Modern Programming for Old Hardware

- Design and Implementation of a Programming Language and Compiler -

4th Semester Project Report

Group 2

Aalborg University

Sofware

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Modern Programming for Old Hardware

Theme:
Modern programming for old hardware

Project Period:
Spring Semester 2023

Project Group:
2

Participant(s):
Amalie Pernille Dilling
Anders Mazen Youssef
Bence Szabo
Freja Lüders Rasmussen
Louise Foldøy Steffens
Magnus Peetz Holt

Supervisor(s):

Copies: 1

Page Numbers: 150

Date of Completion:
August 26, 2024

Abstract:

The development of an imperative language for the MOS 6502 processor from the 1980s is described in this report. The goal was to revitalize this outdated processor by incorporating it with contemporary language design principles. The syntax and semantics of the language were meticulously designed, and a compiler was developed to validate and convert the custom language code into Assembly 6502 code. Users preferred using the custom language over learning the original Assembly language, according to usability tests that assessed the custom language's efficiency and revealed an improvement in accessibility to Assembly 6502 code. In conclusion, the developed language successfully facilitates the utilization of Assembly 6502, ensuring the continued relevance and utility of this dated processor in the modern computing landscape.

Preface

Aalborg University, August 26, 2024

Amalie Pernille Dilling
adilli21@student.aau.dk

Anders Mazen Youssef
amyo21@student.aau.dk

Bence Szabo
bszabo21@student.aau.dk

Freja Lüders Rasmussen
flra21@student.aau.dk

Louise Foldøy Steffens
lfst21@student.aau.dk

Magnus Peetz Holt
mph21@student.aau.dk

Contents

Chapter 1

Introduction and Motivation

In this project, the design, definition, and implementation of a programming language are documented. All software written in any programming language (other than machine language) must be compiled or interpreted before it can be executed, hence, a large part of the paper consists of a walk-through of all necessary steps taken during the construction of a compiler for a language specifically designed for this project.

The problem tackled in this project is the construction of a modern language designed for an old 8-bit processor, the MOS 6502 processor which was first released in 1975 by MOS Technology. The MOS 6502 had a big impact on the 8-bit systems of that era, as it was used in the Nintendo Entertainment System, the Apple II, the Commodore 64, and many other significant machines. At that time, the chip was faster than competitors from Intel (Intel's 8080) and Motorola (Motorola's 6800), both of which sold for nearly 200 USD. The MOS 6502 sold for 25 USD, nearly one-tenth of the price [47].

Intel's 8080 had 246 instructions, and the Motorola 6800 had 72 instructions, both of these instruction sets are complex when compared to the mere 56 instructions in the 6502. According to Bill Mensch, who co-created the 6502, the key factor that enabled the cost reduction was the combination of a minimal, yet effective instruction set and a fabrication process that produced significantly more functional chips than competitors. As a result, the 6502 played a major role in driving down the price of processors and played a critical role in the emergence of the personal computer revolution [47].

The MOS 6502's 56 instructions dwarf in comparison to x86-64's 3,684 instruction variants [55]. Because of this, to get a grasp of coding in a low-level language, many hobbyists choose to learn Assembly 6502 rather than other more complex machine languages, although translating modern ideas to a language that is a lot more limited proposes a whole new set of challenges.

As of the writing of this paper, a revision of the processor is still being manufactured, now by Western Design Center [11]. It is still being licensed to companies around the world, for use in things such as automobiles, and medical equipment like pacemakers and defibrillators. Beyond this, they are also used for various security systems and railroad control systems. 17.

Currently, developers must resort to writing their applications in the low-level language Assembly 6502. To preserve an important piece of technology, and in turn, make it more accessible for developers to use, it could prove beneficial to bring a programming language of modern language design characteristics to an otherwise dated processor.

Starting in this problem field, some requirements are already presented. As mentioned above, Assembly 6502 is the target language, to which the new language must be translated. Some of the main issues, which the new language should improve on are the following [33]:

- Time-consuming: Writing code in Assembly 6502 takes a lot of time and effort compared to higher-level languages.
- Difficult to read: Assembler code is generally difficult to read due to it being incredibly close to the hardware.
- Difficult to debug and maintain: Connected to the last item on the list, the lack of readability also means that debugging and maintenance can be rather challenging.
- Steep learning curve: Assembly language is generally more difficult to learn and requires a deeper understanding of computer architecture and low-level programming concepts. This can make it challenging for beginners or programmers without a strong technical background in computer architecture.

The new language constructed throughout this project should focus on these issues while encompassing some overall characteristics of a good language. But being aware of the limitations of the project is also central; during the limited 3-month time frame of the project, not all elements of a modern-day programming language are realistic to implement. Ideally, the fundamental types (int, boolean, etc.) and control structures (selective, iterative) should be implemented, and if possible, some level of modularity (procedures, functions) would also be a welcome addition, to provide further abstraction.

1.1 Two iterations

In this paper, two iterations of the development are documented, the first one being a simple compiler that has the essential features to function, and the second iteration focusing on refining the language and inner workings of the compiler, providing a more optimal solution.

The planning, designing and construction of the compiler, primarily during the first iteration of the product, is done by following Fischer et.al.'s book, Crafting a Compiler [16].

It was decided that it would be best to have two iterations of the product for this report. Developing a compiler was very much foreign to the developers, which is why the iterative design philosophy was reflected upon when the timeline for the project was created (See GanttBased on the context and schedule, diagram 17).

The Gantt diagram closely follows the university courses, as the lectures teach about the various stages of creating a compiler in chronological order. As the courses progress, the theoretical knowledge about techniques, decision-making, and good practices in language definition and compiler crafting are constantly translated to and applied in the context of the project.

On the other hand, if it was decided to fully complete every single stage on the first iteration, there would be a risk of falling behind on the course material. Thus, it was decided to have two iterations, as it allows the implementation of the relevant knowledge from a recent lecture before the next lecture. This means that the first iteration of the product would be finished by the end of the course, which then gives time to reflect on it, evaluate it, and start on the second iteration afterwards by going through the stages again. Generally, an iteration follows a software development life cycle, which in this project consists of, research, analysis, design, implementation, and evaluation. The second iteration is expected to focus more on the design and implementation phases, using the evaluation of the first iteration as a foundation.

1.2 AI based tools

Throughout the project, ChatGPT 3.5 [1] and GitHub co-pilot [22] are used to assist the team's workflow. ChatGPT is used to answer relatively simple technical questions. It is not treated as a reliable source, but rather a helping hand. GitHub co-pilot is used as a code completion tool, providing contextually aware code suggestions.

Part I

Theory and First Iteration

Chapter 2

Computer Architecture

Since the development of this project's compiler starts with a specific target language (and hardware), setting the initial goals and expectations for the source language is done with regard to the target language at hand, and the limitations it presents. Therefore, before the source language is designed, the specifications of the hardware at hand are explored in this section. Besides providing valuable language design information, understanding the hardware is fundamental for code generation. This is because Assembly 6502 is "close to the hardware", which is designed for a specific type of system architecture. Without sufficient knowledge of the processor's architecture, writing functioning code for it would be needlessly difficult.

2.1 MOS Technology 6502 microprocessor

This section describes the inner workings of the MOS technology 6502 microprocessor. The MOS 6502 is an 8-bit microprocessor, meaning it can transfer data around with a length of 8 bits. The way it transfers data is with the data bus as seen in figure 2.1. The 6502 has three buses in total; the data bus, which can transfer 8 bits at a time; the address bus, which can transfer 16 bits at a time, and lastly the control bus which tries to keep everything synced up inside the processor. The control bus is not important for the project and is therefore not described further. A component that uses the busses is the accumulator, which is responsible for doing arithmetic operations. The accumulator begins by loading a value into its register with an instruction (LDA) and then calls a new instruction which adds a value to the accumulator. The new instruction is responsible for what the accumulator does with the two values, which can be anything from addition to checking if they are equal or not. The new value can then be transferred with the data bus to a new register or the memory [9] [50].

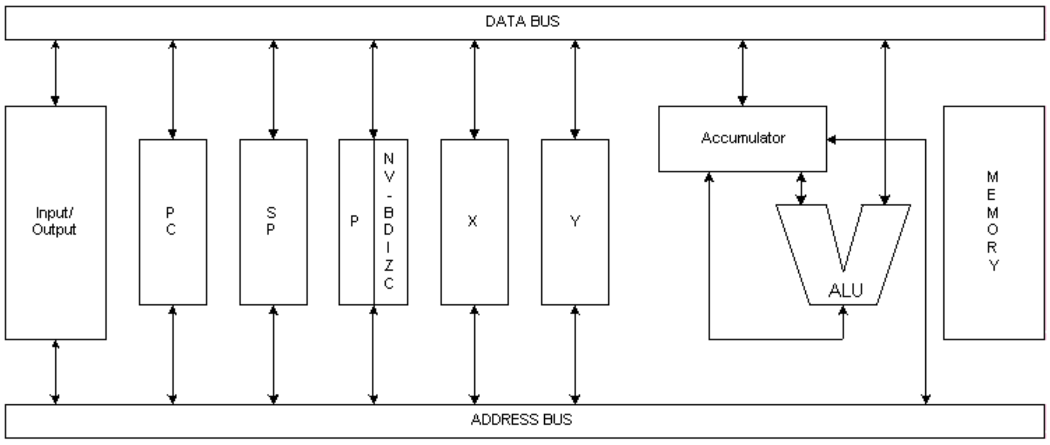


Figure 2.1: The inner structure of a 6502 microprocessor source[4]

As with any other processor, the 6502 microprocessor has some registers which include the two 8-bit registers X and Y as seen in figure 2.1. These two registers are used to store temporary values. The 6502's instruction set, which will be discussed in a later chapter, contains specific instructions to manipulate these registers, such as increment and decrement 7.1. This is useful when moving and modifying data because the instructions do not need a memory location, and therefore require fewer calls. Alongside the two registers is another 8-bit register P (Status register) also seen in figure 2.1. The P register holds eight 1-bit status flags, which can be either 0 or 1, except the last bit which is always 1, indicating specific scenarios during arithmetic operations. The status flags are listed below [50]:

- N (Negative): This flag is set to "1" if the result of an arithmetic operation is less than 0.
- V (Overflow): This flag is set to "1" if the result of an arithmetic operation is incorrect due to signed arithmetic.
- B (Break): This flag is set to "1" if an interrupt is caused by a BRK instruction.
- I (Interrupt): When set to "1", interrupts are disabled, and the processor will not respond to any external interruptions until the flag is cleared.
- Z (Zero): This flag is set to "1" if the result of an arithmetic operation is 0.
- C (Carry): This flag is set to "1" if there are any excess bits from the accumulator's operations.

Another register in the microprocessor is the Program-Counter (PC), which keeps track of the next instruction. It does so with a 16-bit register and holds the memory location of the next instruction. The PC can be manipulated by different instructions like jump (JMP), which adds some value to the PC so the program can jump back and forth in its code. Lastly, there is the Stack-Pointer (SP) register, which is a pointer to the element at the top of the stack in the memory [9]. The stack is discussed following the next paragraph.

Another vital component of the 6502 microprocessor is the memory, which has 16-bit addresses like the address bus. This means that the memory can keep track of 65535 different locations/elements, which is the same as 64KB. The memory is further divided into pages of 256 bits each for 64535 of the addresses in the memory. When going through the program, the programmer needs to keep track of which page and which line they are on [9].

Some pages are reserved by default. One such is the zero-page ranging from $0000-$00ff bits, which is a part of the memory meant for elements that are commonly used. This is because the zero-page's addresses are 8-bit and are therefore parsed quicker, and calls to the zero-page are more efficient than any other page. The bits ranging from $0100-$01ff are allocated for the stack, which is a last-in-first-out (LIFO) list. The stack has its own instructions to push and pull the accumulator to and from itself. The rest of the memory is either Random Access Memory (RAM) or Read-Only Memory (ROM). The ROM, as suggested by its name, cannot be edited and any attempts to do so are ignored. The ROM can, for instance, be a disc containing the data for a game, which of course should not be manipulated [9].

2.2 Simulated environment

To ensure the most authentic result when running code intended for a system using the 6502 processor, having a system which uses that processor would be ideal. However, an emulator is also an adequate solution. Firstly, because it is, in theory, a virtually identical representation of the target system, which means that the behaviour should match up to the real hardware. Furthermore, the code can be run without having to physically transfer it to the machine saving a lot of development time. Also, multiple developers can work using separate machines without having to spend resources on having access to multiple machines. For this project, a web-based 6502 simulator is chosen since it has all the necessary features to test the custom-designed language.

The emulator used in this project is hosted on a website called Easy 6502 [31], which is maintained by Nick Morgan who is at the time of writing this paper a front-end engineer at Twitter. The emulator itself is built entirely in JavaScript by Stian Soereng [46] and runs in the browser. The emulator can assemble, debug and run code written in Assembly 6502 language. It also has a console that displays relevant information regarding the accumulator, X and Y registers, SP and PC. These are especially beneficial during debugging. It also uses memory locations $0200 to $05ff to draw pixels on a display, which gives a total pixel count of 1020. The purpose of the emulator is to be used when completing the tutorials on the site, therefore the emulator lacks some of the advanced debugging features of modern integrated development environments (IDE), as it is meant to be used by novice developers of the language [35].

Chapter 3

Language Design

The initial design constraints of the language are based on the limitations imposed by the target language of the compiler, Assembly 6502, and the hardware specifications of the MOS Technology 6502 processor.

Firstly, the language cannot be *too high-level*, for example, a part of either the object-oriented or functional language paradigms, due to storage constraints and the relatively limited set of instructions the MOS 6502 works with 17. The language will therefore be imperative, providing more abstraction than Assembly, but still low-level to make code generation manageable.

Also, due to storage constraints, as well as the complexity of code generation that comes with composite data types, all types during the first iteration are atomic. Arrays or other composite data types are not considered to be included.

The relevant atomic data types are integer, float, boolean, and pointer. Integers and floating-point numbers are included to have any kind of numeral values and arithmetic operations, which are nearly unavoidable in any kind of programming. Boolean is included to enable logical operations, and pointers are included since they can be used to manipulate the memory directly.

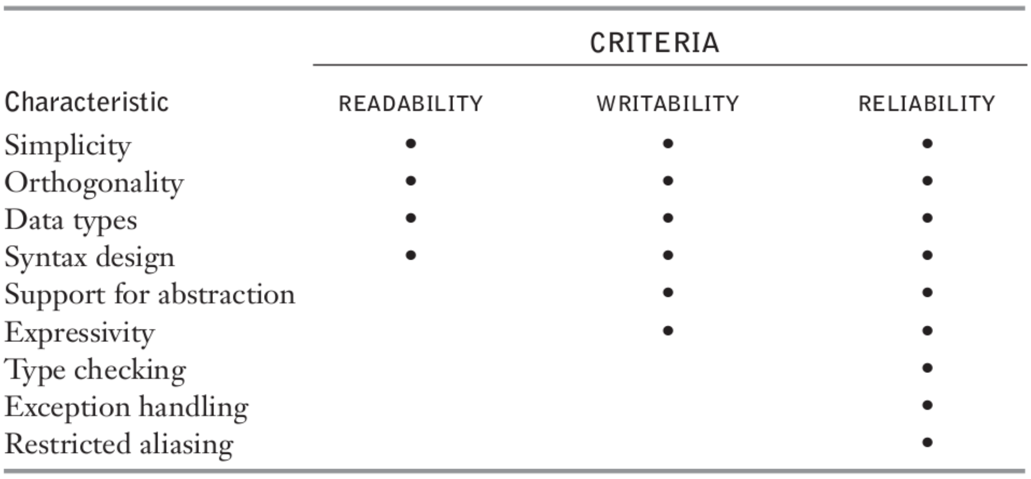
Due to the limited size of the stack in the 6502 processor (only 256 memory addresses), it can easily become a limitation when developing code. If the stack is not managed carefully, it can overflow and overwrite other portions of memory, leading to unexpected behaviour and bugs in the software. Additionally, the limited stack size may require careful planning and optimisation of code to ensure that there is enough space for storing necessary data. This limitation can impact the design and complexity of software written for the 6502 processor and requires careful consideration during development.

Lastly, a crucial requirement for the language is that it cannot be ambiguous. This is because the parser of this project will either apply a deterministic top-down left-to-right (LL), or a deterministic bottom-up left-to-right (LR) parsing technique. The project is restricted to using deterministic parsers because the curriculum of this semester only focuses on deterministic parsers.

3.1 Language Evaluation Criteria

When discussing the design and implementation of compilers for modern programming languages, good language design should be the primary interest. According to Charles N. Fischer, a good programming language has the following traits [16]; The language is easy to learn, read and understand, and it has quality compilers with support for various platforms. The other characteristics that Fischer finds important seem to focus more on the compiler itself, so they are not included in this section.

Beyond this, there are also language evaluation methods to determine the characteristics of the language and how it impacts the software development process.



	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Figure 3.1: Language evaluation criteria and the characteristics that affect them [44]

By reading the characteristics of the table, some of them might seem vague, such as simplicity. In comparison, exception handling is a much more specific language construct. The characteristics will be discussed in the following sections.

Overall simplicity

It is easier to learn, read and write a language with a small number of basic constructs than one with a larger amount. Simplicity can be secured by avoiding feature multiplicity, which is when a language provides multiple ways to solve the same goal [44]. On the other hand, feature multiplicity can help programmers adapt to a new language, applying practices from other languages they have experience with. An example of this is how increments are done in Java, which can be done in four different ways:

```
    count = count + 1
    count += 1
    count++
    ++count
```

These different expressions can have slightly different results. But as a standalone expression, they are the same.

Operator overloading is another hindrance to the readability of a language. This means that an operator symbol has multiple meanings, which is fine when used in commonly accepted contexts, as in mathematics when "+" can be used in an addition operation both for integers and floating-points. Simplicity in programming languages can also go too far, and in turn, make the program less readable. If a language is too simplified, the scope and structure of the program may be less obvious.

Orthogonality

Orthogonality in a programming language means that aspects of the program can be changed without any unseen effect on the rest of the program [44]. This characteristic is defined by the independence of components in a larger system. An example could be class inheritance and interfaces from object-oriented programming languages, in this example, all these components are independent of each other, and changes to one feature would not affect the behaviour of another. Orthogonality is closely tied to the characteristic in the previous section because if a language is orthogonal, fewer exceptions to the rules of the language are required. This also means the language has a higher level of regularity, which makes it easier to learn, read and write.

Data Types

to increase the readability of a programming language, providing relevant data types and structures is important [44]. An example of this could be the absence of Boolean data types in a programming language. In this context, all numeric values other than 0 are true, and 0 is false. This can render the meaning of certain statements unclear to the reader.

Support for abstraction

Abstraction hides unnecessary details and assists in providing a clear interface [44]. To tackle this as a designer of a programming language, one should focus on expressing what the language does, not how it does it. By only providing the user with the required information, it can improve the writability significantly. Although an imperative language cannot reach the same level of modularity and abstraction as a language of a higher level, like Java, procedures and functions can help with abstraction, encapsulating a sequence of statements into a reusable block.

Syntax Design

This characteristic is about the structure of statements in a programming language [44]. *Reserved keywords* are a part of this design aspect, they are often found as the words **while, for, if**. These are words in the language that can not be redefined by the users of the language or be used as an identifier. When designing a good programming language, one needs to think about which words should be reserved, how statements and expressions are written, and whether the reserved keywords could be used as identifiers. Generally, a programming language's lexemes should represent what they are intended to be used for. In many cases, it is advisable to follow tradition in naming lexemes, so code written in the languages is easily understood by people coming from different programming backgrounds.

Expressivity

In this paper, expressivity means that when specifying computations, the methods of doing it are convenient rather than cumbersome [44]. An example of this would be the one mentioned earlier in the simplicity section, where increments can be done faster with a ++ notation after the variable, instead of adding the value of one to the variable with a longer statement. This characteristic is most important for writability.

Type Checking

Type checking is about testing for type errors in a program, which can be done either at compile time or run time [44]. Doing this at compile time means the program is only type checked once. Doing it at run time requires more memory, which the 6502 processor does not have a lot of. Because of this, it is already decided to do this at compile time, to take the performance costs off the target system. This also dictates that the language is statically typed, which will be discussed later.

Exception Handling

Exception handling is used to handle, and intercept run time errors to continue the normal execution of a program (try-catch) [44]. This is possible in some languages like Java, which has extensive tools for exception handling. However, in other languages, this is completely absent, which has a very huge impact on reliability. Due to the limited instruction set of the 6502 processor and its memory, exception handling is not implemented in the programming language of this paper. Nevertheless, the compiler itself is using exception handling in Java.

Restricted Aliasing

Aliasing refers to having multiple methods of writing to the same memory address [44]. The general verdict of this characteristic is that it is a dangerous feature. It is still possible in many languages, for example, to have two pointers or references that point to the same variable. The reason for it being a dangerous feature is that it is the source of many errors and can be difficult to detect and prevent.

Readability, writability and reliability cannot be measured or defined with complete accuracy. But it is valuable to be mindful of these concepts when designing and evaluating a programming language. An evaluation of these criteria will be discussed in section 16.

3.2 Regular expressions

Regular expressions (regex) is a formal notation that can describe the many kinds of tokens required by programming languages. Regex are, in essence, a sequence of various characters which are used to describe a specific search pattern [16]. Regular sets are a combination of strings defined by regex. An example of a regular expression is given below:

$$ID : (['' a'' - '' z'', \ '' A'' - '' Z'', '' '']) \ (['' a'' - '' z'', \ '' A'' - '' Z'', \ '' '', \ '' 0'' - '' 9''])*$$

In the example above, the regular expression for searching for an ID is given. It shows that it must start with either a lowercase or an uppercase letter. After the first letter, a sequence of letters and numbers can be added to the ID, zero or more times.

In the context of the programming language of this report, regex are used to specify the structure of the tokens, which the following examples will explain.

The SKIP token includes all the characters that should be skipped by the lexer 5. This is done by using the backslash notation to define special characters, for example, typing *n* matches the character but by typing \n then a newline character is specified. Comments must start with a forward slash notation followed by an asterisk, and they end with the same symbols in reverse order. Within these start and end notations of the comments, there can be two different contents. Either there can be everything except a forward slash, or one single forward slash followed by any number of characters that are not a forward slash zero or more times. The tokens are specified in listing 3.4.

3.3 Context-Free Grammars

Once the requirements for a language are established, the next step is to produce a formal definition; a context-free grammar. This section describes what Context-Free Grammars (CFGs) are and their role in building a compiler.

CFGs are used to define a programming language's syntax. It specifies the alphabet of the language and which sentence structures are allowed in the language for it to be correct. The sentence structures are defined by production rules. An example of a derivation which uses the production rules in 3.5 is given in listing 3.1 below:

Listing 3.1: Derivation for a = b + c

```
Prog → Stmt* eof eol
→ Assignment eol
→ id '=' expr
→ 'a' '=' expr
→ 'a' '=' OrTerm
→ 'a' '=' AndTerm
→ 'a' '=' EqualityTerm
→ 'a' '=' RelationalTerm
→ 'a' '=' AdditiveTerm
→ 'a' '=' NotTerm
→ 'a' '=' Factor
→ 'a' '=' Val
→ 'a' '=' id
→ 'a' '=' 'b' '+' AdditiveTerm
→ 'a' '=' 'b' '+' NotTerm
→ 'a' '=' 'b' '+' Factor
→ 'a' '=' 'b' '+' Val
→ 'a' '=' 'b' '+' id
→ 'a' '=' 'b' '+' 'c'
```

Derivations, Recursions, and Ambiguity

After the lexical analysis, the language's CFG production rules are applied to check the syntactic correctness of the program. During this process, all non-terminals are expanded into terminals, step by step. One such step is called derivation. Using derivation, a parse tree of the input program is created.

There are two different directions to derivations: leftmost derivations and rightmost derivations. Leftmost derivations expand the non-terminals from left to right, thereby generating a sentence of the language. Top-down parsers utilise this and produce a leftmost parse. Rightmost derivations expand the non-terminals from right to left. Bottom-up parsers use this, as they apply the last production of the rightmost derivations first and in this way work towards the start symbol. By default, the parser-generator applied in this project, JavaCC, produces a parser which uses leftmost derivation, since it is a top-down recursive descent parser. The process of parsing and the specifics of the parser-generator of choice is discussed in section 5.

One's formally defined grammar can either be left- or right-recursive, or none of the two. A production rule is left-recursive if it has the form $A \to A\alpha$ where A is a non-terminal symbol and α is a sequence of terminals and/or non-terminals. A production rule is right-recursive if it has the form $A \to \alpha A$ where A is a non-terminal symbol and α is a sequence of terminals and/or non-terminals. Left-recursive and right-recursive grammars can cause problems when used in parsing because they can lead to infinite recursion in some parsing algorithms. To avoid such problems, it is often preferable to use grammars that do not have left-recursive nor right-recursive rules or to transform the grammar to eliminate the recursion. One common technique for transforming left-recursive grammars is to use left-factoring, which involves splitting the left-recursive rule into two or more production rules. Similarly, right-recursive grammars can be transformed using right factoring. These techniques help to make the grammar more amenable to parsing algorithms and improve the efficiency of the parsing process [52]. The grammar of this project is, therefore, transformed to a state in which it is neither left- nor right-recursive.

One of the early problems in building the grammar is the structuring of the expression. Listing 3.2 is the first attempt on creating expressions with only addition and subtraction. The attempt led to an unforced implementation of a left recursion in the grammar. The thought process is that expression, Expr, could be derivated into a number value, which then could be either added or subtracted with a Factor, which also could derivate to a number symbol. Though, this left recursion, as mentioned earlier, could allow infinite recursion. This would make the grammar ambiguous. Ambiguous grammar is to be avoided, since it generates several different parse trees for the same input, in other words, the grammar allows different interpretations of the input stream.

Listing 3.2: Left-recursion

```
Expr := Expr '+' Factor | Expr '-' Factor | Factor
Factor := '(' Expr ')' | Val
Val := [0-9]*
```

To get rid of the left recursion, the grammar is rewritten into the listing in 3.3. Factor is moved first in the Expr rule since Factor can be derivated into a number. Then, there is added a new rule Expr′ which either scans the plus or minus terminals and then it goes back to the Expr rule or the epsilon which ends the production of rules. Through the Expr in the Expr′ rule, a number can be derivated. Since no rule has itself on the right side, then there is no left nor right recursion.

Listing 3.3: Removal of Left Recursion

```
Expr := Factor Expr'
Expr' := ( '+' | '-' ) Expr | ϵ
Factor := '(' Expr ')'
           | Val
Val := [0-9]*
```

For the sake of simplicity, the grammar of this project is unambiguous, which is defined in the code using EBNF notation. This provides more compact and readable documentation compared to the BNF notation [44]. See listing 3.4 for the grammar written in EBNF notation and note that 3.3 and 3.4 are equivalent.

Listing 3.4: The final grammar in 1. iteration written in EBNF notation

```
Expr := Factor ( ( '+' | '-' ) Expr )?
```

3.4 Typing

In programming, typing refers to the process of defining the data type of a variable or expression. A data type is a classification of data that specifies the type of operations that can be performed on the data, the size of the data, and the way the data is stored in memory [7]. As stated before, the language of this project is statically typed, due to the aforementioned storage limitation and run-time efficiency. In a statically typed language, all variables must be declared with a corresponding type, so any type mismatches or illegal operations are caught during compile time. Typing is further discussed in chapter 6.

Dynamic vs static typing

The concept of typing in programming can be divided into two main categories: static typing and dynamic typing. In a statically typed language, every variable has a type that can be computed at compile time without actually running the code. This makes type checking at compile time possible. In statically typed languages, the validity of arithmetic and logical operations is checked at compile time, recognising potential mismatches, and providing helpful error messages to the developer. Knowing types during compilation can also result in faster code execution since the type information is already known, and the compiler can optimise the code accordingly.

Dynamic typing, on the other hand, allows variables to hold values of any type and enables the type of a variable to be determined at runtime. Variables are also allowed to change their data type at run time and the type of a variable is not specified explicitly in the code. In this approach, the data type of a variable can change during the program's execution, depending on the value assigned to the variable. However, dynamic typing requires more memory, which is a major disadvantage when working with old hardware, such as the MOS 6502. It requires more memory because dynamic typing needs an interpreter to check the types at run time. Furthermore, since dynamic type checking is performed at run time, type errors are caught when the code is executed. This can result in slower code execution since the type information must be checked every time the code is executed. While static typing can make the code more verbose, it can also help with readability, and catching errors earlier in the development process, when they are easier to fix [7].

For this project, static typing is selected since it requires less memory. It also enhances the performance of the program at run time because the target code can be optimised during compilation. Finally, it also leads to fewer errors at run time, such as type errors and null pointer exceptions.

Implementation of Language Design

The core of a programming language is its syntax and token specification. It provides the rules for how the program must be written. In this section, the language's formal and informal syntax and token specifications are explained.

Syntax Specification

The language's syntax is built by zero or more statements followed by an end-of-line and end-of-file tokens. A statement is either a declaration, if statement, or assignment all followed by end of line token.

A declaration is made of a type and identifier and optionally followed by an assign token and an expression.

An expression has a hierarchy of expressions that should be evaluated first. Since it is an abstract syntax tree (explanation in section 5) that is being traversed depth first left-to-right, the deeper expressions have higher priority and get evaluated first. The expression's hierarchy is as follows:

- Or operator "||"

- And operator "&&"
- Equality and Inequality operators "==" and "!="
- Greater Than, Less Than, Greater Equal, and Less Equal operators ">", "<", ">=", and "<="
- Plus and Minus operators "+" and "-"
- Not operator "!"

Lastly, the expression ends with a factor that could be a boolean, int, float, or id. Otherwise, the expression could also be returned to the top of the hierarchy wrapped with parentheses.

The if statement takes an if token with an expression in parentheses followed by a block. Optionally, an else block could come after, and it would contain a block like the if statement. A block is wrapped in curly braces and consists of zero or more statements.

The formal syntax specification is formulated following the EBNF notation. The non-terminals always start with a capital letter and the terminals are all lowercase letters. The syntax specification is shown in listing 3.5 below:

Listing 3.5: EBNF of the language

```
Prog := Stmt* eof eol
Stmt := id assign Expr eol
        | print lParens id rParens eol
        | IfStmt eol
        | Dcl eol
Dcl := Type id (assign Expr)?
Expr := OrOp
OrOp := AndOp ( ( or ) Expr )?
AndOp := EqualityOp ( ( and ) AndOp )?
```

```
EqualityOp := ComparisonOp ( ( eq | ne ) EqualityOp)?
ComparisonOp := ArithmeticOp ( ( lt | gt | le | ge )
    ComparisonOp )?
ArithmeticOp := NegationOp ( ( plus | minus | multiply |
    divide) ArithmeticOp )?
NegationOp := ( not )? Factor
Factor:= lParens Expr rParens
         | Val
IfStmt := if lParens Expr rParens lBrace Block rBrace ( else
    lBrace Block rBrace )?
Block := Stmt*
Type := float
        | int
        | boolean
Val =: intNum
       | floatNum
       | true
       | false
       | id
```

Token Specification

The formal specification of tokens is separated into different types of tokens. The skip tokens in listing 3.6 are the tokens that the parser should ignore.

Listing 3.6: Skip tokens

```
" "
"\t"
"\n"
"\r"
< COMMENT : "/*" (~["/"] | "/" ~["/"])* "*/" >
```

The tokens with numbers and id in listing 3.7 include the declaration of types, the regular expression for the integers and floats, the plus and minus operator, and the regular expression for what is allowed in the id.

Listing 3.7: Numbers, id, and arithmetic operation tokens

```
< INTDCL: "int" >
< INT: ( < DIGIT > ) >
< FLOATDCL: "float" >
< FLOAT: ( < DIGIT > ) (".") ( < DIGIT > ) >
< DIGIT: ( "0" | ["1"-"9"] ( ["0"-"9"] )* ) >
< PLUS: "+" >
< MINUS: "-" >
< MULTIPLY: "*" >
< DIVIDE: "/" >
< ID: ( ["a"-"z", "A"-"Z", "_"] ) ( ["a"-"z", "A"-"Z", "_",
    "0"-"9"] )* >
```

The logic tokens in listing 3.8 contain all the logical operators, the boolean value and its declaration token, and the if-else tokens.

Listing 3.8: Logic tokens

```
< BOOLDCL: "boolean" >
< TRUE: "true" >
< FALSE: "false" >
< IF: "if" >
< ELSE: "else" >
< AND: "&&" >
< OR: "||" >
< NOT: "!" >
< EQ: "==" >
< NE: "!=" >
< LT: "<" >
< LE: "<=" >
< GT: ">" >
< GE: ">=" >
```

Lastly, in listing 3.9, the tokens with parentheses, braces, end of line and file, the assigned token and the print keyword token are present.

Listing 3.9: Assignment operation & basic control structures

```
< ASSIGN: "=" >
< LPAREN: "(" >
< RPAREN: ")" >
< LBRACE: "{" >
< RBRACE: "}" >
< END_OF_LINE: ";" >
< END_OF_FILE: "BYE" >
```

The different types in the language are, as previously specified, integer, float and boolean, and can be seen in listing 3.7 and 3.8. It can also be deducted that the integers can be any whole number in the natural number set. But since the data bus only has 8 bits, the integers are restricted to values ranging from 0 - 255, because the language does not include negative values. The float type can be any rational number but is also restricted by the data bus. Since the 6502 processor does not have a float type, the float type introduced in the new language uses a fixed-point representation using five digits for the integer part, and three digits for the fractional part. This makes the values range from 0 - 31.875 with a precision of one-eighth.

3.5 Code Example

In this section, a piece of code written in the formally defined programming language is shown and explained. The language may be understood from the formal definition alone, but it is easier to convey and understand the rules of the language with an example. The code below, in listing 3.10, includes most of the fundamental parts of the first iteration of the grammar in action.

Listing 3.10: Code example written in the defined language.

```
int a = 2;
boolean c = false;
if (a == 2 && !c) {
    c = false;
} else {
    a = 5;
}
BYE;
```

On the first line of the code, an integer variable is declared with the id "a", and it is initialised with a value of 2. The declaration is concluded with an end-of-line notation, which is a semi-colon. Afterwards, a boolean variable "c" is declared and initialised with the value "false". Note that a variable can also be declared without initialising. Then, an if-else-statement is created with the "if" notation, conditions within parentheses, and a body which has code within it and is enveloped by curly brackets. An optional else statement is also included with another body of code enveloped by curly brackets. The program stops running when the end-of-file notation is reached.

Chapter 4

Compiler Design

Before diving into the documentation of the work behind the first iteration of the compiler, this section gives a high-level overview of what a compiler is. In the following paragraphs, the fundamental components are presented, which are subsequently implemented in this project's compiler, which source language is the one described in section 3, the implementation language is Java, and the target language is Assembly 6502.

Firstly, the compiler crafted and documented in this paper is a multi-pass compiler. In comparison to single-pass compilers, multi-pass compilers are less resource efficient, but a lot more readable and maintainable, which is why this approach was selected [48].

In a multi-pass compiler, the compilation process consists of multiple passes through the AST, each responsible for a specific task, such as semantic analysis and code generation. The AST is built using the tokens created by the lexer, and the language's production rules. The tokens are the symbols used to define a programming language syntax. This process of recognising the syntactic structure of the source code is the main part of the syntax analysis. To further understand the following explanation of the chronological order of the phases of a compiler, see the following figure 4.1.

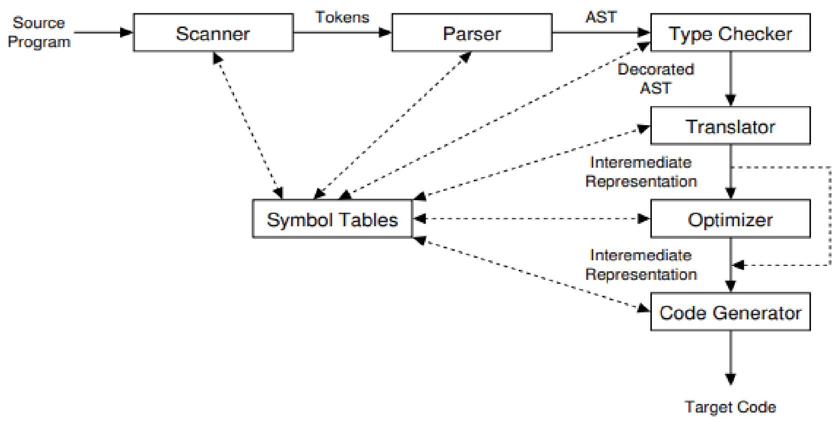


Figure 4.1: The phases of a syntax-directed compiler[16]

Note, that figure 4.1 represents a generally applicable structure, but in the compiler of this project, the translator and optimiser phases are skipped, and the lexer and parser do not interact with the symbol table in any form.

Afterwards, the semantics of the program is analysed in regard to its syntactic structure. This includes checking the correctness of the code by enforcing type and scope rules. Optionally the source language can be translated into an intermediate representation (IR) of the program, which serves as an input to the code generator component that creates the desired target language. Optionally, the representation may be transformed by an optimiser, which can lead to the generation of a more efficient program.

Each component nested within these steps is summarised in short below, and then expanded upon as they become relevant throughout the paper in the following chapters.

Lexer

During this phase, the lexer reads the source code character by character and groups them into tokens [16]. These are then encoded and given as input to the parser, which then starts the syntactic analysis.

Before a token can be built it should be described first, a preferred way to do this is with regex 3.2. The lexer can be made when the tokens have been specified.

Parser

The parser is built by using the rules from the language's CFG. It then takes the tokens and groups them into phrases in regard to the syntax specification [16]. The parser validates the correctness of the provided code based on the syntactic rules of the language. In the case that an error is found, a relevant message is issued. Note that in this project, an AST is constructed and used in subsequent phases of the compiler, rather than reducing a generated parse tree.

Symbol Tables

In order to allow information to be associated with identifiers and share this information with various phases of the compiler, a mechanism such as a symbol table can be implemented [16]. The phases of the compiler use the symbol table to store, share, and retrieve information about the identifier's type and scope.

Type Checker

The nodes of the AST have their static semantics checked by the type checker. These include the properties which the type checker can determine before the program is executed [16]. This includes checking in the symbol table whether the relevant identifiers are declared and whether their types are correct. If this process completes without errors, the nodes are decorated by the type checker, which means that additional type information is added to the AST.

Translator

After the semantics of the nodes have been verified as being correct, they can optionally be translated into IR code. This intends to implement the meaning of the node [16]. An example of this is that a node in the AST can represent the same loop body as the expression in a while loop, but AST lacks the notion that the loop is being repeated. This is what is intended to be fixed during this phase. The IR created by the translator is used for optimisation which will be discussed in the next *Optimiser* section. Where it will be stated that an IR was not created, indicating that a translator was not implemented for this project.

Optimiser

This phase transforms the IR code created by the translator into improved IR code, but it is still functionally the same [16]. If used correctly, the optimiser can significantly increase the efficiency of the program execution. This is done by either simplifying, moving, or completely removing unnecessary computations. For this project, the AST was reduced with a method called constant folding, which will be discussed later in chapter 13.

Code generator

This phase also uses the IR code produced by the translator, but here it is mapped to the target language by the code generator [16]. To do this operation, detailed information about the target machine is required, which includes register allocation and code scheduling. As mentioned earlier the code generation phase for this project uses a reduced AST.

Chapter 5

Syntax Analysis

Once the grammar itself is clearly defined using context-free grammar, a lexer and a parser can be created to check any program's syntactic correctness and produce a higher-level representation of the input. During the syntax analysis, it is important to note that only the syntactic correctness is validated, based on the context-free grammar; potential semantic issues are thus not checked yet. From a high-level perspective, other than checking syntactic correctness, this phase of compiler crafting is also significant, as it results in an AST. The abstract syntax tree is responsible for providing a higher-level representation of the given program. In this section, the lexer, parser, and AST generation are presented and discussed.

5.1 Lexer

A lexer is a component of a compiler that reads the source code of a specific programming language and produces a stream of tokens (terminal symbols). The tokens can be seen in the token specification 3.4, and they are passed on to the parser one by one. The primary purpose of the lexer is to identify the individual tokens of specific categories in the source code, such as identifiers, operators, keywords, literals, etc. Once the AST is generated, the terminal symbols can be seen as the edge nodes of the tree.

The lexer's tasks are the following (all points are cited verbatim from Fischer et.al.'s book [16]):

- It puts the program into a compact and uniform format (a stream of tokens).
- It eliminates unneeded information (such as comments).
- It processes compiler control directives (for example, turn the listing on or off and include source text from a specified file).
- It sometimes enters preliminary information into symbol tables (for example, to register the presence of a particular label or identifier).
- It optionally formats and lists the source program.

While manually writing a lexer is an educational exercise, it is a repetitive and time-consuming process. Additionally, in this process, human error is more probable than any issue with a trusted lexer generator capable of handling CFG. A lexer generator such as the one built into JavaCC can also provide helpful error-handling mechanisms [3]. On the other hand, as with libraries and other forms of third-party code, the development team may not be sufficiently familiar with the inner workings of the package. Therefore, the generated lexer may be more complex and more difficult to understand for a development team than a manually crafted one.

5.2 Parser

The purpose of the parser is to check the validity of the structure of the provided code. The code is passed as tokens according to the production rules of the particular language, organising them into a parse tree that reflects the hierarchical structure of the language. More specifically, the parser reads the tokens passed by the lexer and groups them into phrases based on the CFG. If the syntax of the input does not conform to the rules of the programming language, the parser reports the errors it finds. If the input is valid, the parser generates a parse tree.

The generated parse tree often contains a lot of information. It is generally advised to be condensed into an AST for further work. This has been done for this project by reducing the size and complexity of the input code's representation, and very likely resulting in simpler tree traversal.

Similar to creating a lexer, manually crafting a parser has some disadvantages regarding time and effort and human errors. However, the automatically generated third-party code might be more difficult to comprehend and debug (thereby potentially also harder to maintain), and the compiler now has a dependency (although not runtime related), which could result in a lot of problems regarding future development if it is discontinued or not functioning properly. Furthermore, all parser generators have their limitations. During the research, several parser generators were explored and tested. The following list encapsulates the essential reasons for discarding them:

- ANTLR $\rightarrow$ A top-down recursive descent parser (LL(*) (Left-to-right, Leftmost derivation)), that allows Extended Backus–Naur Form grammar specification and automatic tree generation [14]. ANTLR initially proposes a manageable learning curve, but it produces a concrete syntax tree 17, which then must be traversed and reduced to AST form [5]. In comparison to other options, where an AST can be constructed using action code, in ANTLR the user must reduce the concrete syntax tree directly, which for this project is an unnecessary challenge. Although there are a few community-made plugins and extensions [29] to aid in handling the generated parse tree, it still does not seem like an easy task. Lastly, it was concluded that ANTLR does not have user-friendly error messages during parser generation in comparison to other options.

- JCup (Yacc for Java) $\rightarrow$ JCup does not allow EBNF, and it is better suited for complex grammars. It is also an LALR (Look-Ahead LR) bottom-up, which is more complex to understand and work with, compared to LL parsers [14]. Bottom-up LALR parsers can also struggle with grammar that has left recursion. This can require additional workarounds or modifications to the grammar.
- SableCC $\rightarrow$ Has poor documentation and little to no community support to make up for it. It does not generate a lexer, and again, the LALR parser is difficult to work with in comparison to an LL parser.

The parser generator used for this project was JavaCC, which is primarily designed for LL(1) languages and has no support for left-recursive grammar. It is possible to create a LL(k) parser with JavaCC. This means that the parser can look multiple tokens ahead before determining which production rule is used. JavaCC by default generates a top-down recursive decent parser with a single token lookahead. When selecting a lexer and parser generator, or preferably even before designing the language itself, one must consider the generators' characteristics. Some of the important characteristics of JavaCC generated parser can be seen below [3]:

- Allows EBNF grammar specifications
- Top-down recursive descent parser
- LL(1) by default but allows LL(k)
- Generates pure Java code, without any runtime dependencies
- Provides error handling

In the following paragraphs, the characteristics of JavaCC are discussed to highlight why it was selected as this project's lexer and parser generator.

Firstly, JavaCC allowing EBNF specifications enables a simpler notation of the same grammar. EBNF notation allows for optional parts and one of-choices. An example of a simpler, more concise notation is the following:

Listing 5.1: BNF and EBNF notation for the same grammatic rules

```
BNF:
<expr> -> <expr> + <term>
          | <expr> - <term>
          | <term>
<term> -> <term> * <factor>
          | <term> / <factor>
          | <factor>

EBNF:
<expr> -> <term> {(+ | -) <term>}?
<term> -> <factor> {(* | /) <factor>}?
```

In the example above 5.1, terms and expressions are defined firstly using BNF and then EBNF. The two examples encapsulate the same logic, but EBNF notation is arguably more concise and readable.

As mentioned above, JavaCC generates an LL parser. This is an important detail, as the choice between working with an LL (Left-to-right, Leftmost derivation) or LR (Left-to-right, Rightmost derivation) parsing technique should be based on the purpose and context of the project. Generally, the LL parsing algorithm is simpler to implement than the LR parsing algorithm [12]. Moreover, LL parsers do not use parse tables, but parsing functions. This stands in contrast to LR parsers, which use parse tables to determine the appropriate production rule to apply at each step of the parsing process [40]. This means that the LL parser is faster and requires less memory. If, in the future, the compiler was made available to run on the MOS 6502 machine, this form of the LL parser would be the better choice. Another option for LL parsing is a top-down parsing technique that uses a table-driven approach to predict the production rule to be applied based on the next input symbol. The LL parsing table is typically smaller than the table used in LR parsing, as LL parsing relies on a more restricted grammar subset that makes it easier to predict the next production rule [53]. On the other hand, LR parsing is a bottom-up parsing technique that builds a parse tree from the input symbols by using a state machine to shift and reduce symbols [54].

Generating pure Java code with proper error handling and without any dependencies is also a very helpful property. Firstly, catching parsing errors is made a lot easier with human-readable error messages, and not having any dependencies means that the system does not rely on any third-party code, which could potentially be outdated or discontinued. Furthermore, there are no libraries or such, that the developers must understand before reading through the generated code, which is also fairly readable, and if needed, not too difficult to alter.

Implementation of the Lexer and Parser

The lexer and parser are written in pure Java code and have no dependencies. They are both generated by JavaCC, which uses the aforementioned grammar file that included the grammar specification 3.4. The lexer and parser generation is executed by typing the following commands in the terminal. The first is *javacc parse.jj* which runs the JavaCC tool on the grammar file (parse.jj). *javac Compiler.java* then compiles the Java source file generated by JavaCC in the previous step. During this step, class files are generated. *java Compiler* runs the parser class, which is the entry point for the parser. For ease of use, these commands have been written into a script (shell/bat) that can be run with one click. The generated code also has error handling and is fairly readable; see the example below (5.2).

Listing 5.2: Code example of parser code, generated with JavaCC.

```
final public Node Val() throws ParseException {
    Token t;
    switch ((jj_ntk==-1)?jj_ntk():jj_ntk) {
    case INT:
      t = jj_consume_token(INT);
    {if (true) return new IntNum(t.image);}
      break;
    case FLOAT:
      t = jj_consume_token(FLOAT);
    {if (true) return new FloatNum(t.image);}
      break;

    case ID:
      t = jj_consume_token(ID);
    {if (true) return new Id(t.image);}
      break;
```

This code is part of the parser, generated by JavaCC. The variable *jj_ntk* represents the lookahead token, and it is set to -1 by default which means the parser has not yet determined a look-ahead token. The question mark *?* is a ternary operator, and it checks whether the *jj_ntk* variable is set to -1, and if it is, then the *jj_ntk()* method is called to determine the actual lookahead token. In the case that *jj_ntk* is not -1, then it is used as the look-ahead token. Afterwards, a switch statement evaluates the lookahead token and determines which production rule to apply based on its value.

5.3 Abstract Syntax Tree

An AST, as mentioned earlier, is a tree representation of the abstract syntactic structure of a code written in a formally defined language. It serves as the central data structure for all post-parsing activities for this project. It concludes the syntax analysis and connects it to the next phase, contextual analysis, in which the context of the correct syntax is evaluated. The AST is traversed, decorated, and potentially transformed during subsequent phases of the compiler. An AST should be more concise than a parse tree, but it must include all the information needed for post-parsing activities. Although, theoretically, it is possible to squeeze semantic analysis into a parser, in practice it should be avoided, as each of these components need separate treatments in the compiler. Separation of concerns is vital for a maintainable and scalable application.

At the start of the project, JavaCC's built-in JJTree AST generator was used, but it was promptly replaced with a manually coded AST generator and a pretty printer. JJTree was discarded to have more control over the AST representation itself, as it will be important in subsequent phases. The visitor pattern is implemented to separate concerns and create respective classes for specific purposes rather than having everything jammed into one class. The visitor design pattern allows for the addition of new operations to an object structure without modifying the objects themselves. The pattern defines a way to separate the algorithm from the objects it operates on [45]. In the product, the objects in question are the nodes of the AST, which require the *accept* method to allow the visitor to visit the node. Without the visitor design pattern, the nodes would contain large amounts of logic, and the overall scale-ability of the program would suffer because of the worsened modularity. JJTree generates complex parse trees that can be difficult to understand, traverse and maintain, and this complexity can make it harder to modify and extend the grammar in the future. If the grammar for the language is changed, JJTree changes the AST as well, and the methods used to handle the AST like code generation and type checking must be updated too.

The first step in constructing an AST is defining unambiguous grammar, see the language design section 3. A step taken to avoid ambiguities was for instance the formerly described transformation of left- or right-derivative production rules 3. Next, the level of abstraction is to be determined, discarding grammar details concerned with disambiguation, as well as semantically useless symbols and punctuation (for instance EOL or EOF tokens). Beware that sufficient information must be preserved to allow the compiler's subsequent phases to perform their work. The essential data kept for future phases of the compiler are the hierarchy, node types, and their values. Next, the action code is placed in the grammar to add nodes and their children to the root of the tree.

In order to ease the subsequent semantic analysis and code generation-related work, the construction of the AST is intertwined with a gang of four (GOF) type visitor pattern. In a GOF visitor pattern type checking, scope checking, code generation, and pretty printing are separated into their own classes and are implementing their own visitors. The visitor pattern is a design pattern in object-oriented programming that allows for the separation of the algorithm or logic from the data structure it operates on [16]. It is often used when the data structure being operated on is complex, such as a tree or graph, and there are multiple operations or algorithms that need to be performed on it. Each node has a method for visiting it. The type and scope checking functions, as well as segments related to code generation, has their own classes, to improve segmentation and to ease future work. Using a visitor pattern, phase and node-specific code can be invoked cleanly, while separating the functionality of a phase, such as code generation or type checking into a single class.

An advantage of using a visitor pattern is that implementing new operations is, in theory, an easy task. New operations can be created by adding a new visitor. Furthermore, as previously mentioned, related operations are logically gathered together in their respective sub-classes of the visitor. However, adding a new visitor class to the object structure can prove difficult, as each visitor must be recompiled with an appropriate method for the new class. It also makes the compiler slower because the AST must be traversed for every visitor class. Considering the disadvantages of using the GOF visitor pattern, it was still seen as an optimisation because of the abstraction it provides to the compiler.

5.4 Pretty Printer

As a visual representation for the developers and potential end-users of the compiler, as well as to be able to check the correctness of the generated tree, a simple pretty printer is also written. By comparing the theoretically correct AST of some simple programs to the output, an educated guess can be made regarding the correctness of the implementation of the language's context-free grammar. Note that the correctness of the AST will also be further tested using unit tests 8.

Implementation of the Pretty Printer

The following piece of code 5.3 illustrates the implementation of the pretty printer used in the compiler of this project. This class implements a *Visitor* interface, which defines methods used to visit each type of node in the AST. The current code snippet shows an implementation for the methods and properties of the pretty printer class.

Listing 5.3: Implementation of Pretty Printer.

```
public class PrettyPrint implements Visitor {
    private StringBuilder sb = new StringBuilder();
    private int indentLevel = 0;

    private void indent() {
        this.indentLevel++;
    }

    private void unindent() {
        this.indentLevel--;
    }

    private void printIndent() {
        for (int i = 0; i < this.indentLevel; i++) {
            sb.append("␣␣␣␣");
        }
    }

    public String getResult() {
        return sb.toString();
    }
```

The *PrettyPrint* class has a *StringBuilder* object *sb*, which is used to build the output string. The *indentLevel* variable keeps track of the current indentation level of the output string. The *indent()*, *unindent()*, and *printIndent()* methods are used to manage indentation levels in the output string, which is returned with the *getResult* method.

Listing 5.4: The function for comparison operation in the Pretty Printer class

```
@Override
public void visit(ComparisonOp node) {
    node.getLeft().accept(this);
    sb.append("␣" + node.getOperator() + "␣");
    node.getRight().accept(this);
}
```

The *visit()* method for *ComparisonOp* is seen in 5.4. The method first visits the left child, which ultimately is either an id, integer, float or boolean. The method *accept* triggers the *visit* method inside the left node's class, which adds the value to the string builder. After, the operator of the comparison node is also appended to the string builder. Furthermore, the node's right child is accepted like the left child and is appended to the string builder.

Overall, this implementation of the *Visitor* pattern for a *PrettyPrint* class provides a simple way to print the contents of the AST in a readable format.

Chapter 6

Semantic Analysis

During lexical and syntactic analysis, the lexer and parser ensure that the input conforms to the language's token and CFG specifications. While this is a good starting point for ensuring overall correctness, there are still important aspects which have not been checked yet; type and scope rules. The AST output of the lexical and syntactic analysis is therefore traversed and decorated with relevant data, and all the variable's types and scopes are validated through a series of correctness checks. Decoration means that new information is added to nodes [16]. By traversing this AST, the part of the compiler responsible for semantic analysis examines a program's correctness based on its syntactic structure - the semantics of the program.

This chapter includes both the formal semantics and type and scope checking rules. Note, that only the first iteration of the semantic rules are included in this chapter. The rest of the rules are included in the second iteration.

Metavariables

Metavariables are variables used in place for variables such as arithmetic- and boolean expressions. The metavariables and their definition for the language are as follows:

$$n \in \mathbb{Q} \quad x \in Var \quad a \in Arithmetic_{exp} \quad b \in Boolean_{exp} \quad S \in Statements$$

6.1 Type Rules

This section includes the type rules for the semantics of the language and consists of both the expressions and statements. It can be deduced that the scope used for the first iteration is monolithic because there is only one environment. Furthermore, the language uses explicit static binding, meaning that variables are explicitly declared with a type and an identifier before they can be used. At compile time, we can relate all the binding occurrences (declarations) to applied occurrences (uses).

Expressions

The environment for the language can be written as:

$$E : Var \rightharpoonup \{Bool,\ Int,\ Float\}$$

The expression above states that the variables of the environment can be of type Bool, Int or Float as mentioned earlier, and the set can be seen below.

$$T = \{int,\ float,\ boolean\}$$

$$E \vdash e : T$$

The expression above maps the expressions of the language to their designated types. The type rules for the expressions accepted by the language can be seen below:

$$[Var_{EXP}] \quad \frac{E(x) : T}{E \vdash x : T}$$

$$[Add_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 + e_2 : T}, \quad T \in int, float$$

$$[Subs_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 - e_2 : T}, \quad T \in int, float$$

$$[EQ_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 == e_2 : bool}, \quad T \in int, float$$

$$[NE_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 \neq e_2 : bool}, \quad T \in int, float$$

$$[LT_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 < e_2 : bool}, \quad T \in int, float$$

$$[GT_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 > e_2 : bool}, \quad T \in int, float$$

$$[LE_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 <= e_2 : bool}, \quad T \in int, float$$

$$[GE_{EXP}] \quad \frac{E \vdash e_1 : T \qquad E \vdash e_2 : T}{E \vdash e_1 >= e_2 : bool}, \quad T \in int, float$$

$$[And_{EXP}] \quad \frac{E \vdash e_1 : bool \qquad E \vdash e_2 : bool}{E \vdash e_1 \ \&\& \ e_2 : bool}$$

$$[Or_{EXP}] \quad \frac{E \vdash e_1 : bool \qquad E \vdash e_2 : bool}{E \vdash e_1 \,||\, e_2 : bool}$$

$$[Not_{EXP}] \quad \frac{E \vdash e_1 : bool}{E \vdash !\, e_1 : bool}$$

Statements

This subsection includes the statements that the language accepts. For the first iteration, only the if and if-else statements are implemented besides the assignment statement. The while-loop seen in these semantics is planned to be implemented in the second iteration, but the semantics for it are defined while working on the first iteration. The types of statements can be seen in section 6.1. The metavariable sets can be seen below:

$$a ::= n \quad | \quad a_1 + a_2 \quad | \quad a_1 - a_2 \quad | \quad (a_1)$$

$$b ::= a_1 = a_2 \quad | \quad a_1 < a_2 \quad | \quad a_1 <= a_2 \quad | \quad a_1 > a_2 \quad | \quad a_1 >= a_2 \quad | \quad b_1 \;\&\&\; b_2 \quad |$$
$$b_1 \;||\; b_2 \quad | \quad \neg b_1 \quad | \quad (b_1)$$

$$e ::= a \quad | \quad b$$

$$S ::= T\ x = e \quad | \quad if\ b\ then\ S \quad | \quad if\ b\ then\ S_1\ else\ S_2 \quad | \quad while\ b\ do\ S$$

Below are the type rules for the statements in the set of statements S:

$$[Ass_{STM}] \quad \frac{E \vdash x : T \quad E \vdash e : T}{E \vdash x = e : ok}$$

$$[If_{STM}] \quad \frac{E \vdash e : bool \quad E \vdash S_1 : ok \quad E \vdash S_2 : ok}{E \vdash if\ e\ then\ S_1\ else\ S_2 : ok}$$

$$[While_{STM}] \quad \frac{E \vdash e : bool \quad E \vdash S : ok}{E \vdash while\ e\ do\ S\ \ : ok}$$

Big-step Semantics

This section includes the big-step semantics for the language. It showcases how the computations in the programming language should be expressed. Big-step semantics is a method of defining the semantics of a programming language formally. By specifying how the program transitions from an initial state to a final state, the overall behaviour of the program can be described [24]. The big-step semantics for the first iteration and the while-loop can be seen below:

$$[Num_{BS}] \quad s \vdash n \rightarrow_A v \quad , \quad v = \mathbb{N}[|n|]$$

$$[Var_{BS}] \quad s \vdash x \rightarrow_A v \quad , \quad s(x) = v$$

$$[Assign_{BS}] \quad \langle x := a, s\rangle \rightarrow s[x \mapsto v] \quad , \quad s \vdash a \rightarrow_A v$$

$$[Plus_{BS}] \quad \frac{s \vdash a_1 \rightarrow_A v_1 \quad s \vdash a_2 \rightarrow_A v_2}{s \vdash a_1 + a_2 \rightarrow_A v}, \quad v = v_1 + v_2$$

$$[Minus_{BS}] \quad \frac{s \vdash a_1 \rightarrow_A v_1 \quad s \vdash a_2 \rightarrow_A v_2}{s \vdash a_1 - a_2 \rightarrow_A v}, \quad v = v_1 - v_2$$

$$[Paren_{BS}] \quad \frac{s \vdash a \rightarrow_A v}{s \vdash (a) \rightarrow_A v}$$

$$[Comp_{BS}] \quad \frac{\langle S_1, s\rangle \rightarrow s'' \quad \langle S_2, s''\rangle \rightarrow s'}{\langle S_1; S_2, s\rangle \rightarrow s'}$$

$$[if - T_{BS}] \quad \frac{\langle S_1, s\rangle \rightarrow s'}{\langle if\, b\, then\, S_1\, else\, S_2,\, s\rangle \rightarrow s'} \; , \; s \vdash b \rightarrow_B T$$

$$[if - \bot_{BS}] \quad \frac{\langle S_2, s\rangle \rightarrow s'}{\langle if\, b\, then\, S_1\, else\, S_2,\, s\rangle \rightarrow s'} \; , \; s \vdash b \rightarrow_B \bot$$

$$[while_{BS}] \quad \frac{\langle S, s\rangle \rightarrow s'}{while\ b\ do\ S \rightarrow s'}$$

6.2 Symbol Table

Although the AST includes all the relevant syntactic details about the program, it does not keep track of scopes or types, therefore a symbol table was implemented. A symbol table is a data structure used to keep track of the variables in a program, and their types. The symbol table implemented in this compiler included the type, scope, and memory address of the given variable. It was implemented as a hash map with the name of the variable as the key and a class called symbol as the value. The symbol class contains attributes for type, scope, and memory address. An example of how to add a variable of type boolean to the symbol table is shown in listing 6.1.

Listing 6.1: Code for BoolDcl node in the symboltable class.

```
@Override
    public void visit(BoolDcl node) {
        if (symbolTable.get(node.getId()) == null) {
            symbolTable.put(node.getId(), new
                Symbol(node.getId(), new BooleanType(),
                scopeLevel));
        } else {
            error("variable␣" + node.getId() + "␣is␣already␣
                declared");
        }
    }
```

Since the scope is monolithic 6.4, the scope level will always be 0. If a block or nested scope is implemented, the symbol table must be changed accordingly.

6.3 Type Checking

The objective of type checking is to check the static semantics of each AST node. The type checker must verify that the construct of any particular node is legal. Legality in this case means that all identifiers are declared, and types are correct and consistent with the operations being performed on them. If the node is semantically correct, the type checker decorates the node on the AST by adding type information to it. In case of an error, an appropriate error message should be issued [16].

From a higher-level perspective, the purpose of type checking is to catch errors during compile time rather than at run time, which can potentially save time and prevent unforeseen bugs and crashes as mentioned in section 3.1. The following is an example of how type checking is performed on an assignment:

$$int\ a = 4 + 3$$

Firstly the Var_{EXP} rule from 6.1 is used on the variable a to get its type:

$$[Var_{EXP}] \quad \frac{E(a) : int}{E \vdash a :\ int}$$

Then the $[Add_{EXP}]$ is used on the 4 + 3 part of the assignment:

$$[Add_{EXP}] \quad \frac{E \vdash 4 :\ int \quad E \vdash 3 :\ int}{E \vdash 3 + 4 :\ int}$$

Lastly, the $[Ass_{STM}]$ rule is used on the entire assignment:

$$[Ass_{STM}] \quad \frac{E \vdash a :\ int \quad E \vdash 3 + 4 :\ int}{E \vdash x = 3 + 4 :\ ok}$$

6.4 Scope Checking

Before going into depth about the scope checking part of this compiler, the scoping types are explained. Scoping types can be broken down into different kinds of block structures [51] and can be seen in figure 6.1:

- Monolithic: Declared variables cannot be declared again. In this case, the only block is the entire program. Only one symbol table is required.
- Flat: There can be a local declaration and a global declaration, but no nested declarations. In this case, the program can be subdivided into several disjoint blocks. One symbol table is required for global scope, and one for each block or a single symbol table with flags to identify if a symbol is on the global or local structure.
- Nested: There can be local declarations within scopes, etc. In this case, one can have any nesting. When looking at the program representation, the compiler must start with the innermost scope and look for a declaration going to the next outer scope every time. It is a tree-like structure.

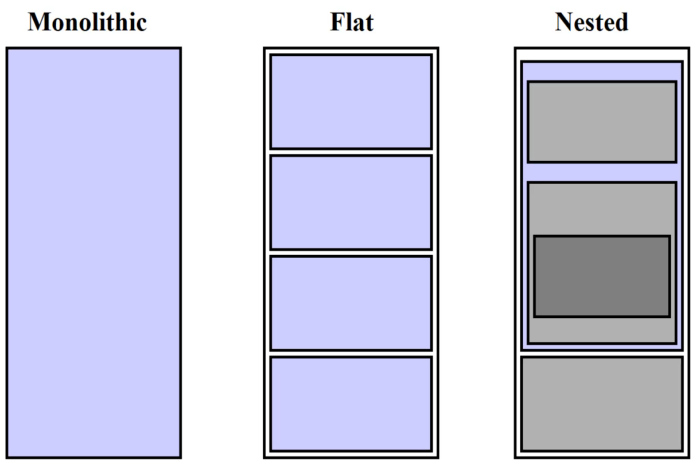


Figure 6.1: Different kinds of scope structures [51]

A monolithic block structure is chosen for the first iteration of this project to keep the scope checking relatively simple.

The objective of scope checking is to ensure that the variables and other identifiers used in a program are declared and accessible within their respective scopes. The scope checker looks at all usages of variables and identifiers and verifies their validity. This is, in most cases, done by finding a usage; firstly checking the closest scope for a declaration and then recursively checking outer scopes. This also implies that the language is considered statically scoped, so scope checking is done at compile time. Static scoping is enabled by static binding, which means that at compile type, the compiler can relate all the applied occurrences to binding occurrences. The technique however depends on the implemented block structure. If the validation is successful, the respective node on the AST is decorated with scope information.

Such as with type checking, the purpose of scope checking is to catch errors related to the use of variables and identifiers at compile time, which can save time and prevent run time errors. By ensuring that variables and identifiers are defined and accessible within their appropriate scopes, the compiler can at times also optimise a program's performance and reduce memory usage.

For the first iteration, the scope checking is implemented in the symbol table, as the compiler only has to check the global scope because of the monolithic structure.

E.g. the scope checking for a boolean declaration 6.2:

Listing 6.2: Scope checking for boolean declarations.

```
@Override
public void visit(BoolDcl node) {
   if (symbolTable.get(node.id) == null) {
        symbolTable.put(node.id, new Symbol(node.id, new
            BooleanType(), scopeLevel));
    } else {
        error("variable␣" + node.id + "␣is␣already␣declared");
    }
}
```

This method checks the symbol table to see if it already has a declared variable with the same id, and if so, throws an error. This simple method is all that is needed to check the global scope in the first iteration, as all declared variables are in the same level of the symbol table.

Chapter 7

Code Generation

Code generation is the process of transforming an IR of a program into a target language. For this project, the aforementioned reduced AST is used in the code generation phase. The AST is not ideal for code generation, but considering the time frame of this project, it works fine. A transformation of the custom-designed programming language is taking place, where the desired result is code in the Assembly 6502 language to run it on the 6502 processor. For this project, this is done by traversing the reduced AST starting from the top and then calling itself recursively to generate its logically equivalent counterpart in the Assembly 6502 language.

7.1 Assembly 6502

The 6502 Assembly language is a mnemonic-based language consisting of 56 different instructions. It is used to write software for the 6502 microprocessor. The 56 instructions can be divided into three basic groups, the first group has the most general purpose instructions such as Store, Load, Add etc. The second group consists of read, modify, and write instructions, such as shift, increment, decrement etc. The third group consists of the rest of the instructions such as the stack and compare instructions. A full list of the instructions can be seen in appendix 17. Each instruction is represented by three characters followed by one or more operands. The operands specify which register or memory location is involved in the operation [50].

This section will provide some information on some of the instructions from Assembly 6502, which was used in the code generation phase. First of all, to load values into a register the instructions *LDX*, *LDY* or *LDA* are used. "LD" stands for load, and the X, Y and A stand for which register to load the value into. Some other instructions that relate to *LDX*, *LDY* and *LDA* are *STX*, *STY* or *STA*. These instructions store the value from the register into a memory location defined afterwards, like this: *STX $0100*, followed by the transfer instructions, which are used to transfer the value from one register to another. The instructions that are capable of doing so are *TXA*, *TAY*, *TYA* or *TAX*. It is also possible to transfer the value of the stack pointer to the X register with the *TSX* instruction and from the X register to the stack pointer with the *TXS* instruction.

The instructions mentioned just before are used to load and transfer data around. This section will include instructions which can manipulate the data like addition and subtraction. To add two numbers, Assembly 6502 has an instruction called *ADD* which adds a value, specified after the instruction, to the value in the accumulator. The instruction for adding three to the value in the accumulator is *ADD #3*. The same goes for the instruction *SBC*, which is for subtracting a number from the value in the accumulator. The Assembly 6502 language does not have instructions for multiplying or dividing two numbers.

To implement selective control structures, the Assembly 6502 language has instructions which can jump to a label if for example a specific flag is set. The instructions for branching used in this project are the following: *BCC*, *BCS*, *BNE* and *BEQ*. *BCC* jumps to a label if the carry flag is 0 and the *BCS* instruction jumps to a label if the carry flag is 1. The instruction *BEQ* jumps if the accumulator and a value is equal and *BNE* jumps if they are not equal. An example of how *BEQ* can be used is *BEQ label1*.

The operands are an important part of the instruction set. The operands can either be one of the three registers mentioned in chapter 2, or it can be a memory location specified using memory addressing. Memory addressing refers to the way that the processor accesses memory locations during program execution. Here is a list of some of the memory addressing modes [50]:

- **Absolute** $c000: This method takes the full memory location as an argument. Optionally it can take the value of the X or Y register and add it to the memory location before accessing it. So if the location is $c000 and we add the X register with a value of 1 to it, the actual location is going to be $c001.

- **Zero page** $c0: This method takes a single-byte address as an argument to access the zero page. Optionally it can take the value of the X or Y register and add it to the memory location before accessing it.

- **Immediate** #$c0: In immediate addressing mode, the operand is an immediate value. For example, the instruction *LDX* #$c0 loads the value $c0 into the X register. An immediate value can also be a constant value from 0 to 255.

- **Indirect** ($c000): The instruction specifies an address of a memory location which contains a 16-bit absolute address. So if the address given to the instruction is ($c000), then the value at that address is loaded as the target address for the instruction.

- **Implicit**: Some instructions have the memory location implied to them like

INX (increment register X).

The addressing modes used for this project are absolute, zero page, immediate, and indirect. This is shown in the different examples later in this chapter.

7.2 Implementation of Code Generation

In this section, we discuss the code generation class and how it uses the previously mentioned visitor pattern to traverse the AST. Each visit method generates the relevant instructions for that specific node. The instructions are appended to a local *stringBuilder* string in the *codeGenerator* class. The class has a method called *generateCode()* that appends *stringBuilder* to the file *output.txt*, which only contains Assembly 6502 code. The file's contents can then be given to a 6502 processor, and the processor will then run the code. Some of the nodes have children nodes and the order of the calls to the children is important for the Assembly code to be correct. For example, the *arithmeticOp* node must call the left child first, because the evaluation of expressions begins at the bottom of the tree and works its way up towards the root. The left child is either another *arithmeticOp* node or an *intNum* or *floatNum* node.

When a new node is introduced, it is shown as a declaration node in the AST. The declaration can either be an *intDcl*, *floatDcl*, or *boolDcl*. For the assembled code to know where the variables are stored, the individual memory address of the variables gets stored in the symbol table. The memory address is stored when the assignment operation node calls the declaration child and can be seen in section 7.2.

The three specific nodes that will be discussed in this section are the assignment operation node, arithmetic operation node, and if-else statement node.

Assignment Operation Node

The assignment operation node's job is to generate code for storing variables. The variables are stored on the stack by pushing the value of the accumulator to the stack using the *PHA* instruction. Because of this, the program is limited to a maximum of 255 different variables. The instruction stores the value of the accumulator on the memory address that the stack pointer is currently pointing to, and after storing the accumulator it decrements the stack pointer. Below is the code example in 7.1 for the assignment operation node in the code generation class:

Listing 7.1: Code generation for AssignmentOp node.

```
public void visit(AssignmentOp node) {
        if (node.getDeclaration() instanceof Id) {
            node.getExpression().accept(this);
            if (node.getExpression() instanceof ArithmeticOp)
               {
                codeBuilder.append("TAX\n");
            }
            storeXRegisterInVariable(((Id)
               node.getDeclaration()).getName());
        } else {
            node.getDeclaration().accept(this);
            node.getExpression().accept(this);
            if (!(node.getExpression() instanceof
               ArithmeticOp)) {
                codeBuilder.append("TXA\n");
            }
            pushAccumulator();
        }
        arithmeticOpCount = 0;
    }
```

The method first checks if its declaration node is an id, indicating that the variable has been declared before. If so, it simply goes into the expression node and stores the value of the expression in the X register. When the expression node's value is in the X register the node calls a method to store the X register's value in the id's location in memory.

If the assignment operation node is a new variable declaration it calls its declaration node. This goes into either the *intDcl*, *floatDcl* or *boolDcl* node and does the same for any of them, which is shown in 7.2.

Listing 7.2: Visit of FloatDcl node.

```
public void visit(FloatDcl node) {
        symbolTable.lookup(node.getId())
        .setMemoryAddress(stackAddress);
    }
```

In the *intDcl* node, which is shown in the example, the node looks up its id and sets the memory address attribute to the current value of the *stackAddress* for the specific id in the symbol table. The attribute *stackAddress* is a global value in the code generator class, that keeps track of what the actual *stackAddress* would be when running the Assembly code. After the symbol table is updated, the assignment operation node calls its expression node, which can either be an *intNum*, *floatNum*, *bool*, arithmetic operation, or comparison operation node. This node will either update the accumulator or the X register, which depends on what the expression node is. If the expression node is not an arithmetic operation node, then it appends *TXA* to the *codeBuilder* string. The *TXA* instruction is appended if, for example, the expression node is just an *intNum*, because the *intNum* node simply appends the string *LDX node.value* where *node.value* is the value of the expression. Furthermore, the assignment operation node calls the *pushAccumulator* method which appends the string *PHA* and decrements the *stackAddress* attribute.

Lastly, it resets the *arithmeticOpCount* by setting it to 0.

Arithmetic Operation Node

The arithmetic operation node's job is to generate Assembly code for addition and subtraction. It does so by calling the left operand's node which is either an *intNum*, *floatNum* or *id* node. When calling one of these nodes an instruction to store the actual value of that specific operand in the X register is appended to the *codeBuilder* string. If it is not a nested arithmetic operation node inside another arithmetic operation node, then the instruction *TXA* is appended to the *codeBuilder* string to transfer the value of the X register to the accumulator. This is done so that the right operand can be loaded into the X register without overriding the value from the left operand. The next if-statement ensures that the left operand is always called first, so the evaluation of the expression is done from left to right. The code of the arithmetic operation node is shown in listing 7.3.

Listing 7.3: Arithmetic operation node.

```
public void arithmeticOp(ArithmeticOp node) {
        node.getLeftOperand().accept(this);
        if (ArithmeticOpCount == 0) {
            codeBuilder.append("TXA\n");
            ArithmeticOpCount++;
        }

        if (node.getRightOperand() instanceof ArithmeticOp) {
            ((ArithmeticOp)
               node.getRightOperand()).getLeftOperand()
            .accept(this);
            addTwoNumbers(node);
            ((ArithmeticOp)
               node.getRightOperand()).getRightOperand()
            .accept(this);
            addTwoNumbers((ArithmeticOp)
                node.getRightOperand());
        } else {
            node.getRightOperand().accept(this);
            addTwoNumbers(node);
        }
    }
```

The function *addTwoNumbers()* simply looks at the operator of the current node and appends the relevant instructions to the *codeBuilder* string and is displayed in code example 7.4.

Listing 7.4: addTwoNumbers method.

```
public void addTwoNumbers(ArithmeticOp node) {
        codeBuilder.append("STX␣$0100\n");
        if (node.getOperator().equals("+")) {
            codeBuilder.append("ADC␣$0100\n");
        } else {
            codeBuilder.append("SBC␣$0100\n");
            codeBuilder.append("CLC\n");
            codeBuilder.append("ADC␣#1\n");
        }
    }
```

The *STX $0100* instruction is appended because the accumulator can not operate directly with the X register. Therefore the value from the X register is stored in the stack's last memory location so that the accumulator can use it.

Below is an example of how the expressions will end up being evaluated when calling the computing node:

$$2+3+4 \rightarrow ((2+3)_1+4)_2$$

The numbers in the parentheses show in which order the expression is evaluated.

If-else Node

The if-else node's purpose is to provide the source language with a selective control structure. If there is no "else"-part belonging to the if-statement, it will just be an if-node. Firstly, the if-else node calls its condition node, which will either be a Boolean, Id or comparison operation node. If it is a *Bool*, the value of the *Bool* node will be stored in the X register and therefore the if-else node will transfer it to the accumulator. The reason for this is that the accumulator is the only register on which the processor can do conditional operations. If the condition is a comparison operation node, the if-else node will call the method *pullAccumulator()* which appends the *PLA* instruction to the *codeBuilder* string and increments the *stackAddress*. This method is called because the comparison operation node will store its boolean value in the accumulator. The remaining piece of code makes the conditional jump to either the if or else block. On line 20, the node appends a label named "end" since the if-block must jump to the end label after it has been executed successfully. Otherwise, the else-block will also be executed, which is not the purpose. The code for the if-else node can be seen in example 7.5.

Listing 7.5: Code generation for If-else node.

```
1 public void visit(IfElse node) {
2     int label = labelCount;
3     labelCount++;
4     node.getCondition().accept(this);
5     comparisonOpCount = 0;
6     if (!(node.getCondition() instanceof Id)) {
7         if (node.getCondition() instanceof Bool) {
8           codeBuilder.append(InstructionSet.TXA.getInstruction()
               + "\n");
```

```
        } else {
          pullAccumulator();
        }
    }
    codeBuilder.append(InstructionSet.CMP.getInstruction() +
       "␣#1\n");
    codeBuilder.append(InstructionSet.BNE.getInstruction() +
       "␣else" + labelCount + "\n");
    codeBuilder.append("ifthen" + labelCount + ":\n");
    node.getThenBlock().accept(this);
    codeBuilder.append(InstructionSet.JMP.getInstruction() +
       "␣end" + labelCount + "\n");
    codeBuilder.append("else" + labelCount + ":\n");
    node.getElseBlock().accept(this);
    codeBuilder.append("end" + label + ":\n");
}
```

As seen in code 7.5, labels are appended to the codeBuilder string. These labels are appended as mentioned earlier to prevent the entire if and else part to be executed. Thus, the if statement ends up looking like this in Assembly 6502 code:

Listing 7.6: If-else structure in Assembly 6502

```
Condition is called

If false jump to else label

if label:
    code for if part of the if-else statement
Jump to end label
else label:
    code for else part of the if-else statement
end label:
```

Chapter 8

Quality Assurance

A definition for quality assurance, QA, is the following: *"Quality Assurance (QA) is a common practice to ensure that the end product of any Software Development Lifecycle (SDLC) conforms to the overall and scope-agreed expectations"* [41]. An iteration of a product is only finished once it is tested and evaluated. It is of course good common practice to perform technical testing as a part of the coding process, but a full evaluation is only complete once the whole system is tested. In this project, due to time constraints, whole system tests will only be written during the second iteration, but smaller parts of the system, as well as user tests, are included in the first iteration. If technical correctness or known bugs are documented, further work can be done knowing how the fundamental parts of the system should function.

The QA for this project is separated into two sections. The first part is user testing, which is used to evaluate the language design and the second part is testing the compiler. Unit tests and integration tests, which are discussed further in chapter 15, are created and implemented to ensure technical correctness, while think-aloud usability tests are used for the evaluation of the designed language.

8.1 Unit testing

Unit testing is the practice of testing smaller parts like individual methods of a program to ensure that each part works correctly. Unit testing is important as it verifies the correctness of the sub-parts of the program and supports the maintainability of the system. As changes are made and new features are added in the iterations, it is crucial to make sure that the code maintains its correctness. With the unit tests in place, the developer simply has to collectively run all of them to make sure that the product still works as it is intended to. For this project, the unit testing framework JUnit5 was used [8].

Unit tests were written in parallel with development. Therefore, the first tests written are focused on assignments, declarations, and ensuring that production rules of the grammar produce the output they should. These unit tests check the correctness of the parser and the generated AST.

An example of a unit test of the node *FloatDcl* is shown in 8.1. Firstly, an instance of the class *Compiler* is created, which value is a file path to a text file. This contains different float declarations written with the syntax of the designed language. The expected AST is then built programmatically by creating each of the nodes. After, the text file is parsed by the parser by using the Compiler's method *Prog()*, which returns the actual AST. Lastly, the test asserts if the actual AST and its nodes are equal to the expected AST. If there is an error during the parsing of the text file, the test will fail with the *assert false*.

Listing 8.1: Unit test for FloatDcl

```
@Test
void FloatDclTest() throws FileNotFoundException {
    // [GIVEN] That we run the parser with a file
    Compiler compiler =
        readFileToParse("Test/TestCode/floatDcl.txt");

    // [GIVEN] That we create the expected AST from the code
        from the file
    Prog expectedAST = new Prog();
    FloatDcl floatDcl = new FloatDcl("floatDcl");
    FloatDcl floatDcl2 = new FloatDcl("my_variable");
    FloatDcl floatDcl3 = new FloatDcl("order123");
    expectedAST.addChild(floatDcl);
    expectedAST.addChild(floatDcl2);
    expectedAST.addChild(floatDcl3);

    // [WHEN] We try to parse the code from the file
    try {
        Prog AST = (Prog) compiler.Prog();
        // [THEN] Assert that the created AST is equal to the
            expected AST
        assertTrue(AST.equals(expectedAST));
    } catch (Throwable e) {
        System.out.println("Syntax␣error:␣" + e.getMessage());
        assert false;
    }
}
```

Next, visitor patterns are tested. The visitor design pattern is used for type checking, scope checking, constructing the symbol table, and code generation. Since the compiler's overall correctness relies heavily on the correctness of the visitors, it is essential to check that these components function properly when they are tested as independent units.

The first test scenario evaluates the implementation of methods in the visitor interface. A test class is created that implements the visitor interface and contains all the required override methods. Additionally, the test class has a boolean property corresponding to each overriding method. These are initially set to false as the default value before each test. The tests contain one of the subclasses of the Node class. Within this subclass, the *accept* method of the superclass is used. This method takes the visitor interface as the parameter and calls the "visit" function from the node triggering execution of the associated method within the interface specific to that particular node. As the test enters the overridden "visit" method, the corresponding boolean property associated with the test is set to true. The test then verifies the expected result by checking whether the subclass of the node's boolean property is actually set to true. An example of the test for the visitor's function for the node *Prog* is shown in 8.2. This structured approach is repeated in subsequent tests allowing a comprehensive assessment of the visitor pattern.

Listing 8.2: Prog visitor test

```
public class VisitorTest implements Visitor {
    private boolean isProgVisited = false;

    @Test
    void VisitorProgTest() {
        Prog prog = new Prog();
        prog.accept(this);

        assertTrue(isProgVisited);
    }
}
```

8.2 Usability test

During usability testing, the focus lays primarily on the source language's design criteria 3. As preparation, a spreadsheet is set up containing fields for assignment description, target, and the actual time it takes to complete the assignments, issues, and their severity. During the test, a moderator is responsible for communicating with the participant and observing their behaviour - their observations are discussed and entered in the sheet subsequently 17. Another person is responsible for taking notes under the test and documenting the relevant feedback and comments. The test is set up as a casual lab test, since writing code is, in most cases, done in a calm environment without notable disturbances [43].

For evaluating the results of the tests, the technique known as Instant Data Analysis (IDA) was applied [28]. Using this method, a complete usability assessment can be carried out within a single day, saving a lot of resources. When conducting an IDA, a test monitor and observer articulate and discuss the most critical usability problems identified during the think-aloud sessions. They rate the severity of each problem and use notes taken during the think-aloud sessions to support the analysis. Problems are also grouped into themes to avoid redundancy.

When organising a usability test, it is crucial to consider that the number of novel usability issues uncovered by an observer tends to decline steeply after testing 5 or 6 participants [32], this can be seen in figure 8.1. As a result, the test was limited to five participants. The participants were software students from Aalborg University, with varying levels of knowledge about statically typed imperative languages. The test participants were selected since real users of the language would likely also be familiar with statically typed imperative languages, and have an education related to software or computer science. However, keep in mind that none of the participants were familiar with Assembly 6502, which, considering the target group of the project, was an unfulfilled prerequisite.

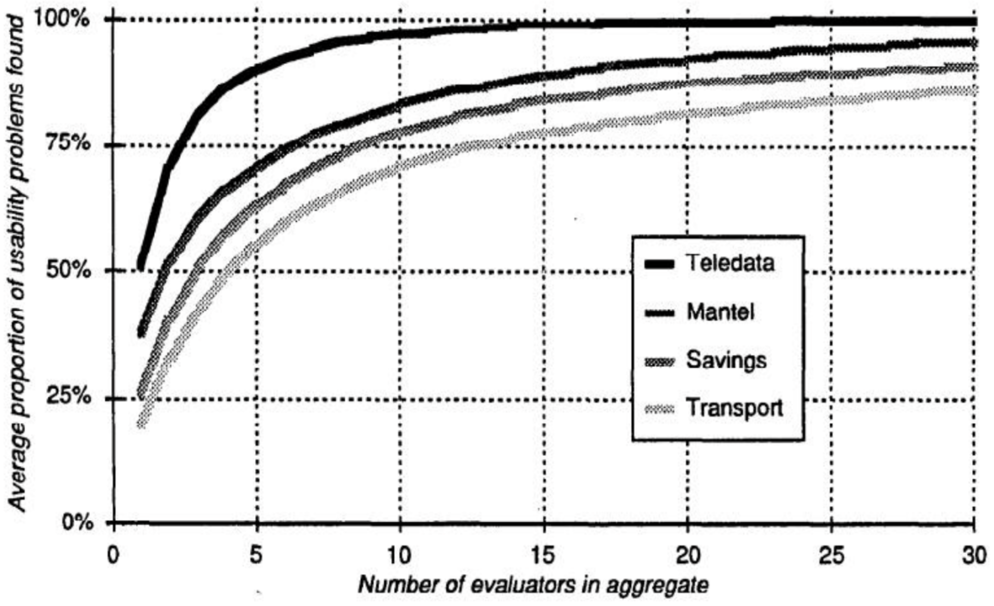
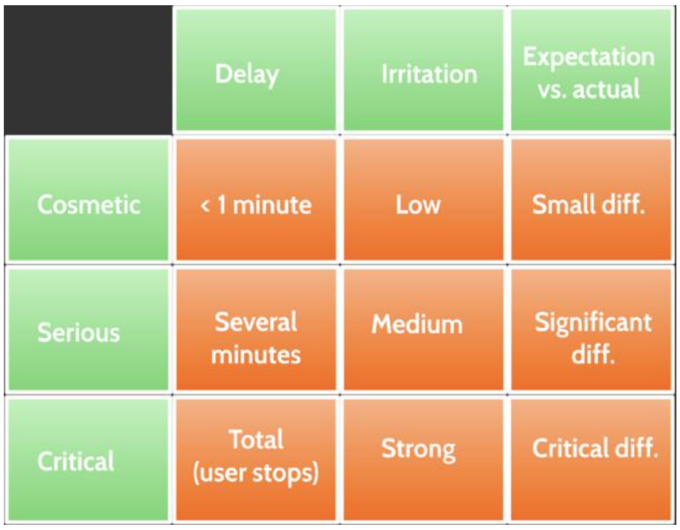


Figure 8.1: Average proportion of usability problems found compared to the number of evaluators in aggregate [26].

The problems discovered during testing are categorised in the following figure 8.2. A problem will be categorised as catastrophic if more than one participant experiences the same critical problem. [2].



	Delay	Irritation	Expectation vs. actual
Cosmetic	< 1 minute	Low	Small diff.
Serious	Several minutes	Medium	Significant diff.
Critical	Total (user stops)	Strong	Critical diff.

Figure 8.2: Problem categories.[2].

8.3 Discussion of the results

In this section, the answers and observations made during the usability tests are discussed. The assignments and follow-up questions of the individual test are meant to guide the user through the writing process of a simple program. During these assignments, specific parts of the language are tested, and the follow-up questions serve as additional clarification to the data gathered. This discussion covers the most relevant answers. For all answers, see appendix 17.

- Task 1: Try to explain what the following code does. You may use this code as inspiration in the following assignments.

The first question puts the readability of the program into focus. Based on the results, there is no significant difficulty in understanding any of the provided code, and participants are good at using only this code as inspiration for their own solutions. The provided code can be seen in listing 8.3 below:

Listing 8.3: Code provided to the test users in the first task

```
int a = 2;
boolean c = false;
if ((a == 2) && a <= 2 && !c) {
    c = true;
}
BYE;
```

- Task 2: Try to declare an integer, float, and two boolean variables. The names are up to you.

Although the provided code applies a large subset of the language, the participants must try and use their own knowledge and experience to find out how to declare a float type for instance and or-operation syntax is not explicitly shown either. During this question, some important discoveries were made. Firstly, some users expected implicit type coercion for float since it is a part of most modern languages (see float f = 1 instead of float f = 1.0). Type coercion is the automatic conversion of one data type to another. Furthermore, some participants would prefer feature multiplicity for the boolean type so it can also be written as bool and one participant would have preferred to declare multiple variables of the same types as a list, where its elements are separated with commas. Also for booleans, using 'true' or 'false' as possible values rather than numeric values was generally preferred. Although, some users would still like to have the option to use numeric values to represent boolean values.

- Task 3: Try to add 2 to the previously declared integer number and save it in a new variable.

The point of this question was to check how the user would combine a declaration with an arithmetic operation. There were no issues with this assignment.

- Task 4: Give the boolean variables a value and write an if-statement using an "and-operation" including the two boolean variables. In the body of the if-statement, add 2.5 to the float variable.

In this question, the writability of the language is tested specifically for the if-statement. Some users attempted to use a compound assignment for adding 2.5 to the float variable (i.e. f += 2.5 instead of f = f + 2.5). Therefore it was added as a priority in the second iteration.

- Task 5: Add a nested if-statement to the one previously written. Add an else statement to the new if-statement, which adds 3.14 to the float variable.

This question focuses on the users' use of curly braces and else-statements - a concept that was not included in the example either. Another aspect, which at some point even the test monitor forgot, was the BYE end of file notation. It was generally considered a useless gimmick that should be removed from the language or changed to a more conventional EOF notation. The assignment went smoothly; however, it did require slightly more time than expected. This is the only assignment, which did not match the expected time.

Relevant information gathered during the follow-up questions can be read in the following list.

- Readability received an average score of 8.2. The only negative point mentioned was the redundant BYE EOF notation. The participants compared the language to C#, C, and Rust.
- Writability received an average score of 8.4. The only unfavourable thing mentioned was the redundant BYE EOF notation with only one participant expressing appreciation comparing it to return 0. Also, compound assignments (+= or -=) and feature multiplicity (bool) could improve writability for some.
- Compared to the input language, the output of the compiler (Assembly 6502 code) was described as unintelligible, unreadable, and incomprehensible. The participants would not even attempt to write the same logic directly as Assembly code. In a more ideal scenario, the participants would

be people who are familiar with Assembly. Therefore, the feedback is skewed.

When taking the information gathered during testing into consideration, the language evaluation criteria 3 regarding simplicity, data types, and syntax design are strengths of the language.

Chapter 9

Discussion and evaluation of the first iteration

In this section, a discussion and evaluation of the first iteration of the language and compiler can be read. Firstly, some notable qualities of the language and compiler are discussed. Thereafter, known issues are described. At last, expectations for the second iteration of the language and compiler are put into a list form and presented.

9.1 Restrictions and implementation choices

Firstly, variables are stored on the stack during the first iteration. Because the first iteration of the language uses monolithic scopes, an easier storage method with the same effect could have been running through the symbol table and allocating the appropriate amount of space for each variable. Using the latter-mentioned method, accessing those variables would also be easier than interacting with the stack. The symbol table saves absolute addresses and in the generated code, absolute addresses are used for accessing variables. A more appropriate solution could be using base offset addressing instead where the base is the start of the stack and with each variable declared, the offset is incremented. Saving the offset also requires less memory than saving an entire address.

In future iterations, it is certainly worth considering a more advanced scoping structure, such as flat or nested scoping, since monolithic scoping has a lot of disadvantages in comparison to them. Monolithic scoping lacks any form of encapsulation, thus limiting modularity and abstraction in a program.

Secondly, arithmetic operations containing constants are evaluated during compile time. This is a common practice in compiler optimisation since it saves memory, and the size of the generated code is reduced. This was not implemented yet but was duly noted for the next iteration.

Due to limited memory, floats and integers have strict boundaries. Both types work with 8 bits. Integers use all of their bits to represent a natural number value, so an integer's value can range from 0 to 255. For future iterations, negative numbers could potentially also be introduced. For this, two's complements could be applied. Float numbers use fixed-point representation and use 5 of their bits for their natural number value and 3 for their decimal values, thus a float's value can range from 0-31.875. Float numbers have a precision of $\frac{1}{8}$ because of the hexadecimal system. In the future, floating point numbers could be added to give the user more freedom in choosing the precision of a float type. Using floating point, the users themselves could decide how bits are divided between natural and decimal numbers in a float-type variable.

Lastly, branching is only possible with -127 and +128 storage spaces away from the branching point. This could result in issues if a user for example writes an if-statement with a massive body. A possible solution to this could be to branch to a new label, which is right beneath the condition, and then that label would use the *JMP* instruction to jump to the if or else part.

9.2 Known issues

The first known issue in the program is that variables can be used in expressions before they are declared. This is because, in the first iteration, the compiler fills the symbol table first by checking all declarations and subsequently checks if the used variables exist. Rather than this, while filling up the symbol table, if any variable is used, its validity should also be checked, since it should be in the symbol table already if it was previously declared. A fix for this could be checking the symbol table while constructing the symbol table every time a variable usage is spotted.

Another known issue in the first iteration is regarding the storage of variables defined inside if-else statements. When a declaration is written in both the if and else part of an if-else statement, the stack address inside the compiler will decrement twice whereas in the compiled code it will only decrement once. This results in issues when trying to access a variable because the stack pointer is incorrectly placed. This is, however, not a major problem, because if nested scoping is implemented as planned, a new symbol table is created for each block accessed with a pointer to the symbol table for the parent scope. This means that when leaving the block, the stack pointer will be incremented the same number of times as there are new declarations inside that block and is, therefore, the same before and after entering the block.

9.3 Potential updates for the second iteration

This section contains a prioritised list of potential updates for the second iteration of the language:

1. Pointer type: It gives the programmer the possibility of more efficient use of available memory. This is especially relevant considering the MOS 6502's very limited writable memory. Directly accessing the memory based on addresses and altering data is very efficient in terms of performance [49].

2. While-loop: Iterative control structures are considered a fundamental part of modern programming. Without them, repetition in code can bear an unnecessary complexity.

3. Nested scope: Instead of the current monolithic scope, a nested scope would benefit the language. It would prevent naming conflicts, encourage encapsulation, and provide an overall more modern feel to the language.

4. BYE EOF-notation should be removed or replaced. During usability tests, it was deemed a redundant feature and unusual syntax.

5. Compound assignment (-=, +=): During usability testing, compound assignments were proven to improve the writability of the language. It is not a major concern since most users quickly found an alternative solution, but it is a highly requested feature.

6. Base-relative offset addressing: In the current symbol table, absolute addressing is used which means that the entire memory location is stored. Rather than doing this, base-relative offset addressing could be used. Using base-relative offset addressing can be a more memory-efficient alternative. Absolute addressing would not function with procedures either.

7. Procedures and/or functions with argument(s): Having procedures or functions in a language helps a lot with modularity and enables efficient reuse of code. Furthermore, they would provide a higher level of abstraction to the language, which is one of the focal points of the project. Arguably, it would also benefit the language's readability by eliminating a considerable amount of code repetition.

8. Multiplication and division: They are considered fundamental arithmetic operations, which are expected to be included in any language. However, the 6502 processor does not have instructions for multiplication and division, and is therefore difficult to implement, which results in low prioritisation.

9. Negative numbers: Generally, being able to work with negative numbers is important in programming. Additionally, doing potential workarounds as a programmer can be a tedious task.

10. Implicit type coercion for integer and float numbers: Implicit type coercion between float numbers and integer numbers is an expected feature, which was also shown during usability tests. Adding this feature could improve writability.

11. Explicit typecast for integer and float numbers: Adding this feature would give the programmer more flexibility and provide the types with improved compatibility.

12. For-loop: Another iterative control structure. It is ranked lower, as it can be replaced by a while-loop, which means that it is not a necessary addition, but nice to have.

During the first iteration, allocating place in writable memory for all variables in the symbol table as the first step in code generation was a possibility. Arguably, this solution would have been easier than working with a stack pointer and allocating storage space while traversing the AST. It was decided against, however, as this method only works for monolithic scoping, which is to be changed out for nested scoping during the next iteration. By putting more time and effort into implementing stack storage, taking a step towards more advanced scoping rules is made considerably easier.

Due to planning for a nested scoping language, the issue of declaring variables inside of if-else statements was not tackled in the first iteration. As of now, the code in listing 9.1 would result in a mistake in code generation.

Listing 9.1: Mistake in code generation for if-else statements

```
int a = 2;      /* stored at 01ff */
if (a == 2) {
    int b = 7; /* stored at 01fe */
} else {
    a = 4;
    int c = 8; /* stored at 01fd */
    int d = 9; /* stored at 01fc */
}

int e = 10;     /* stored at 01fd */
BYE;
```

The reason for the error is that the program automatically allocates storage space for all variables while traversing the tree at compile time. If the if and else part of the if-statement has different amounts of declarations, then any declarations afterwards can have two different offsets. The stack pointer/offset must be updated according to whether the program enters the else block.

Part II

Second Iteration

Chapter 10

Language Design II

In the second iteration, the language is updated with important new features, and small tweaks to already existing grammar. The updates are the following:

- While-loop added (fully implemented)
- Pointer type is added (fully implemented)
- Compound assignments added (fully implemented)
- Multiplication and division added (grammatically implemented, code generation missing)
- Procedures with optional arguments added (fully implemented)
- BYE eof-notation changed to END (fully implemented)

The grammar specification from the first iteration 3.5 is updated so it includes the new symbols, and also the rules for the new additions like while-loop and pointers.

Listing 10.1: EBNF of the language

```
Prog := Stmt* eof eol
Stmt := Assignment eol
```

```
      | IfStmt eol
      | Dcl
      | WhileStmt eol
      | Procedure eol
Assignment := id ( '+' | '-' ) '=' Expr
             | id '=' ( '&' id | Expr )
             | '#' id '=' Expr
Procedure := id '(' ( Val )? ')'
Dcl := Type  id (assign ( '&' id ) | Expr )? eol
     | ProcedureDcl
ProcedureDcl :=  proc id '(' ( type id )? ')' '{' Block '}'
Expr := OrTerm
OrOp := AndOp ( ( '||' ) AndOp )*
AndOp := EqualityOp ( ( '&&' ) EqualityOp )*
EqualityOp := ComparisonOp ( ( '==' | '!=' ) ComparisonOp)*
ComparisonOp := ArithmeticPlusMinusOp ( ( '<' | '>' | '<=' |
    '>=' ) ArithmeticPlusMinusOp )*
ArithmeticPlusMinusOp := ArithmeticMulDivOp ( ( '+' | '-' )
    ArithmeticMulDivOp )*
ArithmeticMulDivOp := NegationOp ( ( '*' | '/' ) NegationOp )*
NegationOp := ( '!' )? Factor
Factor:= '(' Expr ')'
        | Val
IfStmt := if '(' Expr ')' '{' Block '}' ( else '{' Block '}'
    )?
WhileStmt := while '(' Expr ')' '{' Block '}'
Block := Stmt*
Type := float
    | int
    | boolean
    | pointer
Val =: intNum
   | floatNum
```

```
    | true
    | false
    | ( '*' )? id
```

Regarding compound assignments, it must be noted, that to avoid having any side effects in the grammar, they were defined as statements. During implementation, however, this was overseen, and compound assignments were added as a form of an arithmetic expression. This was an oversight.

Chapter 11

Syntax analysis II

In the second iteration, lookahead was introduced to the parser and the logic of building the AST was changed.

Lookahead

In the first iteration, the parser used a single lookahead which made the parser into an LL(1) parser. In this iteration, a local lookahead was introduced multiple places in the production rules. When the parser is LL(k) in some points in the grammar, then the parser remains an LL(1) everywhere else for better performance. This is possible since shift-reduce and reduce-reduce conflicts are not an issue for top-down parsers [3].

The lookahead changed from one to two before and inside the production rule *Assignment* and before the *Procedure* rule, is seen at 11.1. Therefore, the parser was transformed into an LL(2) parser in these rules. The reason for using the two lookahead was the first set of the *Assignment* and *Prodecure* rules share the same id token. If the lookahead was one, the grammar would be ambiguous since the rules start with the id token so the deviation tree could be different.

Listing 11.1: Production rules with lookahead 2

```
Assignment := id ( '+' | '-' ) '=' Expr
            | id '=' ( '&' id | Expr )
            | '#' id '=' Expr
Procedure := id '(' ( Val )? ')'
```

Updated AST

The implementation of the grammar and the grammar itself was the underlying cause of the issue of the construction of the AST. Simply put, arithmetic operations were constructed with incorrect associativity instead of always left-associativity. The incorrect grammar for the arithmetic operation is shown below in example 11.2.

Listing 11.2: The incorrect grammar for arithmetic operation

```
ArithmeticOp = NegationOp ( ( '+' | '-' ) ArithmeticOp )?
```

The *ArithmeticOp* induces right-recursion by calling itself if a plus or minus sign is present in the input code. When reaching the second *ArithmeticOp*, the grammar finds the right side value for the AST to create a node of the expression. This would create a wrong order of operations. This is shown by using the input code *4-5+1* in example 11.3.

Listing 11.3: The order of operations

The right order with left-associativity:
$((4-5)_1+1)_2$
The order produced by the grammar:
$(4-(5+1)_1)_2$

The result of this could either be 0 or -2 depending on the order of operations. The faulty AST is shown in figure 11.1. This structure of building the expression hierarchy on the right side of the operations was a difficult task when traversing the AST top to bottom from left to right.

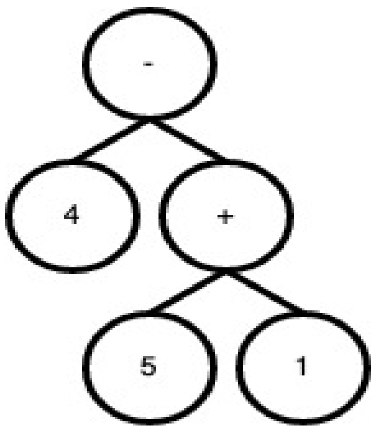


Figure 11.1: Right-associativity AST

The solution to this problem is rewriting the grammar and logic for creating multiple expression nodes in the AST. Instead of using recursion, the grammar uses the option to have zero or more expressions, which is seen in 11.4.

Listing 11.4: The correct grammar for arithmetic operation

```
ArithmeticOp = NegationOp ( ( '+' | '-' ) NegationOp )*
```

The first part of the solution is to set the first *NegationOp* to be the left child of the arithmetic operation. If the parser encounters a plus or minus sign, then a new arithmetic operation replaces the former left child and places it as the new arithmetic operation's left child. This results in correct left-associativity which can be seen in the AST in figure 11.2.

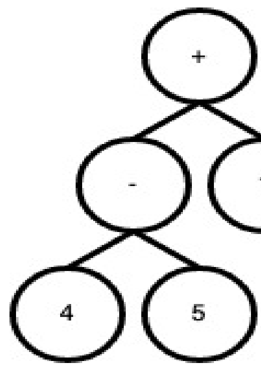


Figure 11.2: Left-associativity AST

Chapter 12

Semantic analysis II

As for the semantics of the language, the monolithic scoping of the first iteration was changed to nested scoping. This affected the semantics of the language. In this chapter, the changes in the semantics are documented.

As a result of implementing the nested scoping rules, the environment-store model is introduced to the semantics. This language has an environment for both the variables and procedures of the program and can be seen below.

$$E_v = Var \cup \{Next\} \rightharpoonup Loc,\ Sto = Loc \rightharpoonup \mathbb{Q}$$

The *Next* keyword in the variable environment set is referring to the next free memory location just like the stack pointer. The introduction of compound assignments means that the sets for arithmetic expressions and statements are updated accordingly. Pointers are also introduced with three different symbols to get the value, the address, and reassign the variable. The updated set S and T can be seen below:

$$T = \{int,\ float,\ boolean,\ pointer\}$$

$$S ::= \ T\ x = a \quad | \quad if\ b\ then\ S \quad | \quad if\ b\ then\ S_1\ else\ S_2 \quad | \quad while\ b\ do\ S \quad |$$

$$proc\ p(a)\,\{S\} \quad | \quad x \mathrel{+}= a \quad | \quad x \mathrel{-}= a \quad | \quad \#x$$

$$Pointer_{EXP} ::= \ \&x \quad | \quad *\,x$$

Below is the environment for procedures with the parameters shown:

$$E_p = Pnames \rightharpoonup Stmt \ \times \ Var$$

The procedure environment needs to include the product of statements and variables because the language uses static scoping. This means that it looks for the variables in the nearest closed scope and then recursively calls its parent scope until it reaches the variable or the global scope.
The following shows the big-step transition rules for procedure declarations:

$$[PROC_{BSS}] \quad \vdash \langle Proc\ P(Var\ x)\ S,\ e_v,\ Sto,\ e_p\rangle \rightarrow e_v,\ Sto,\ e_p[P \mapsto (S,\ x)]$$

Transition rules for procedure calls:

$$[Proc_{Call}] \quad \frac{e_v, Sto \vdash e \rightarrow v \quad \langle S,\ e_v[x \mapsto v],\ Sto,\ e_p\rangle \rightarrow e_v', Sto', e_p'}{\vdash \langle P(e),\ e_v,\ Sto,\ e_p\rangle \rightarrow e_v,\ Sto',\ e_p}, e_p(P) = S,\ x$$

Firstly, when a procedure is called the program enters a new scope. Secondly, the optional parameter expression is evaluated to a value. P is looked up in the procedure environment to check how the procedure is defined, so the call can continue. The parameter x is mapped to the actual value v, and afterwards, the statements of the procedure are executed. During the procedure call the variable environment, procedure environment, and store might all be changed. After the call of the procedure, the store will be changed as a consequence of the statements that have been executed. However, neither the variable environment nor the procedure environment is changed, as the scope for the procedure call is exited.

The semantic transition rules for the different pointer operations can be seen below:

$$[\textit{Pointer Get Value}_{EXP}] \quad \frac{e_v,\ Sto \vdash x \rightarrow v}{e_v,\ Sto \vdash *x \rightarrow Sto[v]}$$

$$[\textit{Pointer Change Value}_{EXP}] \quad \frac{e_v,\ Sto \vdash a \rightarrow v \quad e_v,\ Sto \vdash x \rightarrow l}{e_v \vdash \langle \#x \ = \ a, Sto\rangle \rightarrow Sto[l \rightarrow v]}$$

$$[\textit{Pointer Get Address}_{EXP}] \quad e_v \vdash \&x \rightarrow e_v[x]$$

Chapter 13

Compiler Optimisation

Compiler optimisation is a code transformation technique, whose goal is to improve the intermediate code by making it consume fewer resources. The compiler optimising process includes the following points [17]:

- Optimisation must be done correctly and must not change the semantics of the program in any way.
- Improving the program speed and performance.
- Compile time should be kept reasonable.

The optimisation of a compiler can be many different things such as common sub-expression elimination, dead code elimination and peephole optimisations which include constant folding [16].

When an expression or sub-expression appears in the code and the same sub-expression appears again later in the code, it is called a common sub-expression. Common sub-expression elimination is about reducing the common sub-expressions by eliminating them at compile time. It has the advantage of making the computation process faster by avoiding the re-computation of the common sub-expressions. In addition, it will utilise memory more efficiently since there is less intermediate code to translate to target code [18].

Dead code is part of the code that is never reached or used. Some examples of dead code are a declaration that is never used or a variable that is never used. Dead code elimination aims to remove the dead code situated in the code without changing the effect of the program. This reduces intermediate code which improves the compiling time [19].

The essence of peephole optimisation is to improve performance and reduce both memory footprint and code size [20]. One of the techniques for this is constant folding which reduces the AST and intermediate code by evaluating expressions with constants at compile time. The evaluated expression becomes a new constant and replaces the old operation in the AST.

Constant folding is the method used in this project to reduce the size of the target code in Assembly 6502 and the number of bytes used on multiple constant expressions. It is only introduced in the second iteration because optimisation is not a necessity for a working compiler. There is however an incentive to prioritise this technique of optimisation because of the target language. Assembly 6502, as mentioned, runs on older computers with a limited amount of memory. To be precise, there is 64KB 2 available compared to modern computers with up to 16GB of memory. Hence, constant folding is implemented to use the given bytes more effectively.

This is done by using a visitor pattern throughout the compiler. The visitor traverses the AST to find nodes with either an *Arithmetic* or *Comparison* operation. If the node's left and right child are the same type, the operation evaluates the value. Subsequently, the evaluated value replaces its parent node, and the process is repeated until all constants are evaluated and reduced. An example of the AST, which the line of code in 13.1 produces before and after constant folding is shown in figures 13.1 and 13.2. The full steps of the example can be found in the appendix 17.

Listing 13.1: Code example for constant folding.

```
    boolean a = 2 + (5 - 3) * 4 < 10 || true;
```

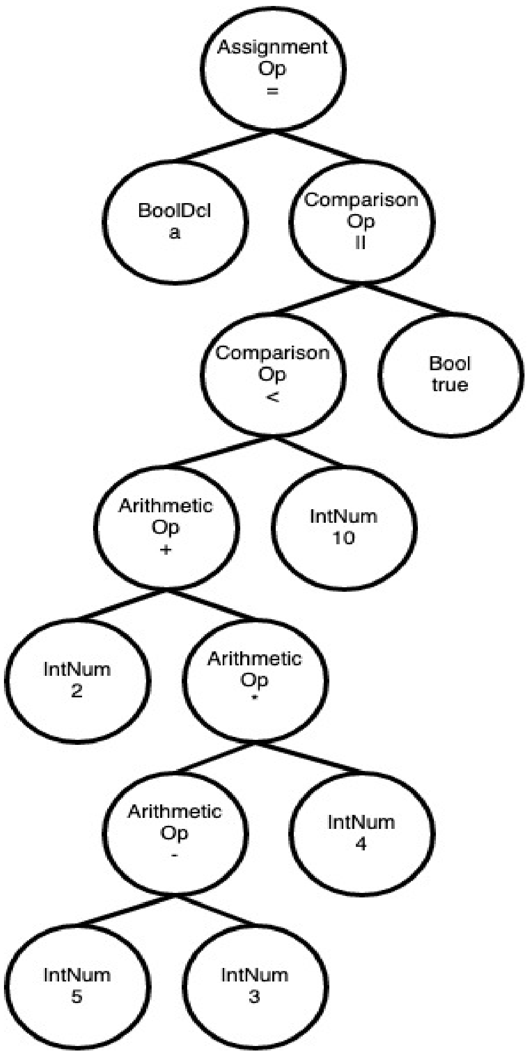


Figure 13.1: AST before constant folding

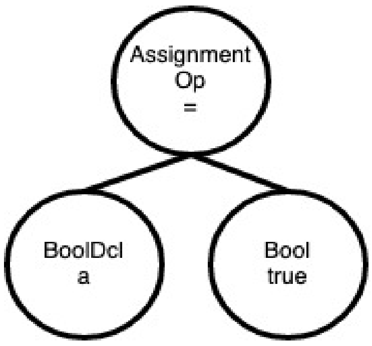


Figure 13.2: AST after constant folding

Chapter 14

Code Generation II

Addressing System

An important update to the code generation process was replacing the first iteration's absolute addressing system, which was used when storing and fetching variables, with a base-relative offset addressing system. This is important as the calling of procedures or functions recursively because recursive functions would not work with an absolute addressing system

The base-relative offset addressing system takes the stack-pointer as the base, which starts at memory address $01FF. The offset is the number of memory locations that are subtracted from the base.

To implement this new addressing system, the variable *stackAddress* from the first iteration was used again. With the new system, *stackAddress* starts from 0 and increments every time a new variable is declared. When the variable is declared, a method is called to set the variable's offset to be the same as *stackAddress* and subsequently increment the value of *stackAddress*. This ensures that the compiler knows where to find the variables based on their offset. An example of this is shown in listing 14.1 below:

Let us imagine that the variables *a* and *b* are both declared and in that specific order. Now the program needs to get the value of *a*, which has the offset 0 meaning it is located at address $01ff. The program needs *a* to be loaded into its accumulator and the instructions look like this:

Listing 14.1: Gets the value of a from the stack using the stack pointer.

```
    TSX
    INX
    INX
    LDA $0100, x
```

This program will transfer the stack pointer to the X register which would be 0xfd and then increment the value of the X register two times. Now the value in the X register is 0xff and the program can then load the accumulator with the address $0100 plus the value of the X register. When the *LDA* instruction has been executed, the value of *a* is in the accumulator.

Pointers

By implementing the pointer type in the language it was made possible to reach all the memory addresses in the 6502 processor. It does so by using indirect addressing mode, which is a method to point to an address and the address plus one. This opens up for accessing 16-bit addresses instead of just 8-bit. So instead of only having access to the zero-page, the pointers have access to the entire memory.
When the pointers get declared, they must either just be declared without a value or be declared with another variable's memory address. The pointers' actual values are relative to how many pointers were declared before. Thus, the value of the first pointer is 0, the second is 2, and the third is 4. The reason behind the two

increments is that they are 16-bit and not 8-bit. It is done in this fashion because pointers must point to the zero page that holds the actual memory address they are pointing to. So if the first pointer points to $0, then the memory it is actually pointing to is located in the addresses 0 and 1. This allows for restricted aliasing 3.1. An example of this is shown below:

Let us say we have a program which declares variable *a* and pointer *p*, which points to *a*'s address, then the program in the project's language looks like this in listing 14.2:

Listing 14.2: Code example for pointer.

```
int a = 0;
pointer p = &a;
#p = 12;
END;
```

The program above gets compiled to Assembly 6502 and can be seen in listing 14.3 below:

Listing 14.3: Code example for pointer in generated Assembly 6502.

```
LDX #0
TXA
PHA
TSX
INX
STX $00
LDX #1
STX $01
LDX #0
TXA
PHA
LDX #12
TXA
```

```
14      STA $0100
15      TSX
16      INX
17      TXA
18      TAY
19      LDX $0100, y
20      LDA $0100
21      STA ($00, x)
22      BRK
```

The program above was made with the project's compiler and shows how the variables get declared and pushed to the stack. The first three instructions show that *a* is set to 0 and pushed with that value to the stack. The instructions on lines 4-5 transfer the stack-pointer to the X register and increment it so that the X register's value is equal to *a*'s address. The X register's value is then stored at memory location $00 and memory location $01 is set to be 1. Now the X register is loaded with the value 0 because it is the first pointer and the value 0 is then pushed to the stack, meaning that the pointer *p* is now declared. Now the program loads the X register with the value 12 and transfers it to the accumulator. The accumulator's value (12) is then stored at the bottom of the stack $0100 for later use. The stack pointer is yet again transferred to the X register and incremented by one. The X register is then transferred to the Y register via the accumulator, and the X register is loaded with the instruction *LDX $0100, Y*. The Y register is in this case the value of *p*. Now the X register's value is 0, and the program loads the accumulator with the memory address $0100 which is 12. To store the value 12 in *a*'s memory address, the instruction *STA ($00, x)* is used which takes the value from (0 + x) and (1 + x) and connects them to create a 16-bit address. The 16-bit address at which the accumulator is stored is going to be $01ff, which is the location of *a*. In conclusion, the value 12 was stored in *a*, and it used the pointer *p* to do so. This specific example could omit the use of pointers, but it was used to show how the pointers in this project's language work.

While-loop

The while-loop was implemented to include an iterative control structure in the language. The code generation phase for the while-loop node looks quite similar to the if-statement node, which made it relatively easy to implement. The only difference is that the while-loop must jump back to its condition every time its block has been executed. This can be seen in listing 14.4 below:

Listing 14.4: Example for while loop.

```
whileLoop:
  // condition
  // if false jump to endOfWhileLoop
  // block
  JMP condition
endOfWhileLoop:
```

The use of labels for the while-loop made the implementation much easier, but it carries the same branching error as the if-statement. If the block of the while-loop is too large, the 6502 processor can not branch to the *endOfWhileLoop* and the code can not compile.
After the implementation of nested scoping, the block inside the while-loop has its own scope but can still interact with any parent scopes. A code example of how a while-loop could look in the language of the project is shown below 14.5:

Listing 14.5: Code example of while-loop.

```
int a = 3;
int b = 0;
while (a > 0) {
    a -= 1;
    b += 1;
}
```

Procedures

As mentioned earlier in the discussion of the first iteration 9, procedures were implemented to enhance the modularity of the language. It was done so by creating labels in the 6502 code and then putting all the instructions for the different nodes inside that label. The only problem that occurred was that when jumping to a sub-routine on the 6502 processor, it incremented the stack-pointer by two. This led to a bug in the node but was fixed by incrementing and decrementing the *scopeLevel* and *blockCount* attributes. This small fix was done so that the rest of the class would not need to be changed. The code for the procedure declaration node can be seen below 14.6:

Listing 14.6: Code generation for procedure declaration.

```
@Override
    public void visit(ProcedureDcl node) {
        codeBuilder.append(node.getId() + ":\n");
        if (node.getLeft() != null) {
            scopeLevel++;
            blockCount++;
            node.getLeft().accept(this);
            scopeLevel--;
            blockCount--;
        }
        node.getRight().accept(this);
        if (node.getLeft() != null) {
            pullAccumulator();
        }
        codeBuilder.append(InstructionSet.RTS.getInstruction()
              + "\n");
16     }
```

As seen on line 3, the first thing appended to the *codeBuilder* string is the id of the procedure followed by *:*. The if-statement on line 4 checks if there are any parameters to include and the increments and decrements mentioned just before ensure that the parameter passed to the procedure is inside the procedure's scope. After the if-statement, the right node, which is the block node, is called to append the 6502 instructions corresponding to the code inside it. The next if-statement on line 11 pulls the optional parameter off the stack once the procedure is done executing. Lastly, the instruction *RTS* is appended which is the instruction to return from sub-routines.

The actual procedure node which is the one that is called when the actual procedure is called and not declared is quite small for the code generation phase. See listing 14.7:

Listing 14.7: Code generation for procedure call.

```
@Override
    public void visit(Procedure node) {
        if (node.getRight() != null) {
            node.getRight().accept(this);
            codeBuilder.append(InstructionSet.TXA.getInstruction()
                + "\n");
        }
        codeBuilder.append(InstructionSet.JSR.getInstruction()
            + "␣" + node.getId() + "\n");
    }
```

Just like the *procedureDcl* node, it checks if the procedure has a parameter and if so, it then loads it to the accumulator. The accumulator is then pushed to the stack before executing the block of the procedure being called, but after jumping to the sub-routine. The jump to subroutine instruction *JSR* is appended after the accumulator is loaded with the value of the parameter. The procedures can be called with either an *IntNum*, *FloatNum*, *Bool*, or an *Id* node which includes a declared pointer.

Chapter 15

Quality Assurance II

This chapter documents the implementation of integration and acceptance testing, and some examples of the programs used for acceptance testing are described. It is important to note that the generated code's correctness is checked, but its cleanliness and efficiency were not considered important.

15.1 Integration and acceptance testing

Integration testing is a way of testing the interaction and data communication between software modules by testing them as a group after integrating them. Even though these software modules undergo unit testing, there remains a possibility of undiscovered defects that can be rectified through integration testing. There are different types of integration testing; the ones researched during this project are the Big Bang Approach and Incremental Approach. The latter is further divided into Top Down Approach, Bottom Up Approach, Sandwich Approach and Functional Approach [23].

A brief overview of how the different approaches differ (everything on this list is directly cited from the sources given):

- Big Bang: All units are linked at once, resulting in a complete system test [37].
- Incremental: Modules of the program are integrated one by one using stubs or drivers to uncover the defects [38].
  - Top Down: Takes place from top to bottom. Unavailable components or systems are substituted by stubs [38].
  - Bottom Up: Takes place from bottom to top. Unavailable components or systems are substituted by drivers [38].
  - Sandwich approach: A combination of the Top Down and Bottom Up approaches [39].
  - Functional: Takes place based on the functions or functionalities as per the functional specification document [38]. A functional specification is a formal document used to describe a product's intended capabilities, appearance, and interactions with users in detail for software developers [42].

An acceptance test is generally understood as determining the degree to which an application meets the end users' approval [21]. However, in this case, the product was not developed for a specific client, therefore acceptance testing is interpreted differently. In this project, to reach a valid level of *acceptance*, the compiler must be able to handle a batch of programs written by the development team. These programs are designed to represent realistic use cases of the language and collectively include everything the language has to offer.

Largely due to time constraints and ease of implementation, Big Bang testing is used, which means that the entire system is tested with a batch of programs. Thus, integration and acceptance testing essentially serve the same purpose. Some of these programs are written during development to test specific functionalities and some of them are written after the completion of the second iteration to test a larger part of the language in a wider logical context; see for instance the Fibonacci sequence program in 15.1.

Listing 15.1: Program for finding the 10 first numbers of the Fibonacci sequence.

```
int a = 0;
int b = 1;
int n = 11;
int i = 2;
while (i < n) {
    int c = a + b;
    a = b;
    b = c;
    i = i + 1;
}
END;
```

The code in listing 15.1, was compiled into Assembly 6502 and executed by the emulator used for this project. The output showed that the program was done correctly meaning that the first 10 numbers in the Fibonacci sequence were calculated correctly.

Another integration test was a program that uses pointers to draw a cross on the screen of the emulator. The code program looks like the code in listing 15.2 below:

Listing 15.2: Program for drawing a cross on the emulator screen.

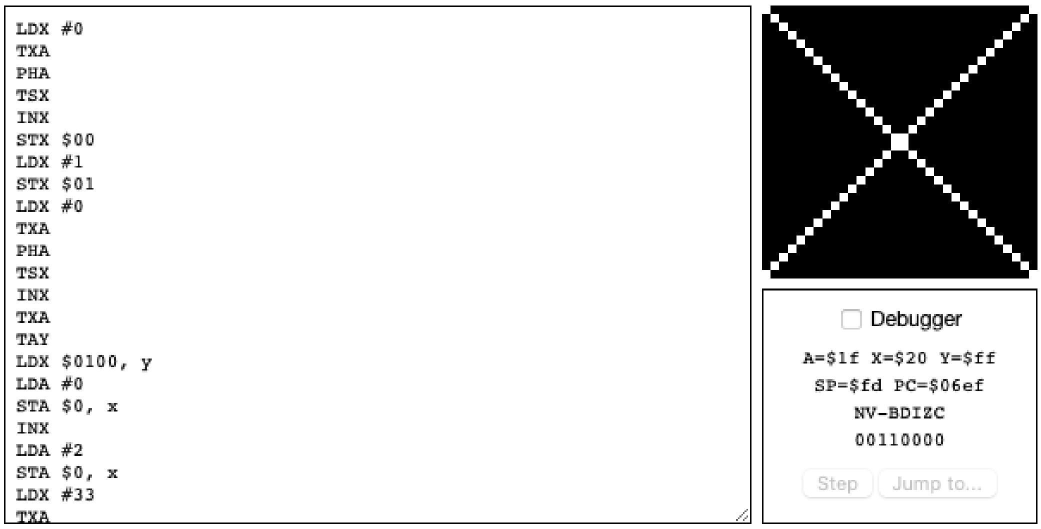
```
int a = 0;
pointer p = &a;

proc proc1(int b) {
    while (a < 32) {
        #p = 1;
        a += 1;
        p += b;
    }
}

p = 510;
proc1(33);
a = 0;
p = 510 + 31;
proc1(31);

END;
```

The code in listing 15.2 above was compiled into Assembly 6502 and executed in the emulator like the Fibonacci sequence. The output of the program is shown in figure 15.1 below:



```
LDX #0
TXA
PHA
TSX
INX
STX $00
LDX #1
STX $01
LDX #0
TXA
PHA
TSX
INX
TXA
TAY
LDX $0100, y
LDA #0
STA $0, x
INX
LDA #2
STA $0, x
LDX #33
TXA
```

```
A=$1f X=$20 Y=$ff
SP=$fd PC=$06ef
NV-BDIZC
00110000
```

Figure 15.1: Writing a cross on the screen in the emulator

The two integration tests mentioned above were not implemented in the testing environment. The reason behind this is that the code for running these two programs in Assembly 6502 is more than 100 lines of code.

The only test example which was implemented was a program declaring three different variables. It was tested by comparing the actual *codeBuilder* string created by the compiler, with a manually created string. To test larger programs, like the Fibonacci sequence, the manually created string would take a very long time to create, and be error-prone. The example program can be seen in listing 15.3 below:

Listing 15.3: Code example showing 3 int declarations.

```
int phone_number;
int creditCardNumber;
int a123456;
END;
```

The program above is expected to output the following Assembly 6502 instructions:

Listing 15.4: Code example showing 3 int declerations compiled into Assembly 6502.

```
PHA
PHA
PHA
JMP Final
Final:
```

The instructions above show that three variables are pushed to the stack with the *PHA* instruction. The *JMP* instruction with the label *Final* is appended to the string in case the program includes procedure declarations. The *JMP* instruction makes sure that the procedures do not execute if they are not called in the program, meaning that the procedures code will be put in between the *JMP Final* and *Final:* instruction.

Chapter 16

Discussion

In this section, potential future improvements of the language and the compiler, and the degree to which requirements and expectations were fulfilled are discussed. Additionally, the groups' reflections on the project are documented highlighting the mistakes that were made and certain aspects which should have been done differently. The following sections were written subsequently to the completion of the second iteration of the project.

16.1 Future Work

Negative values

Negative numbers are supported by 6502, and it is done through the two's complement representation. This would allow for the representation of a wider range of values, as it would not be limited to positive values anymore. The following example (16.1) demonstrates how negative values are created in 6502, as it is not directly supported and requires a combination of instructions.

Listing 16.1: Assembly 6502 instructions for negative numbers

```
    LDA #$05     ;
    EOR #$FF     ;
    CLC          ;
    ADC #1       ;
    SEC          ;
```

The first instruction loads the positive value 5 into the accumulator using immediate addressing mode. The second instruction inverts all the bits of the value in the accumulator effectively performing the one's complement operation. The third instruction clears the carry flag in the processor status register. The fourth instruction increments the value by 1 in the accumulator completing the two's complement operation. The final instruction sets the carry flag in the processor status register to indicate that the value in the accumulator is negative. After these instructions, the accumulator will contain the two's complement representation of -5 and the negative flag will be set to indicate a negative value.

Code generation for multiplication and division

Although multiplication and division were both implemented in the CFG of the language, as well as in pretty printing, code generation was not done for these arithmetic operations. There are no instructions for these operations in Assembly 6502 which makes their implementation more difficult and time-consuming. A possible solution for multiplication could be to use a while-loop to add a number x amount of times.

For-loop

The potential inclusion of the for-loop was also mentioned in the evaluation of the first iteration 3.1. Although it would be a welcome addition, it can easily be replaced with the already implemented while-loop. Implementation of the for-loop would, presumably, not propose any massive challenges.

Functions

Another step towards modularity, other than the already implemented procedures, could be the implementation of functions, grouping a set of instructions, and also returning a value. The main challenge of implementing functions would be returning a value. The Assembly 6502 language does not support either functions nor returning a value from a sub-routine.

Low-Level Virtual Machine Intermediate Representation

LLVM IR is a low-level, platform-independent, Assembly-like language that is used as a target language for various source languages.

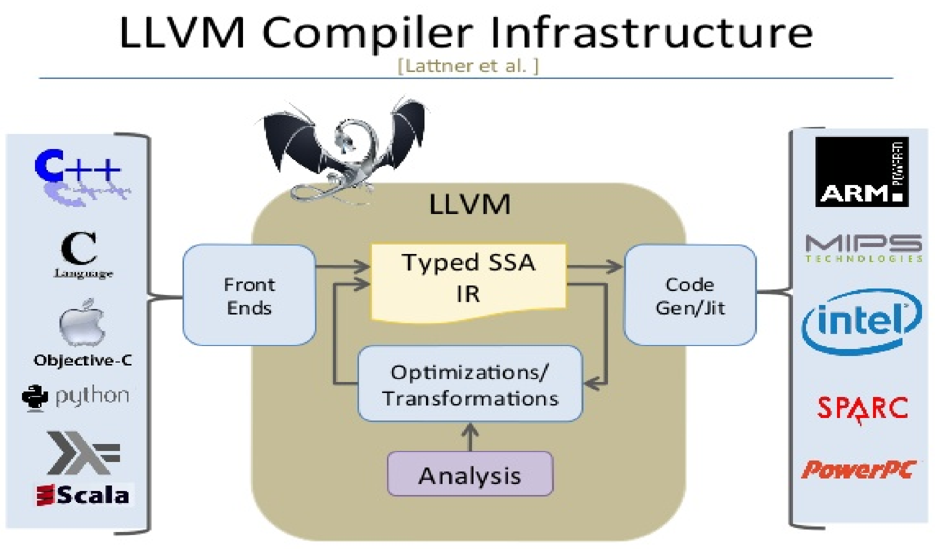


Figure 16.1: The LLVM compiler infrastructure[13].

At the centre of figure 16.1 is the intermediate code representation. If a language can be compiled into LLVM IR format, analysis tools based on this IR can easily be reused.

Clang uses the LLVM compiler as its back-end and it has been included in the release of the LLVM since the LLVM 2.6 [25]. Unfortunately, there is no official release of LLVM with Assembly 6502 as the target language. This means that if the group was to translate their own language to LLVM IR, the group would also have to write another phase of code generation from LLVM code to Assembly 6502. This idea was discarded since learning a whole new language, in exchange for potential code optimisation and translation from other languages that compile to LLVM IR, seemed like a task that is out of the scope of this project.

Implicit type coercion

Implicit type coercion was a feature that users during the usability test 8.1 tried to take advantage of, but failed to because it is not a feature in the language. The Assembly 6502 language does not have built-in implicit type coercion like in higher-level programming languages and it instead treats all data as binary values. Therefore, a solution to this would lie in compile time. The conversion of floats to integers could be solved by rounding the value up or down, which could be handled by the compiler. This was deemed nice to have and was therefore not implemented due to time constraints.

Explicit typecast

Another feature desired by the participants of the usability test 8.1 was explicit type casting. This is similar to the feature in the previous section, however, in this case, the data conversion is explicitly specified. In modern programming languages, like Java, for example, this is performed by using conversion functions or operators [34]. A solution would still lie in compile time as it is determined by the interpretation within the conversion to 6502. As with implicit type coercion, this was deemed nice to have.

Operations with side effects

Currently, there are no operations with side effects included in the grammar. To take a concrete example, incrementation, *x*++, is considered an impure operation since it produces a side effect. This is because the expression *x*++ returns the value of x, but following the return also adds 1 to *x*'s value. As a result of the evaluation, the memory is also changed. This logic also applies to decrementation *x*--. They were left out to keep the language clean, meaning that there are no expressions with side effects. Having no side effects also helps avoid short-circuiting. *Short-circuiting is a programming concept in which the compiler skips the execution or evaluation of some sub-expressions in a logical expression.* [30].

The code in listing 16.2 is an example of short-circuiting due to operations with side effects:

Listing 16.2: Code example showing short circuting

```
int x = 0;
if (++x > 0 || x++ > 0) {
    // do something
}
```

In this code, the ++*x* operation increments *x* by 1 and returns the new value, which is 1. Since 1 is greater than 0, the left-hand side operand of the `||` operator is true. Because of short-circuiting, the right-hand side operand *x*++ > *0* is not evaluated at all, so *x* remains at its current value of 1. It was not implemented as it is only used to optimise the compiled code and does not add any extra functionality.

Bootstrapping

Bootstrapping is a technique used to craft a self-hosting compiler, which is a compiler that can compile its own source code [27]. For further information, see the appendix 17. Bootstrapping was researched, but not considered as a viable option during this project, because of the small storage size of the MOS 6502 processor.

16.2 Discussion of language criteria and requirements

The purpose of this section is to evaluate the language that has been created for this project and, in doing so, a verdict can be given on the readability, writability and how reliable the language is.

This is done by reflecting on the language evaluation criteria 3.1. Overall simplicity is present in the custom language, as feature multiplicity was avoided to increase readability and writability. The inclusion of some features was based on the usability test 8.1 and by observing what features the participants used or attempted to use. Operator overloading was also a concern and was avoided with the only exception being that the * symbol is used for multiplication and reading the value from a pointer location. Beyond this, the expressivity of the language was enhanced by observing the users' behaviour when using the language. It would be ideal to conduct another usability test to see if the new features improved the language.

The syntax of the language mostly follows the traditional naming lexemes, which was recommended by Fischer [16]. The reserved keywords are case-sensitive and therefore still provide flexibility for the user to use them for unintended purposes if they wish to.

Type checking is done at compile time, which fulfils the requirement created earlier in the project 3. This increases reliability, both when using the language and when running the program on the intended platform as it does not use the already scarce amount of memory.

Exception handling is a characteristic that was not a priority, and the reliability of the language suffers greatly because of its absence. The effects of this do not only hurt the user but the developers as well since the error messages are either non-existent or vague. For this reason, it takes a lot of time and effort to find the bugs in the code.

The characteristics included in this language might not be the best example of how they should be implemented, but they improve the readability, writability, and reliability of the 6502 language and that was the goal to begin with 1.

16.3 Reflections on the project

Reflecting on how this particular project was conducted is a major source of learning for the whole development group. Mistakes and questionable choices, that were unclear during the process, are a lot clearer in hindsight. Seeing these aspects of the project as a part of the big picture highlights their importance.

Configuration management

The first item that is reflected upon is the software configuration management of the project. *Configuration Management (CM) is a technique of identifying, organising,* *and controlling modification to software being built by a programming team* [36]. The version control system used in this project is Git. This version control system is simply implemented by initialising a repository for the project and adding the configuration data files to it [10]. The project uses a feature-branch structure in which a main branch contains a presumably stable build while developers integrate new features in separate development branches and merge them into the master branch once they are stable. Normally, there are three layers in a feature branch system; master branch, development branch, and feature branch. In this project, the master and development branches are the same. This is because there is no active "release" of the product.

An aspect which the group did not learn in time to apply in the project is that Git could have been used for much more than version control. Using GitHub actions, continuous integration (CI) could have been implemented automatically running mandatory checks before merging and thus securing a stable release. These checks would ensure that all tests are run before merging and that no files are missing, etc. The essence of CI is that as several members of a team work on code on different Git branches, the code can be merged into a single working branch which is then built and tested using automated workflows such as mandatory tests [15]. Continuous deployment (CD) is a practice that is often combined with CI, which takes a step further and takes automation to the deployment level, automating the deployment of a project to an environment [15].

During the project, a much more ad-hoc, unstructured integration approach was used. Since integration tests were not implemented early on, previous stable versions were compared to the new version of the application for overall correctness. Version control could have been easier if GitHub actions were implemented, and a correct form of CI was applied.

Choice of a parser generator

A part of the project was the choice of a parser generator, which could have been easier and would have taken less time if the team had previous experience with compilers. Since a parser is an integral part of a compiler, the team was careful with selecting one and modularity was initially not considered as a deciding factor. Fortunately, as long as the same AST is generated, which is done using action code added to the grammar specifications, switching out a parser generator should be a relatively easy task. LL and LR parsers have different strengths and weaknesses, 5. Since the generated parser completes the same goal and is not used subsequently in the compiler using ANTLR or Yacc would have been fine choices too. In ANTLR specifically, the action code and grammar could have been virtually the same. Luckily, the implementation of the CFG and AST generation was done in a modular manner without depending on the parsing technology of choice. In short, parsing technology could be switched from JavaCC to another option without any major changes to the code.

The only issue discovered with the parser technology of choice, which was an LL(1) top-down recursive descent parser, was that it cannot handle left recursion. This was subsequently discovered when working with expressions/operations where left associativity must be done to correctly evaluate specific mathematical expressions. Although this mistake was fixed during the second iteration 14, not noticing this characteristic earlier was a mistake that cost a lot of development time.

The timeline of the project

The Gantt diagrams introduced in section 1 were used to plan (aided by backcasting) and also to track the progression of the project. The overall idea of following university courses during the first iteration worked well, and the result of the project was a language with a lot of tools, that could be used to write programs for the MOS 6502. The only clear mistake made was the short time allocation for writing the code generation part of the compiler, which also resulted in some partially implemented features; namely multiplication and division. Comparing the expected Gantt chart from appendix 17 and the actual Gantt chart from appendix 17, the planned and actual time used for code generation is vastly different for both the first and second iterations of the compiler. During the second iteration, the actual time spent on implementing code generation for new features more than doubled compared to the planned amount. This mistake was made because the planned timetable was not updated after the team familiarised themselves with the limitations of the target language. As for the final touches to the paper, the prolonged final adjustments were not explicitly planned, but were expected, and did not put the team under a lot of pressure. Overall, the planning of the project worked out fine but should have been adjusted accordingly to the discoveries of new challenges related to code generation.

Testing

This project could have greatly benefited from a larger test coverage. More extensive unit testing could have ensured that every separate part of the compiler is functioning correctly, meaning that further development can be done knowing that previously tested parts are ensured to function as intended. For example, this could have been utilised with a Test Driven Development (TDD) approach. The reason for not implementing more tests throughout the project was that compiler crafting was new to all the group members. This resulted in a lot of trial-and-error attempts of trying to make components work. Therefore, the TDD approach was not used because the group members did not know what the test should be able to do beforehand.

Specifically, unit tests for scope checking, type checking and code generation were not implemented. Although, the visitor interface itself is tested (used in scope and type checking & code generation) and during integration testing, the whole system is checked, so in a way, every component is tested, but certainly not thoroughly enough. Generally, even if the TDD approach was not implemented, tests should have been written in parallel with developing new functionalities.

16.3.1 Problem definition

In the early stages of the project, it is a good practice to write a concise problem definition that specifies the problem at hand and narrows the scope of the project down. Having a clear-cut problem definition could have been beneficial for this project since it could have provided a more coherent understanding of the requirements and evaluation criteria. The research phase also becomes more directed, since there is a clear goal in mind as to what kind of research is needed and what it should be used for. Without a clear and concise problem definition, it can be difficult to determine how much and what knowledge is needed to document the project process and theory satisfactorily.

Chapter 17

Conclusion

The language and the corresponding compiler developed in this project proposes to address the problems of the Assembly 6502 language. As described in the introduction section 1, writing code in Assembly 6502 is a time-consuming process with a steep learning curve requiring technical knowledge of the hardware. Furthermore, the resulting code is challenging to read, debug, and maintain.

A large part of the development process was language design 3, where the group had to take the limitations of the MOS 6502 microprocessor into consideration while developing a language that improves upon the aforementioned issues of Assembly 6502. After two iterations, this process resulted in a language using static scoping, static explicit typing including crucial control flow statements (if, while), and supporting modularity through procedures. The available data types in the language are integer, boolean, fixed-point float, and pointer.

The compiler of this project is implemented in Java and takes the project's own language as source and outputs Assembly 6502. The correctness of the compiler is tested using unit tests and Big-Bang integration tests taking a batch of realistic example programs applying all aspects of the language.

The main issues with this project, as previously discussed 16, were that the problem field could have been further researched and documented, testing and configuration management could have been more structured, and time allocation for code generation could have been corrected after the first iteration. Despite these missteps, it could be argued that the resulting language and compiler, based on the acceptance tests, is a viable alternative to writing Assembly 6502, providing the user with improved modularity, abstraction, maintainability, readability, writability, and expressivity.

Bibliography

[1] Open AI. *All things about ChatGPT*. URL: https://help.openai.com/en/collections/3742473-chatgpt. (accessed: 18.05.2023).

[2] Lars Mathiassen et al. *Object-oriented analysis & design*. 2 ed. Metodica, 2018. ISBN: 9788797069301.

[3] Francis Andre. *JavaCC*. 2022. URL: https://javacc.github.io/javacc/. (accessed: 16.04.2023).

[4] Mark Andrews. *A Guide To Atari Assembly Language*. 1. ed. Motorola Semiconductor Products Inc., 1984. ISBN: 0881901717. URL: https://www.atariarchives.org/roots/chapter_3.php.

[5] ANTLR. *Parse Tree Listeners*. URL: https://github.com/antlr/antlr4/blob/master/doc/listeners.md. (accessed: 09.05.2023).

[6] Abhishek Attrie. *ANTLR4 grammar syntax support*. URL: https://stackoverflow.com/questions/49660586/is-there-any-alternate-way-to-preview-the-antlr-generated-parse-tree. (accessed: 09.05.2023).

[7] Baeldung. *Statically Typed Vs Dynamically Typed Languages*. URL: https://www.baeldung.com/cs/statically-vs-dynamically-typed-languages. (accessed: 02.05.2023).

[8] Stefan Bechtold. *JUnit 5 User Guide*. URL: `https://junit.org/junit5/docs/current/user-guide/`. (accessed: 18.05.2023).

[9] Andrew Blance. *How do Processors Actually Work?* 2020. URL: `https://codeburst.io/how-do-processors-actually-work-91dce24fbb44`. (accessed: 09.03.2023).

[10] Ian Buchanan. *Configuration management*. URL: `https://www.atlassian.com/microservices/microservices-architecture/configuration-management`. (accessed: 13.05.2023).

[11] The Western Design Center. *65xx - Embedded Intelligence Technology*. URL: `https://www.westerndesigncenter.com/`. (accessed: 26.04.2023).

[12] Mike Dour. *Infragistics Parsing Framework - LL vs. LR Parsing*. 2012. URL: `https://www.infragistics.com/community/blogs/b/mike_dour/posts/infragistics-parsing-framework-ll-vs-lr-parsing`. (accessed: 18.04.2023).

[13] Lattner Et.Al. *The LLVM Compiler Infrastructure*. URL: `https://dl.acm.org/doi/10.5555/977395.977673`. (accessed: 08.05.2023).

[14] Ortin et.al. *An empirical evaluation of Lex/Yacc and ANTLR parser generation tools*. URL: `https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0264326`. (accessed: 09.05.2023).

[15] Colby Fayock. *What are Github Actions and How Can You Automate Tests and Slack Notifications?* URL: `https://www.freecodecamp.org/news/what-are-github-actions-and-how-can-you-automate-tests-and-slack-notifications/#what-is-ci-cd`. (accessed: 13.05.2023).

[16] Charles N Fischer, Ron K Cytron, and Richard J LeBlanc Jr. *Crafting A compiler*. en. Crafting a compiler with C. Pearson, Oct. 2009. ISBN: 9780136067054.

[17] GeeksforGeeks. *Code Optimization in Compiler Design*. 2023. URL: `https://www.geeksforgeeks.org/code-optimization-in-compiler-design/`. (accessed: 21.04.2023).

[18] GeeksforGeeks. *Common Subexpression Elimination*. 2023. URL: `https://www.geeksforgeeks.org/common-subexpression-elimination-code-optimization-technique-in-compiler-design/`. (accessed: 21.04.2023).

[19] GeeksforGeeks. *Machine Independent Code optimization in Compiler Design*. 2023. URL: `https://www.geeksforgeeks.org/machine-independent-code-optimization-in-compiler-design/`. (accessed: 21.04.2023).

[20] GeeksforGeeks. *Peephole Optimization in Compiler Design*. 2023. URL: `https://www.geeksforgeeks.org/peephole-optimization-in-compiler-design/`. (accessed: 21.04.2023).

[21] Alexander S. Gillis. *Acceptance Testing*. URL: `https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test`. (accessed: 10.05.2023).

[22] GitHub. *Your AI pair programmer*. URL: `https://github.com/features/copilot`. (accessed: 18.05.2023).

[23] Thomas Hamilton. *Integration Testing: What is, Types with Example*. URL: `https://www.guru99.com/integration-testing.html`. (accessed: 09.05.2023).

[24] Hans Huttel. *Transitions and trees*. Cambridge, England: Cambridge University Press, Apr. 2010.

[25] Incredibuild. *What is Clang?* URL: `https://www.incredibuild.com/integrations/clang`. (accessed: 08.05.2023).

[26] Rolf Molich Jakob Nielsen. *Heuristic Evaluation of User Interfaces*. 1990.

[27] JavaPoint. *Bootstrapping*. URL: `https://www.javatpoint.com/bootstrapping`. (accessed: 08.05.2023).

[28] Jan Stage Jesper Kjeldskov Mikael B. Skov. *Instant Data Analysis: Conducting Usability Evaluations in a Day*. 2004.

[29] Mike Lischke. *ANTLR4 grammar syntax support*. URL: `https://marketplace.visualstudio.com/items?itemName=mike-lischke.vscode-antlr4`. (accessed: 09.05.2023).

[30] Fraza M. *Short-circuit evaluation in Programming*. URL: `https://www.geeksforgeeks.org/short-circuit-evaluation-in-programming/`. (accessed: 13.05.2023).

[31] Nick Morgan. *Easy 6502*. URL: `https://skilldrick.github.io/easy6502/simulator.html`. (accessed: 18.04.2023).

[32] Jakob Nielsen and Rolf Molich. "Heuristic evaluation of user interfaces". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1990, pp. 249–256.

[33] nikamjaydipashok11. *Advantages and Disadvantages of Assembler*. URL: `https://www.geeksforgeeks.org/advantages-and-disadvantages-of-assembler/`. (accessed: 18.04.2023).

[34] Oracle. *Java Documentation*. URL: `https://docs.oracle.com/en/java/`. (accessed: 23.05.2023).

[35] John Pickens. *6502asm*. URL: http://www.6502.org/tutorials/6502opcodes.html. (accessed: 18.04.2023).

[36] Java Point. *Software Configuration Management*. URL: https://www.javatpoint.com/software-configuration-management. (accessed: 13.05.2023).

[37] Tutorials Point. *Big-Bang Testing*. URL: https://www.tutorialspoint.com/software_testing_dictionary/big_bang_testing.htm. (accessed: 10.05.2023).

[38] Tutorials Point. *Incremental Testing*. URL: https://www.tutorialspoint.com/software_testing_dictionary/incremental_testing.htm. (accessed: 10.05.2023).

[39] Tutorials Point. *Sandwich Testing*. URL: https://www.tutorialspoint.com/what-is-sandwich-testing-definition-types-examples. (accessed: 10.05.2023).

[40] pppankaj. *Difference between LL and LR parser*. URL: https://www.geeksforgeeks.org/difference-between-ll-and-lr-parser/. (accessed: 24.04.2023).

[41] Muhammad Raza. *Quality Assurance (QA) in Software Testing: QA Views and Best Practices*. URL: https://www.bmc.com/blogs/quality-assurance-software-testing/. (accessed: 11.05.2023).

[42] Linda Rosencrance. *Functional Specification*. URL: https://www.techtarget.com/searchsoftwarequality/definition/functional-specification. (accessed: 10.05.2023).

[43] Jeffrey Rubin and Dana Chisnell. *Handbook of usability testing: how to plan, design, and conduct effective tests*. 2nd ed. Wiley Pub, 2008. ISBN: 9780470185483.

[44] Robert W Sebesta. *Concepts of programming languages*. en. Jan. 2016. ISBN: 9781292100555.

[45] Alexander Shvets. *Dive Into Design Patterns*. URL: `https://refactoring.guru/design-patterns/visitor`. (accessed: 02.05.2023).

[46] Stian Soreng. *6502asm*. URL: `https://github.com/inindev/6502asm`. (accessed: 18.04.2023).

[47] IEEE Spectrum. *Chip Hall of Fame: MOS Technology 6502 Microprocessor*. 2017. URL: `https://spectrum.ieee.org/chip-hall-of-fame-mos-technology-6502-microprocessor`. (accessed: 11.04.2023).

[48] Stranger1. *Single pass, Two pass, and Multi pass Compilers*. URL: `https://www.geeksforgeeks.org/single-pass-two-pass-and-multi-pass-compilers/`. (accessed: 18.05.2023).

[49] Kunal Tandon. *Why pointers lost their popularity in Java, C# and other modern languages*. URL: `https://medium.com/developers-arena/why-pointers-lost-their-popularity-in-java-c-and-other-modern-languages-2ad6dba5c847`. (accessed: 26.04.2023).

[50] MOS Technology. *MCS6500 Microcomputer Family Programming Manual*. 1. ed. Motorola Semiconductor Products Inc., 1976. URL: `http://archive.6502.org/books/mcs6500_family_programming_manual.pdf`.

[51] Bent Thomsen. *Languages and Compilers*. URL: `https://slideplayer.com/slide/16680897/`. (accessed: 04.05.2023).

[52] Tutorialspoint. *Compiler Design - Syntax Analysis*. 2014. URL: `https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm`. (accessed: 16.04.2023).

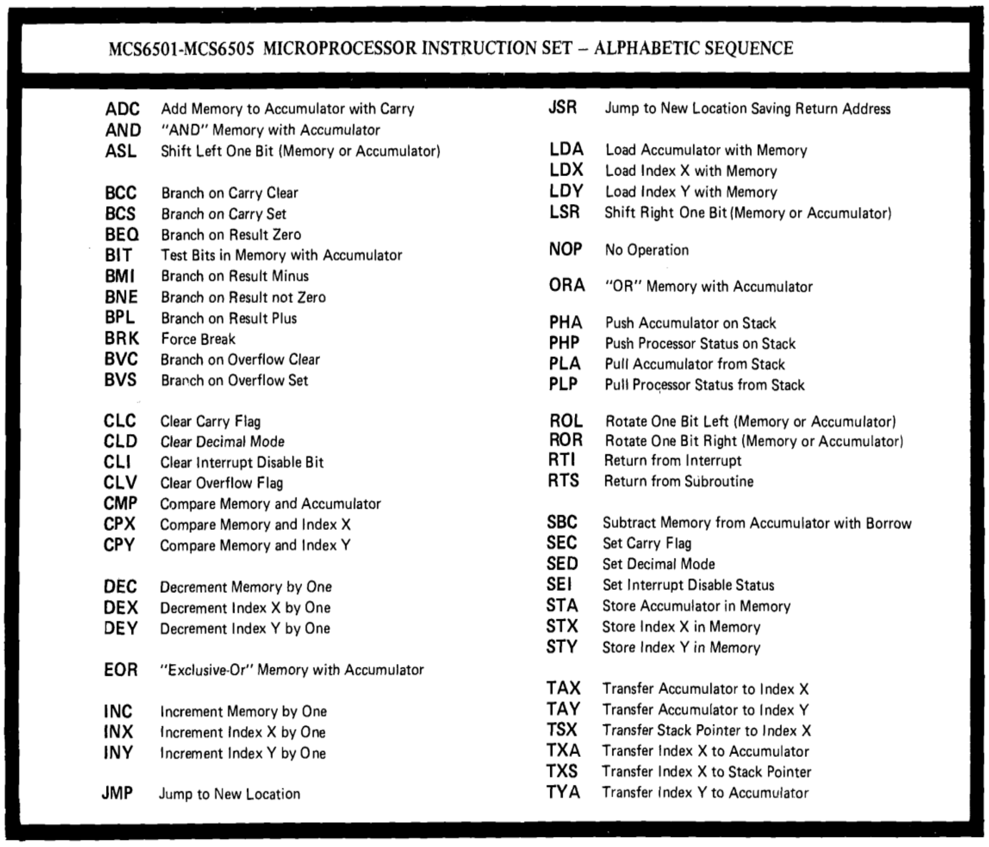
[53] Wikipedia. *LL parser*. URL: https://en.wikipedia.org/wiki/LL_parser#Constructing_an_LL(1)_parsing_table. (accessed: 18.04.2023).

[54] Wikipedia. *LR parser*. URL: https://en.wikipedia.org/wiki/LR_parser. (accessed: 18.04.2023).

[55] J. Todd McDonald William Mahoney. *Enumerating x86-64 – It's Not as Easy as Counting*. URL: https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf. (accessed: 02.05.2023).

Appendices

Appendix A



MCS6501-MCS6505 MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

Figure 1: The full instruction set for the 6502 assembly language [50]

Appendix B

Gmail Anders Youssef <andersmazen@gmail.com>

Writing a story about the 6502 processor

Bill Mensch <bill.mensch@westerndesigncenter.com> 3. april 2023 kl. 18.59
Til: Anders Youssef <andersmazen@gmail.com>

Hi Anders,

What grade and school do you go to?

What class is this for?

Here is a link to one of the more exciting things the WDC 6502 is being used for in education, FYI. We just restarted teaching 6502 System-on-Chip (SOC) design with tapeout (manufacturing experimental chips for SOC educational projects).

The WDC 6502 has been licensed to companies all over the world for automobiles and pacemakers for two examples. Our chips are used in security systems, life support medical devices for pacemakers and defibrillators, railroad control systems, communication systems, various educational systems such as Ben Eater's 6502 Computer Kit, SmartyKit, WDC's Single Board Computers (SBC) such as our SXBs, MENSCH™ Microcomputer and MyMENSCH™.

As you can see on our Where to Buy page we have worldwide distribution of our chips and boards.

We sell our 65816 Programming books through Amazon.

Have you seen the five (5) minute video at Team6502.org?

Have you seen all of the Maker, Hobbyist and Hacker project list at 6502.org?

Have you visited TheMenschFoundation.org website for enriching Computer Science & Engineering (CSE)?

Were you aware we have The 6502 Museum co-located at the business home of the WDC 6502 at 2166 E. Brown Rd. In Mesa, Arizona?

Were you aware we have six Mensch Prizes for Innovation at ASU, Barrett Honors College, The UArizona Franke Honors College and The UArizona College of Engineering?

Did I answer your questions?

Do you have more questions?

Best,

-Bill

[Citeret tekst er skjult]

Appendix C

Problem list

Intro: Assume that you are a programmer familiar with writing explicitly typed languages. Using this knowledge, attempt to complete the following assignments. There will also be 5 follow-up questions.

The tests were conducted in Danish, and all answers were translated to English afterwards.

All of the test participants were Software students form different semesters, as a prerequisite for the test was to have experience with explicitly typed languages. Besides this, the tests aim to encompas as wide a demographic as possible.

Each problem, or each group of problems, is colour coded and marked with a matching “x” for each participant that experienced the problem.

Assignment	Problem description	Category	Problem experienced				
			Participant 1	Participant 2	Participant 3	Participant 4	Participant 5
Try to explain what the following code does. You may use this code as inspiration in the following assignments. (Expected 1 min)	Does not notice that “a” is declared, but corrects mistake after receiving a little hint Experiences a small difficulty with reading the if-statement	Cosmetic Cosmetic		x	x		
Try to declare an integer, float, and two boolean variables. The names are up to you. (Expected 1 min)	Uses a bit extra time, had to fix typos Initialises float with an integer value (1 instead of 1.0) Uses 1 and 0 as boolean values. Slight delay because of not-included float declaration example, but successfully completes assignment. A bit unsure about types and their	Cosmetic Serious Cosmetic Serious Serious	x	x	x	x	x

	possible values. Attempted to declare several instances of the same variable type by putting commas between. Writes bool instead of boolean. Uses type name as variable name (for example int int = 2). Uses a bit more time than expected. Required a bit more clarification before starting the assignment. Uses double instead of float type. Asks if ".", or "," is used for separating decimals in float numbers. Uses some extra time to ask questions and use code examples as reference						
Try to add 2 to the previously declared integer number and save it in a new variable. (Expected 1 min)	Missing explicit type in new declaration, corrects it quickly	Cosmetic			x		
Give the boolean variables a value, and write an if statement using an "and-operation" including the two boolean variables. In the body of the if-statement, add 2.5	Used compound assignment at first Tries to use "print" Tries to add to variable without assignment operation	Critical Cosmetic Cosmetic	x	x	x	x	x

to the float variable. (Expected 1 min)	Missing the curly brackets initially - Realizes the mistake themselves Asks some clarifying questions		
Add a nested if-statement to the one previously written. Add an else statement to the new if-statement, which adds 3.14 to the float variable. (Expected 1 min - tager reelt set 2-3)	Asks for further clarification of the assignment. Missing eof Uses comma for float number instead of period Writes boolean values where the first letter is capitalised Expected feature multiplicity Finds it annoying that you can not write 1f for floats Delay due to syntax fixes, takes a while to get the program running Tries to compare variables of different types in the expression of the if-statement.	Critical Serious Serious	🟥x | 🟥x🟦x🟩x | 🟥x | 🟩x🟥x | 🟧x🟥x

Follow-up questions:

From 1 to 10, how would you rate the readability of the provided code example (from question 1)?

- Participant 1: 8, It is like C#..
- Participant 2: 6, it is fine, would personally rate it 7 but believes that it would be more confusing for others. Proceeds to compare the language to C, but mentions that boolean values are better, since they are true and false rather than 0 and 1. Did not like the optional parenthesis inside if-statement. BYE-eof symbol is arbitrary, and is strange, since the rest of the language is not “natural-language-like”.
- Participant 3: 8, really easy to understand, not in doubt about what anything means.
- Participant 4: 10, very, very simple, a bit like Rust
- Participant 5: 9, fairly easy and makes logical sense

From 1 to 10, how smooth was your personal code writing experience?

- Participant 1: 10
- