language to express ideas, actions, and emotions, as well as more abstract concepts that boil down to the ones mentioned above. The need for a Formal Language for a normal language is analogous to the need for a programming language for natural English. While we can supposedly reach a point where we program using common English as described in another chapter, we humans can't necessarily understand spoken languages just by having their formal description, which consists of sets of rules for that language.

Formal languages are present in multiple fields, such as computer science, linguistics, and philosophy. A definition for a formal language is a set of words, where each word must abide by certain rules. This section describes the theory necessary to be understood for the comprehension of how Waxx works and how it was implemented.

An example of a language is English. An English sentence is a list of symbols from the Latin alphabet, numbers, punctuation marks, and whitespace which follows a set of rules and conveys a meaning altogether.

III.9.2 Alphabet

An alphabet is a collection of symbols with which words can be formed. It is generally denoted by the sign, but in this paper, abiding by "syntax sugar", we shall simply refer to it as Alphabet. This is an example of the formalized version of the English alphabet (excluding punctuation marks, for clarity):

Code III.19

```
English Alphabet = {a, b, c, ... z, A, B, ... Z, 0, 1, ... 9, \_}
```

Note: "_" means white space in this context.

A String over an Alphabet is a sequence of symbols from the given Alphabet. Here are two examples of Strings over the Alphabet above:

Code III.20

```
String = dog
String = Daisy
```

Note that both words and sentences in their common meaning are Strings because a blank space is itself a character form an Alphabet. Japanese, for example, does not have blank spaces in its alphabet; its words are separated by using different subsets (Hiragana, Katakana, and Kanji) of its total alphabet for different words. (cite here)

A blank string is formally written as ϵ.

A set containing all possible strings in an Alphabet is denoted as Alphabet*.

Observation: The star (*) symbol after a term or set means all possible repetitions of that term or (respectively) all possible combinations of any length of symbols from that set.

III.9.3 Formal Language

Finally, a Formal Language over an Alphabet is a set of Strings formed with Symbols from that Alphabet that obey certain rules. An example of a Formal Language over the English Alphabet given above is:

Code III.21

```
Example Language = Set of all Strings of length 2
                   from the English Alphabet,
                   except '_'
```

Formalized as:

Code III.22

```
Example Language = {ab | ab in (English Alphabet} \ { '_' }}
```

Observation: A Language is always a subset of Alphabet*.

Most of the information in this subsection is taken from the “Formal Languages and Automata Theory” book [6].

III.9.4 Regular Expression

A Regular Expression (Regex) is a Language describing allowed, usually used for text searching and pattern matching in text.

III.9.5 Syntax

Syntax refers to how Strings from a Language can be organized together according to certain rules. The result of this organization of Strings forms a correctly written program. Let's take the following Strings from the English Language:

Code III.23

```
{home, dog, my, went}
```

"home my went dog" is not a valid construct in English, because it does not respect English rules. "my dog went home" is correct.

Formally, syntax is a context-free Grammar, a generator of correctly written programs. That means we can use a Grammar to validate our Language.

In common terms, the syntax represents the rules you must follow to write correct code that makes sense / how you write the code to make the language perform tasks; the way a programming language looks and is written. Assuming languages are Turing-complete, they should all be able to do the same tasks. The only differences are in how languages look.

For example, the following variable declaration means the same thing semantically, but uses a different syntax.

```
var x : Int;     // Haxe
int x;           // C++
o x : Int        // Waxx
```

They both declare in integer, x (ignoring the underlying representations of integers in memory).

III.9.6 Grammars

A Grammar (context-free grammar) is a set of rules for generating a language [25].

For example, take the following sentence:

Code III.24

```
the horse eats watermelons
```

The base structure for a very simple English sentence is:

Code III.25

```
<Subject> <Verb> <Object>
```

One could replace <Subject> with any noun-phrase from English, <Verb> with any verb and <object> with any noun phrase. So our Subject-Object-Verb construct becomes:

Code III.26

```
<any noun-phrase> <any verb> <any noun-phrase>
```

The noun "horse", articled with "the" forms a noun-phrase, while "watermelons" is another noun, which itself becomes a noun-phrase, and "eats" is a verb. We can substitute the tokens in our construct above with the terms "the horse", "eats" and "watermelons", resulting in a grammatically correct sentence and also a very happy horse.

Let us define the terms Terminals and Nonterminals as follows: words or structures which can be replaced by another one, such as <Verb> from above are called Nonterminals. Words that can no longer be replaced are called Terminals, such as "eats".

III.9.7 Context-Free Grammar, Formally

A Context-Free Grammar consists of Terminals ($\sum$), a set of Nonterminals (N), a set of rules which we call Productions (P), and a Starter (S), which is also part of the Nonterminals set.

Note: The notations used in this thesis are not necessarily standard mathematical notations; I prefer to use full words to describe terms for clarity, such as using "Terminal" or "Terminals" instead of using the symbol $\sum$. As an additional note, what is called "Starter" in this thesis is commonly called "start symbol".

We formalize it as:

Code III.27

```
Context-Free Grammar = (Nonterminals, Terminals, Productions, Starter)
```

Mathematically:

G = (N, $\sum$, P, S)

Context-Free Grammar = (Nonterminals, Terminals, Productions, Starter)

We use the following notation to express that a Nonterminal $\alpha can be replaced by a Terminal or Nonterminal \beta$:

$\alpha \longrightarrow \beta$

Such a rule as stated above is a Production. The result of a Production can be composed of multiple terms (Terminals and/or Nonterminals):

$\alpha \longrightarrow \beta_1, \beta_2, ...\beta_n$

We can also denote that a Nonterminal can turn into one term or another (or multiple) as follows:

$\alpha \longrightarrow \beta_1 \; \alpha \longrightarrow \beta_2 \; \alpha \longrightarrow \beta_3$

Note: this is also commonly written as:

$\alpha \longrightarrow \beta_1 | \beta_2 | \beta_3$

Lastly, the Language generated by a Grammar is the collection of all possibilities of sentences (lists of Strings) generated by Productions of the respective Grammar.

This theory is very helpful in understanding how Waxx processes and analyzes code. Waxx uses a specific custom Grammar and Waxx's Grammar is fully portrayed in a later section. The grammar is used as a template to transform text cod (written by a programmer) into an Abstract Syntax Tree as it defines what is allowed and what is not allowed in the language.

III.9.8 Types of Grammar - The Theory

As per Noam Chomsky, there are 4 types of Grammar [40]. Considering the rules and notations from the previous section, we define each type of Grammar as follows:

1. Type-3, Regular Grammar

A Regular Grammar's productions can only be from the following two:

a. Nonterminal $\longrightarrow$ Terminal

b. Nonterminal $\longrightarrow$ Terminal, Nonterminal

This is the simplest type of Grammar.

2. Type-2, Context-Free Grammar

A Context-Free Grammar encapsulates a Regular Grammar and adds the following possible production rule:

Nonterminal $\longrightarrow <$ any combination of Terminals and Nonterminals $>$

Explicitly:

$\alpha \longrightarrow \beta_1, \beta_2, ...\beta_n$

Where α is a Nonterminal and $\beta_{1...n}$ are Nonterminals and/or Terminals.

3. Type-1, Context-Sensitive Grammar

A Context-Sensitive Grammar adds the ability to check conditions regarding neighboring terms of a Nonterminal:

$\alpha_1, \gamma, \alpha_2 \longrightarrow \alpha_1, \beta\alpha_2$

Where $\alpha_{1,2}$ and β are Terminals or Nonterminals and γ is a Nonterminal. Colloquially, this means that if a Nonterminal appears in a given context, only then it is replaceable.

We will see in a later section that Waxx's Grammar is Context-Sensitive.

4. Type-0, Unrestricted Grammar

Unrestricted Grammars are actually recursively enumerable Languages [13], meaning that every possible String from that Language can be processed by a Turing machine (and Strings not from that Language are not accepted).

III.10 Compiler Theory and Terminology

This subsection describes the information needed to understand how Waxx compiles code and analyzes Waxx's implementation from a theoretical standpoint.

III.10.1 4 Steps of Compilation

When compiling source code, the compiler passes it through multiple layers of processing (called Compilation Phases), each with a different goal, to obtain a runnable program or source code in another language.

These layers of processing vary from compiler to compiler, but usually, they follow a common pattern:

1. Lexical Analysis

This is the first Compilation Phase. A Lexical Analyzer, or Lexer, takes source code in the form of text and splits it into Tokens [41] (also called Lexemes). These tokens are just separate data structures that hold a text component and, usually, a type.

For example, a Lexer may take the following code as input:

Code III.28

```
var x = 20;
```

And it will output a list of Tokens:

Code III.29

```
[
    ('var', KEYWORD),
    ('x', IDENTIFIER),
    ('=', OPERATOR),
    ('20', CONSTANT),
    (';', SYMBOL)
]
```

A Lexer does this using a finite state automaton (explained later).

Observation: For compilation efficiency, a Lexer usually sends tokens one by one to the next compiler Phase, immediately after finding one. This helps improve speed and memory because all the computation is done in one iteration of the main text, O(n) complexity (where n is the number of characters in the input text).

2. Syntax Analysis

The tokens resulted from Lexical Analysis go through a Syntax Analyzer (also known as Parser) and, using a Grammar, creates an "Abstract Syntax Tree" that can easily be iterated over recursively. A Syntax Analyzer can do this using a type of Push-Down Automaton.

There are two types of Parsers: Top-Down Parsers and Bottom-Up Parsers. Top-Down Parsers begin iteration at the Starter and try to make sense of a tree down to Terminals. A Bottom-Up Parser starts at Terminals and tries to produce a tree up to the Starter.

Waxx uses a Top-Down Parser.

3. Semantic Analysis

The Abstract Syntax Tree goes through a semantic analysis that checks that the AST makes sense semantically and tries to make sense of it if it doesn't. This includes steps such as applying type inference, checking types, checking variable scopes, etc [21].

3.1. Optimization

This optional step reiterates the AST and tries to optimize various constructs to make the program faster at run time, such as getting rid of unnecessary code

4. Code Generation

A recursive algorithm passes through the semantically analyzed AST and generates runnable code.

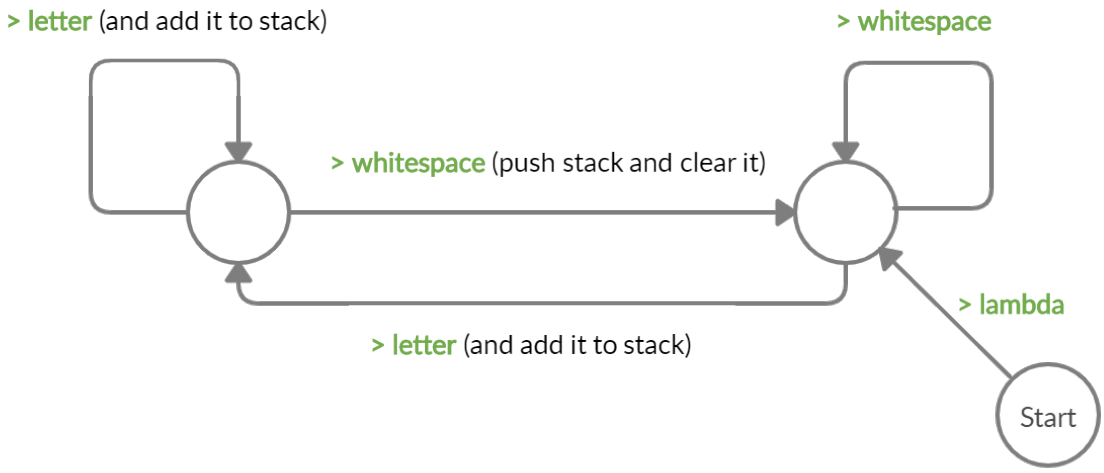
III.10.2 Semantics in a Transpiler

Semantics refers to the concept of giving meaning to syntax and either transforming it or running it as code. It breaches the gap between syntax and computation [29] - or between Abstract Syntax Trees and generated code.

A method known as Syntax-Directed Translation is used by transpilers to transform Abstract Syntax Trees into source code through a parser, where nodes are extracted and run through a Grammar to produce the desired result. By processing a sentence using a Grammar, Syntax-Directed Translation obtains a progression of rule applications.

III.10.3 Finite State Machines

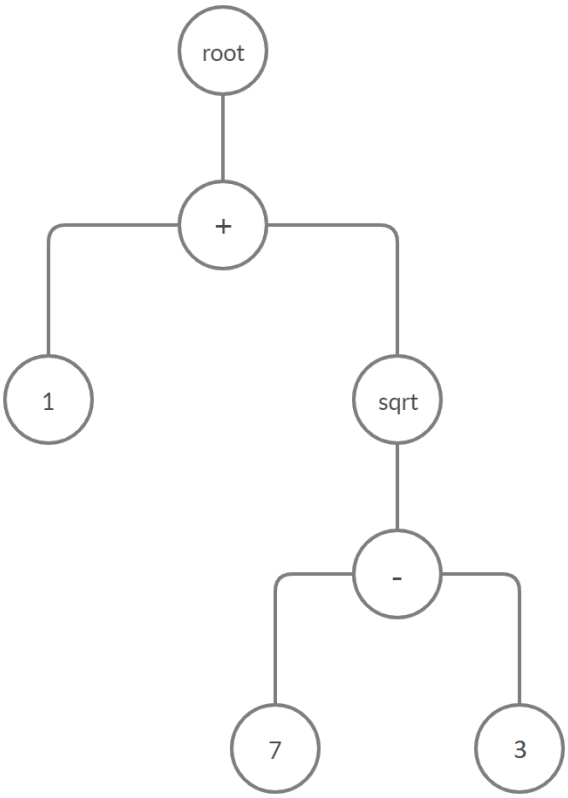
Figure 1: The diagram of a simplistic Lexer that splits a string by whitespace



A Finite State Machine (or Automaton) is a program akin to a directed graph (flow chart), where every node represents a state and every edge represents a transition to another state based on a condition [10] [14]. It usually iterates over a collection and, at every step, the program is in a certain state, which dictates what the next state is depending on the current input of the iteration.

III.10.4 Abstract Syntax Tree

Figure 2: The Abstract Syntax Tree representation for the piece of code ”1 + sqrt(7 - 3)”



An Abstract Syntax Tree is an acyclic graph (sometimes a binary tree) that represents a hierarchical structure for a piece of code. Occasionally, the Parser first outputs a Concrete Syntax Tree which also contains, for example, parentheses [24]. The CST is then modified and simplified down to a more “abstract” form, the AST, where some node trees become expressions and some information is generally lost, such as comments.

IV Implementation and My Contribution

This section describes the process of implementing Waxx. It describes the algorithms used and each one of its compiler phases together with the grammar of Waxx and the command-line interface. I implemented all of Waxx from scratch; 100% of the code was written by me, although I did draw inspiration from other sources occasionally. Nevertheless, Waxx contains no copy/pasted code from other sources.

IV.1 Project Organization and Preliminary Files

This subsection describes the file hierarchy in the Waxx repository on GitHub.

Code IV.1

```
node_modules/..
waxx/
    codemirror/..
    src/
        Collapser.mjs
        Expressions.mjs
        Expressizer.mjs
        Grammar.mjs
        index.mjs
        NullCoalesceExpressizer.mjs
        Outputter.mjs
        Parenthesiser.mjs
        Parser.mjs
        ParserStates.mjs
        Scoper.mjs
        Splitter.mjs
        Utils.mjs
        Words.mjs
        WordTypeAssigner.mjs
    languages/
```

```
        HaxeOutput.mjs
        JSOutput.mjs
        PythonOutput.mjs
    index.html
cli.mjs
package.json
package-lock.json
run-server.mjs
```

IV.1.1 Base Directories

The repository is organized into a simple file hierarchy. The only files at the base of the repository are the "README.md" file and the ".gitignore", which are specific to git. The entirety of the project exists in the "waxx" directory.

This folder contains the required files for a Node.js project, "package.json" and "package-lock.json", which are standard JavaScript module requirements for Node.js, as is the "node_modules" directory. The "node_modules" is generally not held on the GitHub remote repository, but in our case, it's fairly small (120Kb) and helps with running the everything smoothly and without any other setup.

The "run-server.mjs" script is only a JavaScript file that hosts the test website situated inside "waxx-demo". It uses the "connect" and "serve-static" Node.js modules to host the static web page. By default, the web page starts on "http://localhost:8080".

The second file at this level is "cli.mjs" which acts as the command line interface script.

cli.mjs

This is the script called by the user who wants to use the Waxx CLI. It is a straightforward program that does some argument processing (process.argv). The only actions currently supported are compilation and help.

The compilation only reads the given file, compiles it using the Waxx algorithms from the other scripts (explained later) and writes the output file with the same name as the input file, with the extension changed from ".wx", ".waxx" or ".xx" (up to user's choice) to the extension of the target language source code (e.g. ".py").

IV.1.2 The "waxx" Directory

This is another wrapper script over the source files for Waxx. The "index.html" file is the one and only HTML file hosted by "run-server.mjs". The other two directories are "codemirror" and "src".

The "codemirror" directory holds all files for CodeMirror, the library used for integrating a nice text editor inside the "index.html". In our case, CodeMirror functions as a "black box". It does what it needs to do without our concern. It exists only so it makes the project look and feel more professional, but holds no actual value for how Waxx works or operates. It could easily be replaced by a simple <**textarea**> tag.

The "src" folder contains all the source code for the Waxx compiler.

*Note: In modern JavaScript, modules end in ".mjs" to mark that they are modules. The advantage of modules over traditional Node.js "require" function is that modules can be used in browser and offers a much better project structure than importing scripts via the <**script** src="..."> tags.*

IV.2 The Code

All the titles of the following subsections are respective to a JavaScript source code file and assume the base directory for these is "waxx-js/waxx/src".

IV.2.1 index.mjs

This is the main script of the Waxx compiler which calls, one after another, each compiler phase and generates the final output.

Even though it imports many modules, its only code is one function:

Code IV.2

```
export function go(sourceCode, languageString) { ...
```

This function (which gets exported to be used anywhere else) takes a string that contains Waxx source code and a "languageString" flag, which represents the language to compile to (e.g. "js"). Simple enough!

Compiler Phases

Waxx compilation goes through several phases of processing of the base source code and each of them is handled by a different module, each with only one in/out function that takes care of all the process. These phases are separated into 3 categories: preprocessing (A), parsing (B), and outputting (C).

Here are the phases, in exact order:

A. Preprocessing

1. Line Splitting: split string into lines
2. Splitting: split lines into tokens called Words
3. Parenthesising: assign each "pair token" (e.g. parentheses) to its pair for quick reference
4. Collapsing: collapse parentheses, so that parentheses always start and end on the same line (this is a feature of Waxx)
5. Assigning word types: each token is assigned its type based on its string content

The final form of the data after preprocessing is a list of WordLine (explained later).

B. Parsing

1. Expressizing: group up parentheses, brackets and other such constructs so that they are always contained in Expressions; it also encapsulates every line of Words into a base-level Expression
2. Null coalescing expressizing: a special phase required for the null coalescing operator feature of Waxx
3. Scoping: organize the Expressions into a scope hierarchy based on their indentation; returns one single scope, the base scope of the code
4. Parse: the final step of transforming tokens into a syntax tree. Takes care of all else based on a finite state machine modeled after a context-sensitive grammar.

The final form of the data after parsing is a single Scope object (explained later).

C. Outputting: iterates over the tree and outputs code in the target language in the form of a string, using simple configurable "Language" scripts (explained later).

The final form of the data after outputting is a string.

All of these phases are called one after another by the "go" function.

Note: Indeed, all of these phases can be further optimized (quite heavily) and condensed into much fewer steps. However, one of my goals in implementing Waxx was code clarity: there is a clean, straight forward code structure without unwanted side effects. The data always goes from one layer to another until it reaches its final form. Throughout the development of Waxx, I encountered scenarios in which I had to add an extra compiler Phase (such as the obvious Collapsing and Null Coalescing Expressizing). I preferred not to change the main code functionality so far and instead add another compiler Phase. This way, the whole Waxx flow is neat and tidy and massively eases debugging.

IV.2.2 Utils.mjs

This is a light module with useful and very basic functions used throughout the project. These are as follows:

- `capitalize(String) : String` : an elementary functions which takes a string and makes it's first character upper case

- `dashToCamelCase(String) : String` : takes a "dash-case-string" and outputs a "camelCaseString".

- `spaces(int) : String` : returns a string whose only content is that many spaces (this function is used a lot for indentation)

- `isSpace(String) : boolean` : returns true if the given string contains only whitespace

- `isRunningInBrowser() : boolean` : at the time of the writing, there is no standard way to check whether the code is running in the browser or in Node.js. This function tries to assign the "window" global variable to a random variable. If it works, it means we are in the browser and returns true. If it throws an exception, it means we are running on Node.js and instead returns false.

- `doTimes(int, func) : void` : does the function given as an argument a number of times

- `splitArrayByIndicesExclusive(array, indices)` - this is the only more complicated utility function. It takes an array and some indices and splits the array into chunks, separated by those indices, excluding the elements on the indices themselves. It returns an array of arrays. A simple example would be giving as input `[0,1,2,3,4,5,6,7,8,9]` as the array and `[3,6,7]` as the indices. This will produce the following output: `[[0,1,2], [4,5], [8,9]]`.

IV.2.3 Grammar.mjs

This module acts as a namespace for multiple variables and functions. It contains data and logic for identifying token types.

- `getTokenType(String)` : takes a string and returns a token type according to the data described in this Grammar.mjs. The returned token type is always an uppercase string. This function is used when assigning types to Words (explained later).

- `operators` : An array of strings, containing all possible operators in Waxx; this is used only by the Splitter when lexing the source code

- `separators` : an array of pairs of separators, such as for strings or comments

- `flowControlConditions` : an array of keywords for flow control constructs that take a condition (ifs, whiles, etc). If a keyword exists in this array, then it is given the FLOWCONTROL type. Note that, for example, "else" is not a FLOWCONTROL, but is its own type, ELSE, as explained below.

- `accessModifiers` : an array of keywords portraying the allowed access modifiers; in the syntax trees, these modifiers aren't held in their Nodes, but instead other Expressions can have multiple modifiers (explained later).

- `tokenTypeMapping` : a dictionary (JSON) object which holds the type for some specific keywords, such as VAR, CLASS, or ELSE. Also, some operators have specific types that are represented by a string containing that exact operator (such as "=", "[", etc). One interesting tidbit here is that the "constructor" keyword seems to have the "null" type. No, "constructor" does not have a null type, but when the getTokenType function checks this tokenTypeMapping object, it will continue to search other constructs if it finds null. The reason "constructor" needs to be null is because, by default, JavaScript objects have a property called exactly "constructor" and this would break parsing and result in errors.

- `isString` and `isNativeCode` are functions that check the first character of the given string. Native code written between backticks.

If none of the above toke types is found, then the returned token type is ATOM.

IV.2.4 Words.mjs

This is a module that holds 2 base data types for the Preprocessing phases: Word and WordLine.

A Word object as four properties:

- `string` : the actual string content of the word; assigned at initialization

- `pairLine` and `pairWord` : the indices of its pair's positions, pairLine for which WordLine and pairWord for which Word in that WordLine; assigned by Parenthesiser

- `type` : the type of the token which is represented by a string; assigned by WordTypeAssigner

Among these properties, it also has some methods: toString (for debugging only), hasPair (returns true if it has a pair), and getMatchingPair (returns the pairLine and pairWord as an anonymous object).

The other data type in this script is WordLine, which is essentially a wrapper over an array of Word.

It contains three properties:

- `indentation` : an int representing the number of spaces before the content of this line

- `words` : an array of Word

- `lineNumber` : this WordLine's position in the array of WordLines

All of these properties are assigned at initialization.

Aside from them, it also has a few methods: toString and toTypeString (only used for debugging) and getLength, a getter for the length of its words property.

Here is a list of all possible types of Words:

- any of "[", "]", "(", ")", "{", "}", "=", ":", ",", "—", "?.", "."
- STRING
- NATIVECODE
- OPERATOR
- FLOWCONTROL
- MODIFIER
- VAR

- OVERHEAD
- CLASS
- DATA
- FUNC
- YAML
- ELSE
- ATOM

IV.2.5 Expressions.mjs

This module contains yet two more data types used at the Parsing and Outputting phases: Node and Expression. These represent nodes in the syntax trees generated while parsing.

Node

A Node is very similar to a Word from earlier, except that it has no "pair" and always exists as a leaf-node in a syntax tree. Its only two methods are toString (used only for debugging) and clone (which returns an exact clone of the node). A Node has the following properties:

- `content` : the string content of the node, like a Word's "string" property

- `type` : the type of the token which is represented by a string, like Word's "type"

- `isNode` : a constant which is always true, used for ease of development

The types of a Node are the same as the types of a Word.

Expression

An Expression is also a node in the syntax tree, although an Expression generally has multiple child-nodes, which can be either Nodes or other Expressions. Although an Expression's content is always an array, an Expression is not always a non-leaf node, as this array can be empty. An expression has the following properties:

- `parent` : the parent Expression or Scope (detailed later)

- `content` : an array of Node/Expression; this content can be an array of length 0

- `accessModifiers` : an array of strings, representing access modifiers such as “private” or “static”; it can be an empty array

- `type` : the type of the expression

- `isTuple` : a property which is true if the Expression is a tuple

- `isExpression` : a boolean which is always true, used for ease of development

Here is a list of all the types possible Expression types:

- EXPRESSION (unclassified expression)
- PAREXPRESSION (an expression which was given with parentheses)
- INDEXEXPRESSION (an expression which was given with square brackets)
- GENERICEXPRESSION (an expression which was given with curly braces)
- FLOWCONTROLEXPRESSION (an expression given by a FLOWCONTROL Word)
- OVERHEAD (an expression given by an OVERHEAD Word)
- YAMLPROPERTYVALUE (an expression holding two values: the left and the right parts of a YAML key:value pair)
- INLINEIFEXPRESSION (an expression holding elements of an inline-if construct (or the tertiary operator in many Java-like languages)
- VARDECLARATION (the declaration of a variable)
- FUNCDECLARATION (the declaration of a function)
- CLASSDECLARATION (the declaration of a class)
- DATADECLARATION (the declaration of a data-class)
- ATTRIBUTION (the attribution of a variable or tuple)

IV.2.6 Splitter.mjs

This module takes the role of the Lexer in Waxx. Its main algorithm splits a given one-line string into an array of string and returns that array. The class which does the logic is called Lexer. It uses several helper functions:

- `isAnySubstringAt(substrings, string, start)` : this function returns something if the string, from the "start" position on, starts with any of those substrings. If it doesn't, it returns null. If it does, it returns the index of that substring in the substrings array.

