Maturitätsarbeit HS 2022/23

Nylium – Writing my own Compiler

Written by Nieswand Lukas

09.01.2023

Contents

Introduction

This project originates from the idea to create my own programming language called Nylium. In this paper, I want to document the development of the Nylium language and its compiler. In the first part, we will look at the basic concept of a programming language and what criteria it has to fulfill. In the second part, I will shortly explain how a simplified Central Processing Unit, or short CPU, works. Finally, I will show how I implemented a compiler and how to use my programming language Nylium. The Translation is purely theoretical as of January 9th, 2023.

What are programming languages and compilers?

Programming is telling a CPU what to do. As everything in computers is stored in bytes, this information is provided to the CPU in bytes as well. Writing or reading these bytes as a human is impractical since we learned to communicate using words and punctuation marks, which form meaning through a sentence. To make up for the fact that we can barely understand the byte code that the CPU needs, we use programming languages which use parts of our common language. Since the CPU does not understand our made-up languages, we need a translator which produces the byte code the CPU understands. Such a translator is called a compiler. Compilers translate from the code of one programming language to machine code, byte code or another programming language.[1]

Runtime is an important term to describe a state in the lifetime of a program.
It is the time span during which the operations are executed. In some languages, the runtime is another program that enables the primary program to run, since it cannot run on its own. It is complementary to the compilation. Thinking about functionality helps with creating a working programming language.[2]

Designing a Programming Language:

Words & Punctuation:

Matching our common language, we ideally want to design our language based on text.
Text can be split into two kinds of elements: Words and special characters, where the latter are usually referred to as punctuation marks. Processing text in this way is called parsing. Not all languages follow this concept, for example, Piet[3]. However, none of these languages have the primary goal of creating an easy language. To use this concept in our programming language we have to create a clear definition of what a word and what a special character is.
A word in the English language consists of alphabetical characters, meaning all characters from A to Z, upper and lower case. To include numbers like “3” or shorten words like “three” we widen the alphabet of our language to alphanumerical characters. In addition, we add the underscore character, acting as a space character to fuse multiple words to one, or to define a special element with this special character.

Syntax:

Even if we know all words, we have to know how to use them. A single Word does contain a tiny piece of information. To form a true meaning, we have to know the relation of these words. We can define rules, so-called syntax rules, which define the order certain words have to be in, to form a certain relation between them. Many spoken languages know additional concepts like declension and conjugation. These concepts have the purpose of matching deviation to certain rules instead of being one on their own and they improve the flow of speech[5], coming at the cost of a more complicated rule set to learn. Since we can define our own rules and can forbid any deviations we can exclude those concepts for an easier rule interpretation. Misunderstandings are another artifact of unclear rules. We must avoid that one sequence of words matches multiple syntax rules.

Code Structures:

The meaning which is defined by a syntax rule we call a structure. A structure can have two basic functions: It can store values or it can modify values. Structures that modify values are called operations and the ones storing values are called declarations. Operations have to access those declarations to modify the. This requires the operations to know the location of the declaration. The location in the memory is known through an address. This address would be one or more bytes in the computer. To make it easier to use the language, we can assign a name to this address. This also allows for the location to be determined later because we know where the same address is used. Every declaration associates a name with the values it stores. A declaration cannot only store data values, but also other structures. For example, if we have an operation sequence as seen in **Error! Reference source not found.**.
This operation sequence contains identical parts. These parts can be stored in a declaration as seen in Figure 2. If we want to use the operations in the declaration, we refer to the declaration. Such a declaration is called a function. Only a function can contain operations.

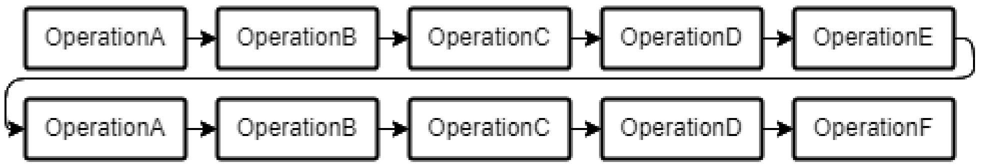

Figure 1[4]

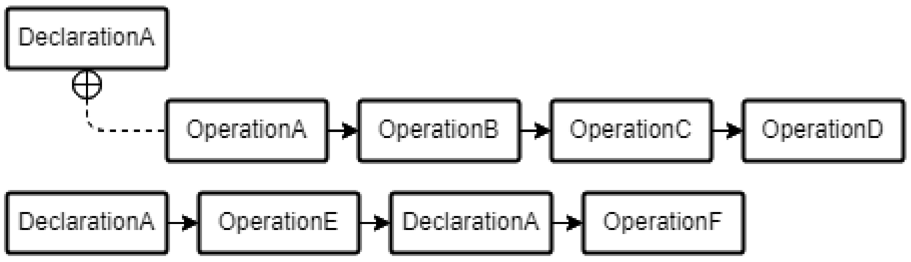

Figure 2[4]

This makes the combination of the declaration and operations identical in execution to Figure 1. This does not only shorten the code the more operations are stored in a single declaration, but it also allows for recursion. Recursion is the concept of a function referring to itself as shown in Figure 3.

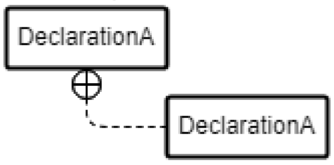

Figure 3[4]

This is one way of allowing a certain set of operations to be repeated indefinitely.

Another common declaration is the variable. A variable has the sole purpose of directly referring to the location of a stored data value.

Another kind of declaration is the class. The class is only used in programming languages, following the concept of Object Orientated Programming.

Object Orientated Programming:

Object Orientated Programming, or short OOP, introduces objects to a language. An object is a data value that contains multiple variables and functions. To know which variables and functions these are, every object has an assigned type, which is defined in a so-called class. A class is a declaration that only contains other declarations. These are the variables and functions associated with the object.

Inheritance:

Inheritance is a key feature of OOP. Imagine you create your class `Image`. It should store the RGB values of an image and be able to set the value of a pixel. To do this, we declare the variable `rgb` and the function `setPixel` in this class. In some instances, we want to load and save this image from and to a file. To do this we create a similar class `FileImage`. This class is defined identically to class `Image` but additionally with the variable `file` and the functions `save` and `load` illustrated in Figure 4.

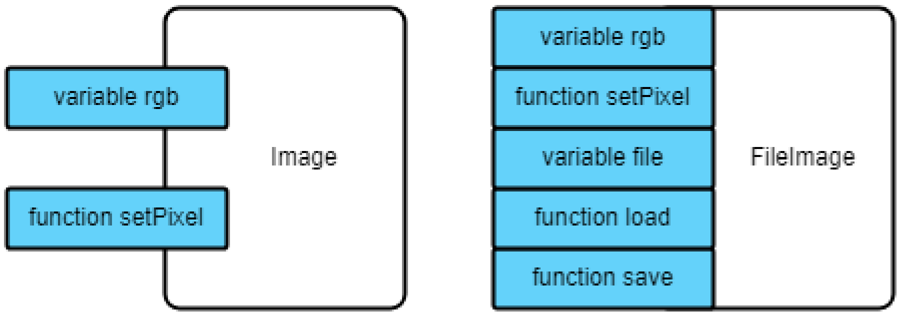

Figure 4[4]

This concept is called composition. All we can do with `Image`, we can also do with `FileImage`. However, during runtime, the program sees two different types with two different compositions. Even though the compiler could find scenarios where one type can be used as another, it is susceptible to change in the composition. To use one type as another the programmer must explicitly tell the compiler. This is where inheritance comes into place. A class can inherit all the contents of another class. The inheriting class is called the subclass or child class and the class inherited from is called the superclass or parent class. The relation is called: the subclass extends the superclass. All content of the superclass is no longer defined inside the subclass. This shortens the code and guarantees the compiler that the subclass type can be used as the superclass type at any time. We can display it as in Figure 5.

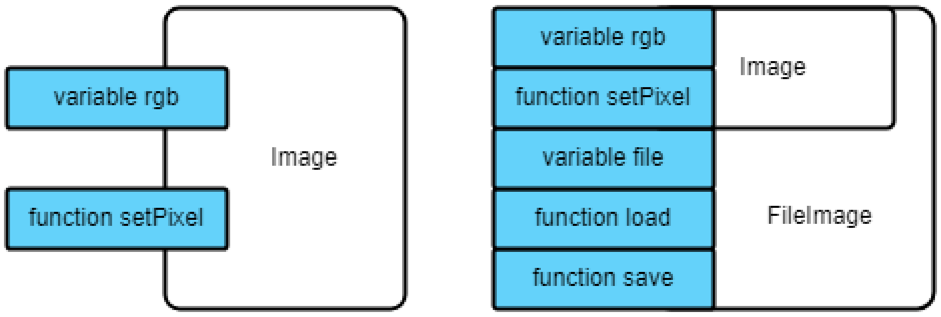

Figure 5[4]

Inheritance is one of the many advantages of OOP, however, in a specific scenario, a significant problem arises. Assuming we declare 4 classes: `Numbered`, `Image`, `File` and `FileImage`. An `Image` can be its own object and should get a `number` declared in the `Numbered` class. So does `File`. `FileImage` should now contain both `File`’s and `Image`’s features. So, `Image` extends `Numbered`, `File` extends `Numbered` and `FileImage` extends `File` and `Image`. Illustrated in Figure 6.

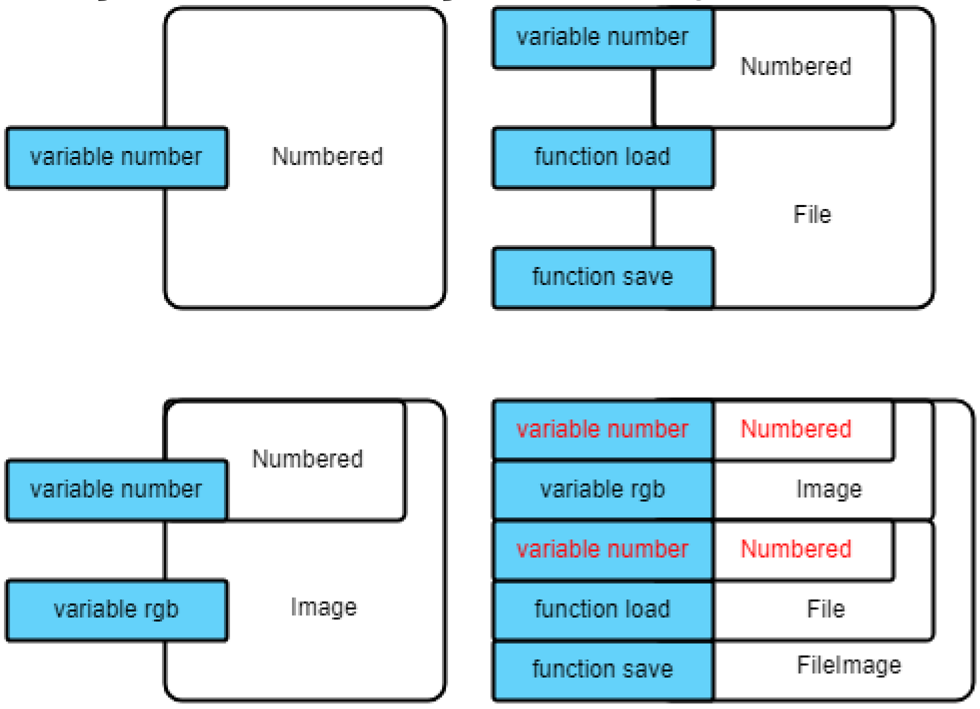

Figure 6[4]

We see that `Numbered` is now present twice in the `FileImage` class. Neither of them can be removed because it would remove a property of either `File` or `Image`. Some languages using OOP ignore this issue and leave it to the programmer to avoid these constructs or to always refer to the same one of the two number variables in this example.
Other programming languages face this issue by treating the inherited properties as unique properties. They essentially remove one of the `number` variables (Figure 7), which we have established cannot be done.

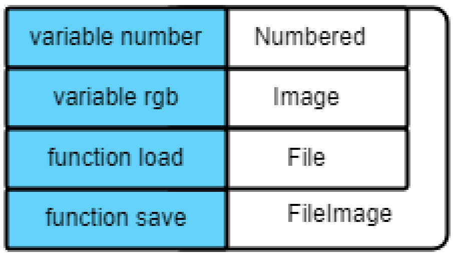

Figure 7[4]

However, in those languages, `FileImage` could not be used directly as either `Image` or `File`. They use address templates that reference the contents of the object. Whenever a class is used as another class, active conversion of the said template is required.
The `FileImage` template (Figure 8) is converted to the `File` template (Figure 9).

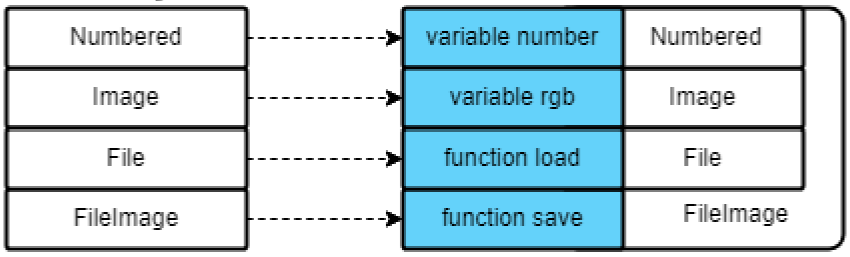

Figure 8[4]

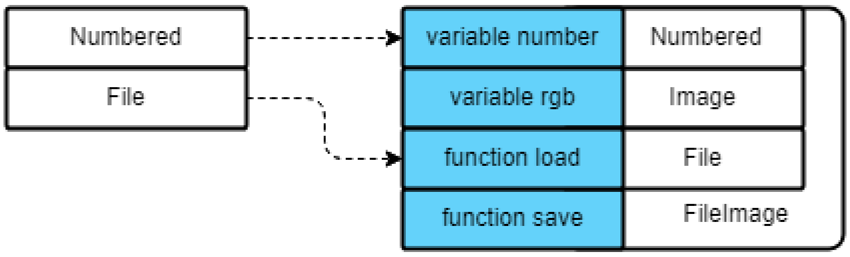

Figure 9[4]

Any conversion from one type to another, active or passive, is called casting.
Casting is possible in both ways, but it is only guaranteed in one way. Whenever a not guaranteed cast is performed the types are compared during runtime, since all the compiler knows, is the previous type. For this reason, any OOP programming language that considers potential mistakes by the programmer stores the type of an object during runtime.

Encapsulation:

A feature coming along with OOP is encapsulation. It adds restrictions to the accessibility of objects. It is called the visibility of a declaration. There are usually three visibilities: `public`, `protected` and `private`. `public` declarations can be accessed from anywhere. `private` declarations can only be accessed locally, inside the class or the file of the `private` declaration. `protected` declarations can also only be accessed locally but they have an extended range compared to `private`, for example to the directory or subclasses. Encapsulation allows the programmer to hide variables or functions that should only be used locally, to ensure that no unwanted values arise from them.[6]

The CPU:

The CPU is the most important microchip in a computer. It is the component of a computer capable of executing a program's instructions in form of bytes. The instruction executed by each byte is determined by the hardware of the CPU called architecture. There are many instruction architectures on the market, but the most used consumer CPU architecture nowadays is AMD-64. Ideally, compilers can produce byte code for multiple architectures, but they can also specialize in a single architecture.

Registers and Memory:

A CPU must be able to store bytes for later use, either instruction bytes or data bytes. It can do this in different locations: registers or memory. Registers can be differentiated between general-purpose registers and specialized registers (

Figure 10).

General purpose registers in the AMD-64 architecture can store 8 bytes each, specialized registers store up to 16 bytes. Memory refers to the set of bytes stored in an external computer chip, the so-called RAM (Random-Access-Memory). Registers are the preferred medium of storage because the CPU can access them fast. However, there is a limited number of registers accessible to a single process. To store more data the CPU can store them in memory. To do this, the CPU loads large chunks of memory and stores them in the so-called cache, an intermediate memory unit inside the CPU to accelerate memory access. The CPU can access this memory via an address. This address is a single number called a pointer. In the AMD-64 architecture, a pointer is 8 bytes or 64 bits long. The CPU provides simplified memory access by always storing a pointer and creating a virtual stack with it (

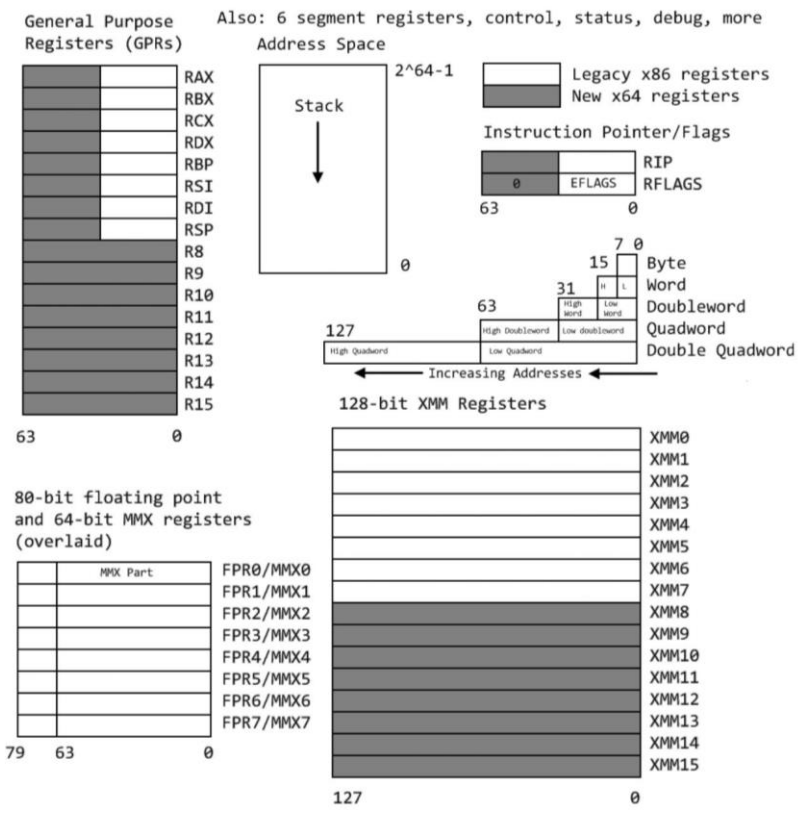

Figure 10). A stack is a data structure that allows you to store values in a list, following the Last-In-First-Out principle. The memory stack allows us to store values and retrieve them later without worrying about their address.

Figure 10[7]

Nylium:

Compiler:

The Nylium compiler is written in C++17 and compiled with CMake. It is available via GitHub at https://github.com/ttehbx222/Nylium. It is open source licensed under the Apache2.0 license. It is designed to produce AMD-64 byte code. The compiler is a Windows executable without any argument being provided. In theory, it should provide the possibility to select a different folder as the project folder by the path. Its language is similar to C++ itself but its runtime works fundamentally different.

Project Structure:

Whenever the compiler runs, it has a target folder. This folder is called the Project folder. All contents of this folder are searched for .nylium files which are considered to be code written in the Nylium language. The folders these files are in are considered packages and can be referred to by name.

Parsing:

The Compiler reads the file line by line. Every line is parsed using regular expressions. Regular expressions, or short regex, are character sets, defining a search pattern to find certain character sets.[8] The compiler currently differentiates between 6 types of words and 6 types of special characters.

Words:

Name:

A Name is the basic concept and most commonly used type of a Word. It is defined by the regex `[A-Za-z_][A-Za-z0-9_]*` which refers to any character set starting with an alphabetical character or an underscore and being followed by any amount of alphanumerical characters and underscores.

Hex:
A Hexadecimal integer is defined by the regex `0x[A-Fa-f0-9]+`. For example, 0xE7 where E7 carries the integer value.

Bin:
A binary integer is defined by the regex 0b[01]+. For example, 0b11100111 where 11100111 carries the integer value.

Integer:
A decimal integer is defined by the regex [0-9]+. For example 231.

Character:
A character is defined by `'[^'\\]'|'[\\].'` which refers to any character except `'` and `\` or any character prefixed with `\` and prefixed and suffixed with `'`. Notice that this includes characters that would otherwise be classified as special characters.

String:
A String is defined by `"([^\\]|[\\].)*”` which refers to any set of characters which are not `"` and `\` or are prefixed with `\` and prefixed and suffixed with `"`. A String is constructed from Characters.

Special characters:

Empty:
The empty sequence is defined by the regex `[ \t]+` and is not stored in any way. It consists of empty spaces and tabulator characters and serves as a separator between words and other special characters.

Bracket:
Brackets are defined by the regex `[(){}[\]]` meaning to a single character out of `(`, `)`, `{`, `}`, `[` and `]` however `[` and `]` currently have no use in the Nylium language. Every opening bracket requires a matching closing bracket later in the code. An opening and a closing bracket form an independent space.

Operator:
An Operator is defined by the regex `[+\-*\/&|^!=:.<>]+` meaning any character set build of `+`, `-`, `*`, `/`, `&`, `|`, `^`, `!`, `=`, `:`, `.`, `<` and `>` containing at least one character. It is used to create short names for functions and add mathematical notations to the language.

ListSeparator:
The ListSeparator is defined by the character `,`. Lists created by this character are called Open-Line-Lists since each element did not end with the proper End sequence.

End:
The End sequence is defined by the character `;` indicating the end of a virtual line. Whenever an End sequence occurs, all following characters are treated as their own line.
A line always has to end with the End sequence. If not, the next line of the file will be treated as an extension of the previous line. Separating list elements with this sequence creates a Closed-Line-List.

Comment:
Defined by the character set `//`. It forces the compiler to start reading a new file an with that ignoring all the remaining characters in the current line.

The compiler checks whether the line starts with any of these types. If there is a match, the characters set is stored respectively and is removed from the line, and the process repeats until there are no characters left to process. If there is no match the compiler notifies the programmer with an error and it continues with the loading as if the mismatch didn't exist, however, it does not continue past the syntax reading.

The Nylium compiler organizes these character sets during the character processing stage. Any opening bracket creates a list of lines, where every line is ended or separated by a `,` or a `;`. A matching closing bracket defines the end of this list. In the case of the scope bracket `{}` the line the brackets are in ends as if it was ended with `;`. This organization helps to treat brackets as a special character, meaning that a structure requiring a bracket does not have to read the contents of the bracket.

Structures:

In the Nylium compiler, all the structures are defined in C++ classes in an inheritance tree seen in Figure 11. Objects from these classes are called Bodies or CodeBodies.

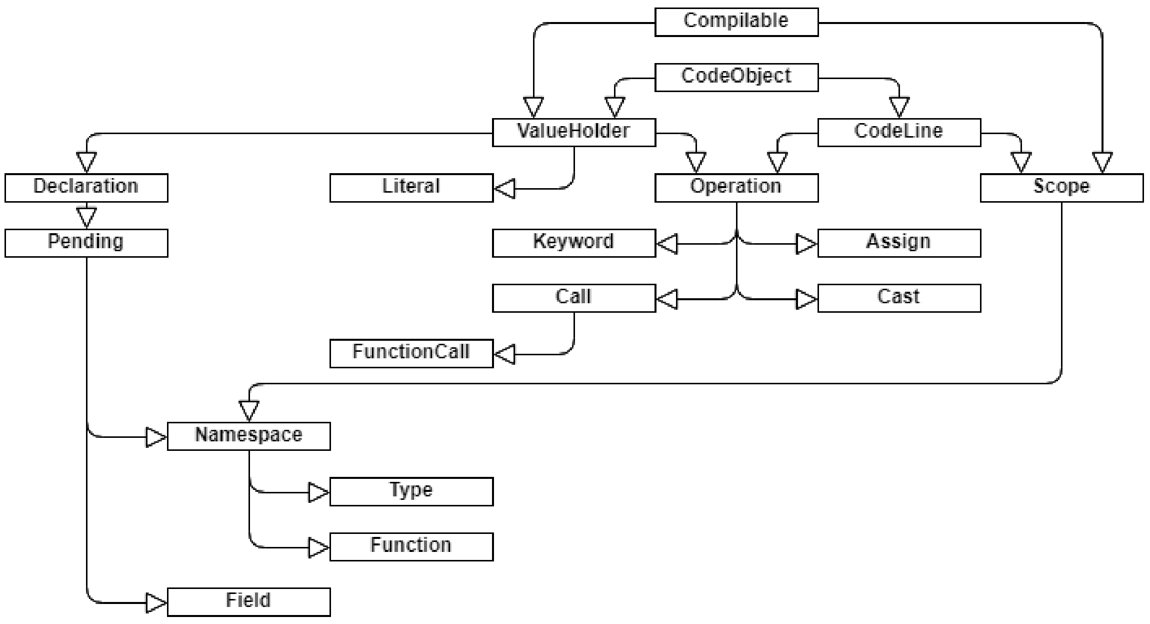

Figure 11[4]

Pending, Literal, Keyword, Assign, Cast, Call, FunctionCall, Scope, Namespace, Type, Function and Field are structure types while Compilable, Declaration, ValueHolder and Operations act as common properties of those structure types. CodeObject and CodeLine are obsolete after the introduction of Compilable.
A Pending, or PendingDeclaration, is composed of a single name. A PendingDeclaration is only used to name a reference to another Declaration without knowing whether it exists.

Use of Structures in Nylium:

A **Declaration** can always have attributes: one of the three visibilities: `public`, `protected` or `private`, `private` being the default. A Declaration can also optionally have `static` or `final` as attributes. `static` removes a Declarations association to an object. All Declarations outside a class are always `static`. `static` Declarations are always referred to with the `::` operator. For example

```
Foo::bar
```

`final` prevents a Field from being reassigned with few exceptions, prevents functions in classes to be overwritten by new functions and prevents classes from being extended. Constants should always be labeled `static final`.

A **Scope** is a Closed-Line-List defined with the `{}` brackets.

A **Namespace** does not have any effect on the execution of code, it simply forces a static call from outside the namespace on a declaration inside a namespace. It is declared with the `namespace` keyword, for example:

```
namespace Foo {
}
```

A **Type** is the structure equivalent to a class. It defines the template of an object and is declared using the `class` keyword. For example:

```
public class Bar {
}
```

If a Type has the `static` attribute it is converted into a Namespace. Nylium provides certain Types by default, they are called native Types. The `bool` Type is one of the most relevant, it can have two values: `true` or `false`. `void` is another relevant Type, representing the non-existent Type. It is used whenever a Function should not return a value.

A **Field** is declared with the Type of object it stores and with the name it is associated with, in this order. For example:

```
Bar foo;
```

Where `Bar` is the Type and `foo` is the name. Names of Fields must be unique inside their scope.

A **Function** is declared with the return Type, its name, and an Open-Line-List in `()` brackets in which the parameters are declared like Fields. For example:

```
Bar foo(Bar param) {
}
```

Where `Bar` is the return Type, `foo` is the name and `param` is a parameter of Type `Bar`. Names of Functions do not necessarily have to be unique inside their scope as long as they differ in their parameter Types. The return Type currently has no use as no return statement has been implemented yet. Every function must therefore be of return Type `void`.

Operations can only be placed inside functions. They carry the executable code of the program.

The **ValueHolder** class holds all properties for a structure, which does hold or return a value.

A **Keyword** Body is an operation which is initialized by a Keyword. Nylium currently knows 4 such Keyword Bodies: if, else, for and while. They are defined by the template

```
if ( <ValueHolder<bool>> ) {
    //…
}
else {
    //…
}

for ( <ValueHolder> ; <ValueHolder<bool>> ; <ValueHolder> ) {
    //…
}

while ( <ValueHolder<bool>> ) {
    //…
}
```

`if`, `for` and `while` require a ValueHolder holding a value of Type `bool`. This ValueHolder can be a reference to a Field or any Operation. Only if this value is `true` the scope will be executed. `else` requires an `if` in the previous line and is only executed if the value given to the `if` statement is `false`. `for` and `while` provide loops, which means that the Scope of the statement is repeated until the value provided is `false`. `for` has to be provided with an Operation being executed before the loop starts and an Operation being executed after every iteration of the loop. This allows for Operations determining the duration of the `for` loop to be written in a single line.

A **Call** is the structure where an Operation accesses a Field which is not locally accessible. Calling a Field from an object is done with the `.` operator. Calls to static Fields are done with the `::` operator and do not affect the execution. Since the location of a static Field is known during the compilation the compiler will directly refer to the location of the Field.

A **Function** is called with the FunctionCall structure. It is identical to a call suffixed by the arguments in an Open-Line-List defined by `()` brackets. The order of argument must match the order of parameters which should store the argument values. For example:

```
bar.foo(value);
```

Where `bar` refers to the target object with the function `foo`, and `value` is the provided argument.
A special kind of FunctionCall is the use of operators. Such a FunctionCall can look, for example, like this:

```
bar + value;
```

Here `bar` remains the target object. `+` is the function and `value` is the argument. These FunctionCalls can only hold a single argument. They allow for shorter and mathematical function notations. Currently, only native Types can have such Functions since no way of declaring them has been defined yet.

A **Cast** is performed whenever the Type of an object is changed. If we have the classes `Foo` and `Bar` and `Foo` extends `Bar`, we can use `Foo` whenever we need `Bar`. Still, a conversion takes place. This type of Cast is called an implicit Cast. It is not written by the programmer but detected by the compiler. If we want to cast `Bar` to `Foo` however, this cast is not guaranteed. We need an explicit Cast which is defined by the new Type in `()` brackets, prefixing the ValueHolder. In this example it would be:

```
(Foo)value;
```

Where `value` is of Type `Bar` and is cast to `Foo`.

An **Assign** looks like a FunctionCall with the = operator, however, it is handled differently by the compiler. It can be performed on any Field and reassigns the value the Field holds.

A **Literal** character set is always converted into a Literal. It can be used at any place where a ValueHolder can be used.

Linking:

Linking is an important process for the compilation. Before linking, every reference to a Declaration is only a PendingDeclaration. During linking all these references are resolved, meaning the compiler searches for the Declaration with the name of the PendingDeclaration and saves it in the Body referring to it. During linking the compiler checks whether a certain Type matches the required Type. Because of this linking requires a certain order. First, all Types are linked to the Types they extend. Then the superclasses are resolved recursively so that every type knows all the types it extends directly or indirectly. Functions and Fields are linked before the Operations. Linking the Operations last is the best option since they do not contain any declarations themselves so when linking takes place, all declarations have already been finalized.

Linking happens twice during compilation:
Once after the Code has been analyzed and once after the byte code has been created. The linking on the byte code does no longer resolve Declarations but rather references to instructions and data.

Translating:

Everything after this point in this documentation is purely theoretical as of January 9th, 2023.

After the first linking process translation takes place. During translation, all Bodies are translated into so-called assembly blocks. Assembly blocks are a set of assembly instructions and data. They are not stored as byte code yet but are C++ objects in the compiler. Every Body class has its proper compile function in which it constructs its corresponding assembly block.

Field:
A Field is compiled to a memory allocation of 128 bits in a 64-bit system. This memory will be freed as soon as the scope closes. In Objects, the memory allocation happens during the

construction of the object together with all the other fields in the class. The size of this memory block is calculated during compilation in order to save time and resources during runtime. A static Field is located directly in the byte code and its memory is neither allocated nor freed during runtime.

Function:
When a Function is compiled its assembly block consists of all the Operations it contains. Furthermore, it notifies a Field Body when the scope closes to create the instruction to free its memory. All parameters are compiled as Fields with their original location being the memory stack.

Type:
The Type is the most complex structure to compile since its functionality should be flexible but also efficient. This comes at the cost of huge quantities of memory being used.
A Type needs to know its memory size which is composed of the number of Fields and Functions. Usually, this quantity is rounded up to match a value the memory allocation algorithm can process faster. A Type needs to know whether it is compatible with another Type, so the assembly block contains a byte map. The byte map contains as many bytes as there are Types in the code. If the code is dynamically loaded into another process later, this segment should be resizable during runtime. For now, we exclude this feature and create the byte-map statically in the byte code. Every Type in the code has its unique location which is identical in every byte map. Every byte contains a pointer to a template. If the byte is zero then the template does not exist. Whenever the program checks whether a Type is compatible with another Type, it checks whether there is a template in the byte map of the old Type at the position assigned to the new Type. When receiving a non-zero template this template is used. The template is composed to match the new Type and instead of containing the actual values it contains offsets that tell where in the object a value of the new Type is stored. A Field stores both the pointer to the object and a pointer to the Template, hence 128 bits on a 64-bit system. To access a member of the Type the process adds the object pointer and the offset in the template to get the location of the desired member. The complexity and intensity is the main problem of safe inheritance, where a Type can inherit another Type multiple times but doesn’t contain it multiple times.

Assign:
During an Assign, the object pointer and the template pointer are set to the values of another ValueHolder. For this, the compiler always needs to know the memory location or register location of every ValueHolder.

Cast:
A Cast retrieves the template from an old Type to match a new Type. If there is no matching template the Cast will throw an error. Since Nylium does not have the concept of throwing and catching yet, the program will crash.

Call:
A Call retrieves the object pointer and the template pointer from another ValueHolder.

FunctionCall:
A FunctionCall makes the CPU jump to the location of the Function. It has to remember the previous location to continue where the function was called. All arguments are pushed into unsafe registers or the memory-stack

Assembly:
Every instruction in an assembly block modifies a value. The compiler must always know the location of a value. Therefore, during compilation, everything is compiled in the order it will be executed. Whenever a value's location is changed, the new location is remembered by the compiler for the instructions to follow. Whenever we don't know the order, calling a function for instance, we define where these values are actively put, if they are not already in place.

Bytes:
Every assembly block now contains a set of instructions and data. The data can be converted directly to bytes. As in the assembly language, they can be translated one by one without any context. For instructions, this requires a translating table. As of January 9th, 2023, Nylium does not have a translating table.

Current State:
As of January 9th, 2023, the Nylium compiler is incomplete. It does not yet produce any byte code. However, it produces text output to show all structures detected by the compiler. Linking is only partially implemented and a return statement structure is not yet defined. If I were to continue the compiler those two features would be implemented next. I would have to generate a translation table for Assembly instructions and learn about the structure of an EXE file, so that the compiler can build the final byte code as an executable file.

Conclusion:
During this project I have experienced the struggle that comes with the creation of a compiler. I had to look into various fields of computer programming. When I started creating the compiler, I knew how to write code but today I know in many ways how it works. Of course, I am missing many features modern languages have but the basics help a lot already. With today's knowledge I would rewrite large parts of the compiler and rethink many concepts used in it. I didn't design Nylium to be a special language. Rather than implementing some special features, I designed the language to be similar to the ones I know. This helped immensely with understanding how a compiler works and will help me with further programming projects.

References

Nylium GitHub: L. Nieswand: Nylium (2022), URL: https://github.com/ttehbx222/Nylium (05.01.23)

[1] : R. Sheldon: Compiler (Kompilierer), URL: https://www.computerweekly.com/de/definition/Compiler-Kompiler (05.01.23)

[2] : Runtime (2006), URL: https://techterms.com/definition/runtime (05.01.23)

[3] : D. Morgan-Mar.: Piet (2018), URL https://www.dangermouse.net/esoteric/piet.html (05.01.23)

[4] Diagrams created with: Visual Paradigm Online: https://online.visual-paradigm.com

[5] : Declension (2022), URL: https://en.wikipedia.org/wiki/Declension (05.01.23)

[6] : D. Braunschweig: Encapsulation, URL: https://press.rebus.community/programmingfundamentals/chapter/encapsulation/ (05.01.23)

[7] : Introduction to x64 Assembly, URL: https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf (05.01.23)

[8] : regular expressions Definition, URL: https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_reg_express.htm (05.01.23)