- `isOperator(text, position, operators)` : returns something if the text, from that position on, starts with any of the operators given. If it doesn't, returns null. If it does, it returns the operator itself. This function is a more specific wrapper over isAnySubstringAt.

- `startsString(text, position, openSeparators)` : returns something if the text, from that position on, starts with any of the separators given. If it doesn't, returns null. If it does, it returns the operator itself. This function is also a more specific wrapper over isAnySubstringAt.

- `endsString(text, position, closeSeparator)` : if the text, from that position on, starts with that closeSeparator, returns true. Else, returns false.

- `findIndentation(text)` : returns how many spaces are at the beginning of that string. Tabs count as 4 spaces.

The Lexer class is used to instantiate one-time-use objects for splitting text. Its constructor takes in an array of strings, lines, the operators, and the separators (given explicitly from the Grammar.mjs module).

Lexer's most important method is "splitLine()", which parses the next line the Lexer is at.

The returned type of splitLine() must be a WordLine. Whenever we say a "string is pushed", we refer to adding that string to the current WordLine's "words", which is an array of Word objects. First and foremost, the Lexer checks the indentation of the string being processed (the number of spaces it starts with) and sets the current WordLine's "indentation" property accordingly. Then, splitLine() starts at the first non-whitespace character it finds - the character the code line actually begins with after the indentation.

The Lexer works as a finite state machine with three states: "blanks", "words" and "string". Each of these states has a corresponding method in the Lexer: stepBlank(), stepWord(), and stepString() and they always operate on a single character. Thus, a state processes one character from the input string line. For every character in the input string line, one single state is called. There is no state for operators, because, when encountering an operator and finding one that

matches the isOperator function, it pushes the whole operator. This is appropriate because there are no operators more than 2 characters long. When switching states, marks the iterator index position and pushes the token to the list of strings that is returned (if there is anything to push at all). The token is a substring of the base string line being parsed, from the previously marked iterator index to the one currently at. All three Lexer states are analyzed below:

When splitting a line, the automaton always starts at the "blanks" state, which represents a state when it's reading whitespace. When it's in this state:

- Upon finding whitespace, nothing happens
- Upon matching the startsString function mentioned above, it marks the start of parsing a string and switches to the "string" state.
- Upon matching the isOperator function mentioned above, it pushes the whole found operator and the character index iterator (cursor) of the splitting moves in the front. The state does not change.
- If nothing else matched, it marks the start of a word at the position of the cursor and switches to the "words" state.

The "words" state is very similar to the "blanks" state:

- Upon finding whitespace, it pushes the word from the marked position to the cursor position and switches to the "blanks" state.
- Upon matching the startsString or isOperator functions, the process is identical to the "blanks" state, but it also pushes the substring from the previous mark to the cursor's position.
- If nothing else matched, nothing happens and continues

The "string" state is the simplest, as it only has one thing to check:

- If it's at the end of a "string" construct (meaning it starts and ends with quotes, double quotes, backticks, and whatever else Grammar.mjs permits), then it pushes the string (quotes included) and switches to "blanks". Note that for a string to end, it checks whether it's ending in the same type of character it started as (if a string started with double quotes, then the Splitter only ends the string upon encountering double quotes again; it will not end the string upon encountering a single quote).

The parse() method of the Lexer applies the splitLine() method for every string in the list of strings given and remembers the WordLine returned by splitLine(). The final output is the array of all processed WordLines.

IV.2.7 Parenthesiser.mjs

This is a much simpler module. Firstly, the obvious question is why this step is necessary. After multiple iterations and trials and errors with various techniques for Preprocessing phases, the conclusion was that this Parenthesiser phase can avoid lots of troubles in further trials and errors. Alas, this phase remained only as a helper for the next phase, collapsing.

This module only has one function: parenthesize. The pairing up of parentheses is done by iterating over every Word on every WordLine and, upon encountering an open parenthesis, it adds the current position of the iteration cursor (WordLine index and Word index) to a stack (which starts empty). Upon encountering a closing parenthesis, it pops the stack and sets the "pairLine" and "pairWord" of each of them to each other.

The Words mutate while being processed here, but the function returns the original wordLines regardless, for consistency with the rest of the functions.

IV.2.8 WordTypeAssigner.mjs

WordTypeAssigner.mjs contains only one, 7 line long funtion: assignTypesToWordsInWordLines. It iterates over all words in all wordLines and sets their type based using the getTokenType function from Grammar.mjs.

IV.2.9 Expressizer.mjs

The module does the first step of the parsing phases: grouping tokens together into expressions based on parentheses, brackets, and curly braces. The module is divided into two classes: ExpressizerStates and Expressizer, which implements ExpressizerStates. The FSA states are separated from the other non-FSA states for cleanness.

Expressizer takes one single WordLine and outputs one single Expression. The states are remembered in the memory as strings starting with "$" (dollar sign) and each has a corresponding method in ExpressizerStates. For example, the state "$-par-tuple-expression" is handled by the "$parTupleExpression" method. Each state The FSA processes one Word from the line at a time, and a state is called once on the current Word. States will switch from one to another very often.

When parsing a line, first, a new root Expression of type EXPRESSION is created and we consider this our base expression. Then, Words from the WordLine are processed by the FSA one by one and for each Word, a new element is pushed to the contents of the base Expression. The element can be of two types: a normal Node with the contents of the current Word's string and the same type or, if we are at an open parenthesis, bracket, or other, we create a new Expression and push it to the base Expression's content. This newly created Expression becomes the new base Expression of the FSA. When we encounter a closing parenthesis, bracket, etc we go back to the previous Expression, the parent Expression of the base Expression. That previous Expression becomes the new base Expression.

Note: in code, the base Expression is "this.currentExpression". I used the term "base" earlier to describe the process in an easy to understand way.

The general structure of how this FSA works will be straight forward once we explain the essential helper methods of the Expressizer class:

- `push` : pushes what is given as an argument to the root Expression's content

- `branchOut(newExpressionType, newState)` : creates a new Expression with the given type, sets it as the new root expression and changes the state to the given state

- `brateIn()` : sets the current expression to the previous expression and goes back to the state where our last expression left off

- `wrapOver(newExpressionType, nextState)` : Wraps the current Expression's content in a single Expression with the given type. Sets the new Expression as the current Expression and advances the state stack.

Code IV.3

After having these methods, the way the Expressizer's code works is evident. One interesting part, however, is when reaching a comma. That's when the "wrapOver" function is used.

Using the operations described above, the FSA traverses each WordLine and splits it into expressions to make the data ready for the next compiler phase.

IV.2.10 NullCoalesceExpressizer.mjs

This phase was inserted in between Expressizing and Scopifying to allow for the Null Coalescing Operator feature. This phase needed to be separate because it has to do a backtrack at some point, while the implementations of other compiler phases do not backtrack.

The Null Coalescing Operator is "?." and for an "object?.property" it means *if the object is not null, then access the property (like "object.property" commonly).*

The NullCoalesceExpressizer class is initialized with an Expression and, after calling the "parse()" method, returns an Expression, which was the given Expression after being processed.

When parsing it, the NCE iterates over all the elements of the given Expression's content. That element is then recursively processed by the NCE, then added to a new array of Nodes and Expressions. When a normal Node is given to the NCE, it is instead just returned as it is and pushed. When an Expression is given to the NCE, it's processed by another NCE and what that new NCE returned is pushed. However, upon encountering the exact "?." node, the NCE backtracks the current "?." and "." chain to where it starts. It removes all elements found in the backtrack and inserts Expressions and Nodes in their place to match an inline-if as in the following example:

Suppose we have the following line of code:

Code IV.4

```
o myName = dog?.owner?.name
```

After being split by the Lexer and Expressized, the Expression will look like this:

Code IV.5

```
Expression('o', 'myName', '=', 'dog', '?.', 'owner',' ?.', 'name')
```

The NullCoalesceExpressizer iterates through every element in the expression and by the time it reaches the "?." operator, it's new Expresison that is returned looks like this:

Code IV.6

```
newContent('o', 'myName', '=', 'dog')     # Next node will e '?.'
```

When it reaches "?.", it backtracks to all nodes in the back that respect a logical "?." or "." chainig; these nodes are remembered in a variable called backNodes and removed from the newContent.

Code IV.7

```
newContent('o', 'myName', '=')
backNodes('dog')
```

The newContent is then completed with a structure of an inline-if, where, in this example, E represents a wrapper expression over the backNodes with an extra null equality condition:

Code IV.8

```
newContent('o', 'myName', '=', 'if', E('dog', '==', 'null'), ':',
           'null', 'else')
```

The NCE continues to iterate over the elements of the Expression until it finds another "?." sign. The elements found during the backtrack are ('dog', '?.', 'owner') and the "?." signs will be replaced by normal "." signs, so that backNodes will look like `backNodes('dog', '.', 'owner')`. The same process is done as before. The final result of this example is:

Code IV.9

```
newContent('o', 'myName', '=', 'if', E('dog', '==', 'null'), ':',
           'null', 'else', 'if',
           E('dog', '.', 'owner', '==', 'null'),
           ':', 'null', 'else', 'dog', '.', 'owner', '.', 'name')
```

After this compiler phase is done, the data is ready to be processed by the Scoper.

IV.2.11 Scoper.mjs

The Scoper's job is to organize the root Expressions we have so far into a hierarchy based on the indentation of their original lines. By root Expressions, we understand that it's the Expression resulted from parsing a WordLine earlier. Luckily, our program has kept track of the indentation of root Expression with respect to their original WordLine!

Let us describe a Scope object first by looking at the Scope class. Its constructor takes as arguments the parent (parent Scope), the root Expression, the "content" which refers to child scopes and, as the final argument, the indentation, which is self-explanatory. For ease of debugging and cleanness, a Scope has a "type" property which is always equal to "SCOPE".

The logic of this module is fairly simple: there is only one function that gets exported, namely `scopify(expressionsWithIndentation)` which takes an array of pairs of Expression and indentation in the form of anonymous (JSON) objects. The function first creates a base Scope with a null parent and -1 indentation and marks it as the current base Scope. It iterates through the given Expressions and, when it finds an Expression with a

higher indentation, it wraps it in a Scope and adds the newly created Scope as a child to the current base Scope and changes the current base Scope to this one. When it encounters an Expression with lower indentation, it goes back from parent to parent until it finds a parent Scope with higher or equal indentation, then sets that found Scope as the current base Scope and continues by checking again whether this Expression's indentation is higher or equal to the current base Scope. Upon encountering an Expression with equal indentation, it creates a new Scope wrapping the expression, adds it to the content of the current base Scope's parent Scope, and sets the current base Scope to this new Scope.

The result is a single Scope containing every other Scope processed by the Scoper and the data is ready for the final and most complex Parsing compiler phase: parsing.

IV.2.12 Parser.mjs and ParserStates.mjs

The Parser class from Parser.mjs is also a Finite State Machine, very similar to the Expressizer one, although here, its states are separated into another module called ParserStates.mjs because there are many more states than before. The Parser takes an Expression, a starting state, and an "isYAML" flag (explained later). The output of the parser is, of course, an Expression.

The Parser iterates through the elements of the given Expression's content and, based on the states, it pushes to a new array of elements Expressions and Nodes in (sometimes heavily) altered structures. An important note is that, whenever the Parser encounters an Expression, it first parses that Expression with a new Parser recursively, then uses it for whatever needs to be done.

Describing all the Parser states individually would prove impractical, as there are too many states and too many possibilities, but also because the states are manually modeled after the Waxx's grammar productions described below and that information should be enough to understand the flow of data within the Parser and its states.

The Parser uses a handful of helper operations (methods) for smooth sailing. Three of these operations are identical to the ones in the Expressizer, namely "branchOut", "brateIn" and "wrapOver". Aside from these, there is one more operation: "redirectToState". The purpose of this is to reuse code and states so that, when "redirectToState" is called, the state switches to the given state, and the whole step is done again. This pattern is very frequently used in the Parser states.

Each state can accept certain Node types and reject others. When a Node is rejected, it means a syntax error was produced by the programmer in the given source code (e.g. a sequence like "VAR CLASS" will stop the Parser and

give an error, because the “VAR CLASS” construct does not exist).

Waxx’s Grammar

Here are all the productions of Waxx’s grammar, separated into three categories: Expressizer productions, Null coalescing productions, and Parser productions. The Parser productions are the only ones happening in the Parser; the other two are implemented by the Expressizer and the Null Coalesce Expressizer, respectively.

Expressizer productions (parentheses, brackets, and braces are taken literally):

Code IV.10

```
_                       <- <any token or expression>
ATOM                    <- ATOM
PAREXPRESSION           <- ( _ )
PAREXPRESSION tuple     <- ( _* , _* , ... )
INDEXEXPRESSION         <- [ _ ]
INDEXEXPRESSION tuple   <- [ _* , _* , ... ]
GENERICEXPRESSION       <- { _* }
GENERICEXPRESSION tuple <- { _* , _* , ... }
```

Null coalescing productions (parentheses and brackets are taken as delimiters for regular expressions):

Code IV.11

```
FLOWCONTROL AUX == null : null ELSE AUX . _ <- AUX ?. _
    Where AUX1 is: [ATOM] [[.] PAREXPRESSION | INDEXEXPRESSION]*
```

Parsing productions (parentheses and brackets are taken as delimiters for regular expressions):

Code IV.12

```
ATTRIBUTION             <- _* = _*
PAREXPRESSION           <- | _*
VARDECLARATION          <- [MODIFIER]* VAR _
CLASSDECLARATION        <- [MODIFIER]* CLASS [GENERICEXPRESSION] ATOM
FUNCDECLARATION         <- [MODIFIER]* FUNC [GENERICEXPRESSION]
                                [:] ATOM PAREXPRESSION [:]
```

```
DATADECLARATION          <- [MODIFIER]* DATA ATOM _*
INLINEIFEXPRESSION       <- FLOWCONTROL _* :  _* ELSE _*
FLOWCONTROLEXPRESSION    <- FLOWCONTROL _* [:]
OVERHEAD expression      <- OVERHEAD STRING
lambda                   <- YAML :
    YAMLPROPERTYVALUE    <- _* : _*    (only inside YAML scopes)
```

The Parser.mjs module has the "parseScope" function which is given the base Scope and the Parser module takes care of it. The base Scope is mutated and, with parsing concluded, the data is set up as a final and complete syntax tree and it is ready for the final compilation phase: outputting.

IV.2.13 Outputter.mjs and the "languages" Folder

The Outputter module's primary function is "outputNode" which takes a node, the parent scope of that node (even if it's not directly the child of that Scope; it's ok as long as that Scope is the closest Scope parent of the node) and the target language (defined in one of the language modules in the "languages" folder, currently JavaScript, Python, and Haxe). It also has an "options" argument for extra options.

On top of the "outputNode" function is wrapped the "outputScope" function, which takes a Scope and a language (like above) as parameters. Also, the "outputNode" is a wrapper function over the logic of the Outputter class.

The "outputScope" function is recursive: not only will it call "outputNode" on the root Expression of the given Scope, but it will also call "outputScope" on all child Scopes. The same pattern of recursion goes for the "outputNode" function, which, if given an Expression, will call "outputNode" recursively on every element of its content.

When we have everything set up as nested Expressions with all the information we need, it becomes trivial to take the contents of an Expression and process them to output source code in the target language. Every module in the "languages" folder acts as an interface: they are all required to implement certain functions that take a specific type of Expression or construct and output code. These function implementations act as a sort of reverse Grammars. Instead of accepting or rejecting a construct, it creates a final Terminal from multiple Nonterminals.

Firstly, every language module has a "macros" object which holds a simple dictionary with string keys and string values. When a node is being processed by the Outputter, it uses that language's "macros" property to see if that exact string becomes something else in the target language. Very basic examples include "o" becoming "let" in JavaScript, and "my" becoming "self." in Python.

The first thing that is used from a language module is the “outputScopeLine” function, which takes an indentation parameter, the full final string line, the “hasChildren” flag and the parent Scope of the original Expression (the whole function returns a string - the final output of that line). The job of this function implementation is to, well, say how that line is processed. Pretty much always, inside the function. the indentation is translated to that many spaces, then the scopeLine is appended. In the case of Haxe, if the given Scope has children, it will also add a “{” at the end of the line and, if it has no children, it will add a “;”. In the case of Python, a “:” sign is added at the end. Very configurable!

Hand in hand with the “outputScopeLine” function, there exists an “endScope” function the language module implementation which tells the Outputter what to do when a scope ends. In the cases of Haxe and JavaScript, it just writes a “}” sign (with correct indentation). Python is lucky in this regard: the function outputs nothing!

Here is the list of all required functions implemented by a language module:

- `getDataDeclaration: outputs what a ``data class'' looks like`
- `getOverhead: handles overhead`
- `getFunctionDeclaration`
- `getVarDeclaration`
- `getClassDeclaration`
- `getFlowControlExpression`
- `getYAMLExpression`
- `outputScopeLine`
- `endScope`
- `getInlineIfExpression`
- `macros (dictionary object)`

And that’s it! The final output of all of these compiler phases, one after another, is a string containing correct source code in the target language. Currently, the only implemented language modules are “HaxeOutput.mjs”, “JSOutput.mjs” and “PythonOutput.mjs”, but, at the time, a language module only has around 100 lines of code, so more languages can be easily added.

IV.2.14 Implementation Conclusions

This closes the implementation analysis of the Waxx compiler. To conclude, every compiler Phase makes use of a Finite State Machine developed over a Grammar, which is a set of guidelines and rules. Multiple phases (Parser, Scoper, and Outputter) make heavy use of recursion to avoid unnecessary complications and stacks on their state machines. Due to how the Outputter and language modules are implemented, Waxx can support an unlimited number of target languages and each only takes little code to setup. It is also easy to add new features to Waxx, as we can insert another layer (compiler phase) in between two compilation steps.

IV.3 Other Mentions on Implementation

This subsection tells about other notable implementation points, as seen from my perspective. It contains information about what I tried and did not work, the scalability of the project, what other implementation options I had, what I learned, and what can be improved and done in the future.

IV.3.1 What I Tried and Did Not Work

At first, I tried implementing the language in Python. However, I noticed that I used anonymous objects very frequently throughout the project and had no way to clearly visualize the debugging process in Python. On the other hand, modern browsers have built-in debugging, meaning JSON viewers and running code directly from the console while the application is running. Unfortunately, objects in Python are not always serializable and viewable as concrete JSON objects. Another reason to use JavaScript over Python is in-browser compiling (on the front-end). If I had decided to carry on with Python, the application would have used a back-end part written in Python which would communicate with the front-end at all times, thus adding an additional layer of complexity. Therefore, I switched to JavaScript from Python during the first half of the implementation. The transition was smooth since I am already proficient in both Python and JavaScript. From then on, debugging was much easier.

Another thing I tried to do at the start is adhering to pure functions, meaning using only functions that take input and output something, without mutating any outside data during the process. I still find this design pattern very useful and desirable, but, unfortunately, it did not work with the implementation of a Compiler, as it not only made code more difficult to write (requiring constant use of stacks and backtracking), but it also made it slower at run-time, because every time a function had to return an object, it had to create the new object from scratch. There are currently 10 compiler phases and that means creating 10 times more objects than if it could mutate objects freely. This is, of course,

worst-case scenario, since some compiler phases are, indeed, pure. But for the ones that aren't, I am glad I switched early on and avoided some massive headaches. The lesson here was that pure functions aren't always the best functions. Sometimes, the naive solution is the better solution.

IV.3.2 Is It Scalable?

Waxx is absolutely scalable! Waxx is very modular, and other compiler phases and languages can easily be added. It adheres to the Open-Close Principle, meaning classes can be extended but not modified, and also to the Liskov Substitution Principle, meaning Nodes and Expressions can be used interchangeably. There are many optimization opportunities in the Waxx compiler and that also makes it scalable.

IV.3.3 Why Not Lex/YACC

Lex and YACC (Yet Another Compiler Compiler, also known as Bison) are two commonly used tools for making programming languages. Lex takes a series of regular expressions and respective functions (in C) and outputs a Lexer ("scanner") which is a C source code file called, by default, "lex.yy.c".

For splitting code into tokens, Lex generates a Finite State Machine, very much like the one implemented by Waxx.

As explained in a previous section, the Lexer of Waxx has a fairly simple implementation and Lex does not give enough control over the whole process to the programmer. In addition, my language of choice for Waxx, JavaScript, does not support an official Lex port, and other options are discontinued (JS-Lex and lex) or too unreliable for my preference. Therefore, I resorted to writing my own Lexer from scratch.

YACC is a tool for parsing tokens into syntax trees and interpret or output code in another language [27]. YACC is not supported for JavaScript, but there are alternatives, such as Jison or ANTLR, which can generate a JavaScript parser. In Waxx, all compiler phases are clearly separated, so I chose to write my own parser for easy debugging and the ability to modify it, but also, most importantly, for the practice of actually writing a parser from scratch.

V Waxx Guidelines and Documentation

This section acts as the Waxx documentation. It contains a subsection for each step of installation and each feature of Waxx. Firstly, the syntax of Waxx is briefly described to give a heads up for what to expect. Then, the Install the Compiler and Configure the Text Editor subsections describe, in-depth, how to setup Waxx so that it runs properly on Windows 10. Following that is a dive into how to compile Waxx (Compiling Waxx to a Target Language), then the high-level constructs and the syntax sugar of Waxx are described, each in its subsection.

V.1 Install the Compiler

The first step for installing Waxx is cloning the project to the user's computer. This can be done by going to the link in the appendix [A] and clicking "Clone", then "Download ZIP" and extracting the archive afterward. If git is installed on the machine, one can simply do "git clone https://github.com/daverave1212/Waxx" and the download will happen in the currently open folder in the terminal.

To setup further, Node.js needs to be installed. For that, one can go to the official Node.js website referenced in the appendix [D] and download the Windows binaries. At the time of writing this, the latest version of Node.js is "14.4.0", but the latest stable release is "12.18.1 LTS". The latest version of Node is required, and Waxx will NOT work on "12.18.1". The version of Node must be at least 14. The process of installing Node is straightforward.

After installing Node, open a terminal and navigate to the "waxx-js" folder, where the "package.json" file sits (trick: on Windows 10, this can be done by typing "cmd" in the address bar of the file explorer). Run the "npm install" command there. After finalizing, everything should be good to run.

Note: if any of those steps fail, starting the terminal as Administrator might help solve some issues.

V.2 Configure the Text Editor

The only supported text editors at the moment of writing this are Notepad++ and the online demo version of Waxx. In the "extras" folder there is the "WaxxNPP.xml" file which can be imported into Notepad++.

1. In Notepad++, go to "Languages" at the top, then click on "Define your language..."
2. Click "Import", select "WaxxNPP.xml" and click "Open"

3. When working on a Waxx source file, go to “Languages” and select “WaxxNPP” from the bottom

Notepad++ should now be configured to work with Waxx! Additionally, you can type “waxx” (or whichever extension you want to use for Waxx source code) in the “Languages”, then “Define your language...”, then “WaxxNPP” in the “Ext.” text box upright.

V.3 Syntax of Waxx

Waxx syntax tries to be as non-mathematical as possible and uses words instead of symbols wherever it can. Furthermore, Waxx tries to drop parentheses whenever possible, as well as colons and semicolons. Semicolons are a no-no scenario in Waxx, so they were designed to delimit multiline comments (although not yet a feature).

That said, the way to imagine what Waxx looks like is to imagine Python, but with even less punctuation. That also means no traditional JSON notation, which is replaced by the YAML notation in Waxx.

Very importantly, whitespace (indentation) is used to delimit scopes - no curly braces allowed!

V.4 Compiling Waxx to a Target Language

When using the online demo compiler, this is as easy as typing the code in the text area, selecting the preferred language from the dropdown menu on the right, and clicking “Go”. The code is translated to the target language and, optionally, run directly in the browser if the target language is JavaScript.

With the command line interface, the base command is `waxx` if the language was installed fully and added to the windows PATH variable, or `node cli.mjs` otherwise. To compile a “.waxx” file to another language, just type `waxx <language> <filepath>` (or `node cli.mjs <language> <filepath>` respectively).

By typing “waxx help”, a helper will show up with more details:

Code V.1

```
Argument #1: the language [js | py | hx | (javascript | python | haxe)]
Argument #2: the path to your waxx source file [_.wx | _.waxx | _.xx]
```

Just like in the instructions, typing `waxx js myFile.wx` will compile “myFile.wx“ to “myFile.js” in the same folder as “myFile.wx”.

V.5 Coding Style and Guidelines

When writing Waxx code, follow the naming conventions of the target language (e.g. camelCase for JavaScript, snake_case for Python, etc). Also, by default, the size of a tab is equal to 4 spaces and that is important for the compiler to work properly. Whenever possible, the programmer should try to limit themselves to 80 characters per line of code.

V.6 Waxx Syntax

This subsection describes the basics of Waxx syntax, like declaring variables, using flow control, and defining functions and classes. It then goes on to describe the various features of Waxx such as data classes, null coalescing operator, inline-if's, the pipeline operator, and YAML object notation.

Variable Declaration

To declare a variable in Waxx, use the "o" keyword (coming from "object", but one should see it as a little circle instead of a letter).

Code V.2

```
o name = "Daisy"
```

This is equivalent to "let" in JavaScript, "var" in Haxe, and nothing in Python. Haxe uses type inference to tell the type of a variable. However, if the type at declaration is needed, it can be given with a colon, Kotlin style:

Code V.3

```
o name : String = "Daisy"
```

Flow Control

Waxx supports the regular flow control blocks most programming languages have: if/else, switch, while, and for. For all of these, parentheses are not required. As Waxx delimits scopes with indentation, an if/else block looks like this:

Code V.4

```
if name == "Daisy":
    console.log('Yes,')
    console.log('My name is Daisy')
else:
    console.log('No.')
```

This translates to JavaScript as:

Code V.5

```
if (name == 'Daisy') {
    console.log ('Yes,')
    console.log ('My name is Daisy')
}
else : {
    console.log ('No.')
}
```

Two observations about scopes: colons (":") are optional when delimiting a scope. Also, several keywords are preferred over traditional operators such as "==". Here is the previous example program is written with those features in mind:

Code V.6

```
if name is 'Daisy'
    console.log('Yes,')
    console.log('My name is Daisy')
else
    console.log('No.')
```

The "==" was replaced by the "is" keyword and the colons were dropped to make the code more readable. To join together multiple conditions, we use the "and" and "or" operators instead of "" and "——".

Code V.7

```
if name is 'Daisy' and age is 13
    # Do code
```

The “while” block looks identical to the if block, and, generally, so does the “switch” block, although “break” may be required at the end of a case. The for block, however, is specific to the target language. Although the traditional C-like for with 3 statements are impossible in Waxx, most languages nowadays support “for each” features which can be used instead.

Code V.8

```
for element in my_array     # Python and Haxe
for o element of myArray    # JavaScript
```

Functions

Functions are defined with the “func” keyword and no return type. The type of the returned value is inferred by the target language compiler (in the case of Haxe). Unlike variables, functions are always type inferred, and explicitly giving the return type of a function is never required. Thus, declaring the return type is not possible in Waxx.

Code V.9

```
func sayHello()
    console.log('Hello!')
```

But, with the possibility of adding more languages to the output of Waxx in mind, explicit function types are work-in-progress.

Classes

Classes are somewhat more standardized in Waxx. They follow mostly the same structure, no matter the target language. Classes do support generics, but they are explained later in a section dedicated to generics.

Code V.10

```
class Dog
    o name
    o age = 0
```

```
    func bark()
        alert('Woof!')
```

To JavaScript, this compiles to:

Code V.11

```
class Dog {
    name
    age = 0
    bark() {
        alert ('Woof!')
    }
}
```

In Python, however, the compiled code looks like this:

Code V.12

```
class Dog:
    name
    age = 0

    def bark(self):
        alert ('Woof!')
```

Notice that the “self” argument is automatically inserted in Python code. With this said, Waxx methods and properties support various access modifiers, such as “static”, which are automatically taken care of by the Waxx compiler and outputted as their correct equivalent in the target language:

Code V.13

```
    def bark():          # From "static func bark()"
        alert ('Woof!')
```

When an access modifier does not exist in the target language, an exception is thrown and a message printed to the screen. Also, in Waxx, there is no “this.” or “self.”. Instead, there is the “my” keyword which does the same thing:

Code V.14

```
my name = 'Daisy'   # In JS, "this.name = 'Daisy'"
```

Constructors are not language-agnostic. Each constructor must be created in a way specific to the target language:

Code V.15

```
    func constructor(...)   # For JavaScript
    func __init__(...)      # For Python (without 'self' argument)
    func new(...)           # For Haxe
```

At the moment, inheritance is not supported, but it is a high priority in the “to do” list.

Generics

Generics are specified through curly braces, because there was no other use for them in the language and it’s cleaner to have specific symbols for generics instead of the “less” or “more” operators. Generics are usable not only for classes, but also for functions (although no currently supported language has function generics, but it’s made with future in mind):

Code V.16

```
o dogs : Array{Dog}


class {T} MyCollection
    # Code
```

The Pipe Operator

The Pipe Operator serves as a way to reduce the number of parentheses and make chaining functions much smoother. The Pipe (“|”) works as an open parenthesis, but it does not need to be closed, as it takes everything until the end of the current expression as part of the “parenthesis” closure:

Code V.17

```
print | Math.sqrt | x   # Same as: print(Math.sqrt(x))
sum(Math.sqrt | 4, 2)   # Same as: sum(Math.sqrt(4), 2)
```

Notice how, if enclosed in another function which could take more than 1 argument and is called with parentheses ("sum" in the above example), the Pipe operator goes until the first comma it finds on the same level. However, if the Pipe is used on the base level and inside no other parentheses but only other Pipe operators chaining, then it will consider everything to the right as part of the Pipe expression (like in the "print" example above).

Inline If

Waxx supports the inline-if construct, equivalent to the ternary operator in C-like languages. As Waxx adheres to words instead of symbols, the syntax changes slightly from the usual ternary operator use. In the Python version of the inline-if,

Code V.18

```
o owner = if dog == 'Daisy': 'David' else 'Someone else'
# Same as:
o owner = (dog == 'Daisy') ? 'David' : 'Someone else'
```

Null Coalescing Operator

This is also called "Null Conditional Operator" in C# or "Optional Chaining Operator" in JavaScript. It is another way of accessing the property or method of an object, but only if that object is not null. Traditionally, trying to access the property of a null object results in a runtime exception. That's there the NCO comes in: it accesses that property only if the object is not null. If the object is null, it returns all the property accessing chain as null. Here's an example:

Code V.19

```
o name = dog?.name
# Same as saying
o name = if dog is null: null else dog.name
```

The NCO can be chained through multiple property accesses and it will work as expected in every scope where the syntax is correct:

Code V.20

```
o name = dog?.owner?.grandma?.name
```

The name will be “dog.owner.grandma.name“ if all of those 3 first objects are valid. If any of “dog”, “owner” or “grandma” is null, then name will be null.

Data Classes

A data class is a type of “one-liner” class where all properties are public by default and it has some useful functions built-in, such as “clone()”.

Code V.21

```
data Dog(name : String, age : Int = 0, isMale : Boolean = false)
```

In JavaScript, it is easy to work with anonymous objects, but this is not so trivial in non-interpreted languages, where proper data structures must exist. Hence the “data” class type. All data classes act exactly as normal classes, it’s just their definition that gets implemented in the background without the programmer writing any of the boilerplate code. The code above gets transpiled to the following structure in JavaScript:

Code V.22

```
class Dog {
    name
    age = 0
    isMale = false
    clone() {
        let __clone = new Dog()
        __clone.name = this.name
        __clone.age = this.age
        __clone.isMale = this.isMale
        return __clone
    }
}
```

At the moment, a data class automatically implements only the “clone” function, but more are planned for the future.

YAML

Most languages nowadays support anonymous objects and explicit maps with String keys. The usual notation for these objects is the JSON notation (in all three JavaScript, Python, and Haxe). The JSON notation delimits scopes with curly braces and separates properties with commas, which Waxx does not like. Thus, Waxx uses a different method for defining anonymous objects: YAML. YAML stands for “YAML Ain’t Markup Language” and looks similar to JSON, except that scopes are delimited with whitespace and no commas are required to separate properties. This makes YAML the perfect candidate for a less noisy anonymous object notation!

To declare a YAML object, the “yaml” keyword is used:

Code V.23

```
o dog = yaml:
    name: 'Daisy'
    owner:
        name: 'David'
        age: 22
    home: 'Romania'
```

For JavaScript, this gets translated to:

Code V.24

```
let dog =  {
    name: 'Daisy',
    owner: {
        name: 'David',
        age: 22,
    },
    home: 'Romania',
}
```

That wraps up all the Waxx syntax so far and is all a programmer needs to write a full program in Waxx. There are

many other features planned for the future, as I intend to continue the development of the Waxx compiler after finishing this thesis.

VI Conclusions

Waxx is an ambitious project and its development was not free of hardships. With the help of good references, articles, and online tutorials, Waxx is developed successfully and aims to be a legitimate and useful programming tool. Waxx will continue being developed and updated even after this paper is done and the project submitted since it was an ambition as much as it was a passion project.

VI.1 Disadvantages

The disadvantages of writing Waxx code instead of native code is that, even though Waxx adds features, it might also strip away features, since the code you write needs to be somewhat compatible with other target languages. Another disadvantage is the lack of community support like there is for CoffeeScript. This is mitigated, however, due to Waxx having only syntax and no semantics and because Waxx uses target language-specific functions and types, so that community support would have to be only for Waxx syntax constructs.

Waxx is also in its early stages and not particularly fast at run-time, although it does have the potential to be optimized for very fast running and scalability.

All in all, Waxx is a powerful programming language and a complex project, so not all bugs might have been caught during its development. Heavy testing is a domain of its own and requires either good testing tools or certain manpower to test and make sure Waxx runs as it's supposed to. That being said, Waxx also has no support for Linux and Mac and is not guaranteed to work on older versions of Windows (below Windows 7). It should be rather easy to port Waxx to other platforms, since it runs on the cross-platform JS engine, Node.js.

VI.2 Lessons Learned

To quickly go over the things I learned from this project, they include:

- Pure functions are not always the solution; mutating objects is okay for the sake of speed and cleanness. - Code modularity is extremely important. I already knew this going into the project, but I was still surprised at the end by how easy it was to add new features and target languages! - Many things about how programming languages work - all the research, trials, and errors resulted in a much better understanding of the subject. I am now way better informed and in tune with the tools and algorithms used to develop programming languages.

VI.3 Possible Improvements

There are limitless possibilities of improvement for the Waxx compiler as it currently is (without the addition of extra features). Run-time performance can be drastically improved by condensing some compilation layers into a single one, especially the Preprocessing phases. During my research, I also learned how to efficiently assign types to tokens in one single iteration, in exactly O(n). This can be done by creating a large keyword/operator tree, where each node contains one letter. For example, the word “catch” can be remembered as a series of nodes in that tree. The word “case” can be remembered using some of the same nodes used for the word “catch”:

Code VI.1

```
c -> a -> t -> c -> h
       -> s -> e
```

Once we reach the “h” letter and then a whitespace character follows, we know for sure that our substring so far, “catch” is a keyword. If it were, for example, “catchy”, there’s no other node after “h”, so we would know for sure it is an ATOM.

Another improvement to be done, again, consists of feeding tokens one by one from layer to layer. That way we don’t have to keep in memory unnecessary lists and we also don’t have to do multiple iterations. This would, however, imply changing the architecture of the compiler phases, especially the NullCoalesceExpressizer.mjs module, since it is required to backtrack a little, thus affecting the perfect O(n) performance. Nevertheless, I am sure an optimal solution also exists for the NCE.

Other improvements include replacing constants and token types with Ints instead of using Strings. It would make everything faster at runtime.

Another obvious possible improvement is writing the compiler in a more runtime-efficient language, such as C++, Go or Rust, but this is further ahead in the future.

VI.4 Future Work

There are quite a few features that were planned for the finalization of the Waxx project but were scrapped due to the lack of time. These features include better string processing, such as accepting escaped string characters, multiline strings, and even string interpolation (writing ”Hello $worldName!” instead of ”Hello ” + worldName + ”!”).

Waxx currently needs a real comment processing feature and better property accessing expressions, which would avoid backtracking at the Null Coalesce Expressizing phase.

Smarter “for” loops were also planned for release, but scrapped, along with “try” blocks without “catch”, the possibility of making domain-specific languages like Kotlin and Nim do with their scope blocks. Also inline functions (lambdas), function type checking, array/list comprehension, and one-liner for/while constructs. Alas, a lot of work was done, but there is still a long road ahead.

References

[1] Ager, Mads & Biernacki, Dariusz & Danvy, Olivier & Midtgaard, Jan. (2003). *From Interpreter to Compiler and Virtual Machine: A Functional Derivation. BRICS Report Series*. 10. 10.7146/brics.v10i14.21784.

[2] Baweja, Chaitanya. *Contemplating the Zen of Python*. https://medium.com/better-programming/contemplating-the-zen-of-python-186722b833e5.

[3] Bonita Sharif, Jonathan I Maletic. *An Eye Tracking Study on camelCase and under_score Identifier Styles*. Department of Computer Science Kent State University Kent, Ohio 44242.

[4] Brown, William Henry, Raphael C. Malveau, Hays W. McCormick and Thomas J. Mowbray. *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*.

[5] Burnham, Trevor. *CoffeeScript, Accelerated JavaScript Development*. The Pragmatic Programmers.

[6] D. Goswami and K. V. Krishna. *Formal Languages and Automata Theory*.

[7] Donghoon Kim, Gangman Yi. *Measuring Syntactic Sugar Usage in Programming Languages: An Empirical Study of C# and Java Projects* Department of Computer Science, North Carolina State University, Raleigh, NC, USA & Department of Computer Science & Engineering, Gangneung-Wonju National University, Wonju, South Korea.

[8] Heller, Martin. *What is Kotlin? The Java alternative explained*. https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html.

[9] JCGs (Java Code Geeks). *Groovy Programming Cookbook*. Exelixis Media P.C., 2016.

[10] Keller, Robert M. *Computer Science: Abstraction to Implementation*. Harvard Mudd College.

[11] K. R. Srinath. *Python – The Fastest Growing Programming Language*. International Research Journal of Engineering and Technology (IRJET).

[12] loup-vaillant. *What is good code?*. http://loup-vaillant.fr/articles/good-code.

[13] Matuszek, David. *Definition of Unrestricted Grammars*. https://www.seas.upenn.edu/ cit596/notes/dave/ungram1.html.

[14] Mordechai Ben-Ari, Francesco Mondada. *Finite State Machines*. 10.1007/978-3-319-62533-1_4.

[15] Niemann, Thomas. *A Guide to Lex & YACC*. e-papers Information Online.

[16] P. J. Landin. *The mechanical evaluation of expressions*.

[17] Peters, Tim. *The Zen of Python*.
`https://www.python.org/dev/peps/pep-0020/`

[18] Picheta, Dominik. *Nim in Action*. 2017 Manning Publications.

[19] Ronald E. Prather. *Structured Turing Machines*. Department of Mathematics, University of Denver, Denver, Colorado 80208.

[20] Scharhag, Michael. *Reduce Boilerplate Code in your Java applications with Project Lombok*.
`https://dzone.com/articles/reduce-boilerplate-code-your`.

[21] Schmidt, David A. *Programming Language Semantics*. Department of Computing and Information Sciences Kansas State University.

[22] Sengstacke, Peleke. *JavaScript Transpilers: What They Are & Why We Need Them*.
`https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them`.

[23] Sheikh, Ghazala & Islam, Noman. (2016). *A qualitative study of major programming languages: teaching programming languages to computer science students*. International Journal of Information and Communication Technology.

[24] Srivastava, Harshit & Pranjal. *Create a NEW programming Language from scratch*. Udemy.

[25] Tao Jiang, Ming Li. *Formal Grammars and Languages*. Department of Computer Science, McMaster University, Department of Computer Science, University of Waterloo.

[26] Verhagen, Bart. *Designing costless abstractions*.

[27] Johnson, Stephen C. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[28] *dollar sign*.
`https://typeclasses.com/featured/dollar`. Typeclass Consulting, LLC.

[29] *Difference Between Syntax and Semantics*.
https://techdifferences.com/difference-between-syntax-and-semantics.html.

[30] *Haxe Manual, Macros*.
https://haxe.org/manual/macro.html.

[31] *Haxe Manual, Map*.
https://haxe.org/manual/std-Map.html.

[32] *Haxe Manual, String Interpolation*.
https://haxe.org/manual/lf-string-interpolation.html.

[33] *What programming language has the best syntax?*.
https://www.slant.co/topics/16402/ what-programming-language-has-the-best-syntax. May 2020.

[34] *Optional chaining (?.)*.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining. May 2020.

[35] *Plain English Programming*.
https://osmosianplainenglishprogramming.blog/2018/05/16/the-journey-begins/.

[36] *WurstScript Manual*.
https://wurstlang.org/manual.html.

[37] http://python.org.

[38] Mathworks. *range*.
https://uk.mathworks.com/help/stats/range.html.

[39] Microsoft. ?? *and* ?? = *operators (C# reference)*. May 2020.

[40] tutorialspoint. *Chomsky Classification of Grammars*.
https://www.tutorialspoint.com/automata_theory/chomsky_classification_of_grammars.htm.

[41] tutorialspoint. *Compiler Design - Lexical Analysis*.
https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm.

A Waxx Github Repository

The source code for Waxx can be found at this link:

`http://github.com/daverave1212/Waxx`

B Online Haxe Compiler

The Haxe online demo compiler can be found at this link:

`https://try.haxe.org/`

C Reddit Discussion

The discussion I held on reddit.com can be found at this link:

`https://www.reddit.com/r/ProgrammingLanguages/comments/exx5bl/what_modern_syntax_-sugar_do_languages_really_need/`

D Official Node.js Website

Node.js can be downloaded at this link:

`https://nodejs.org/en/`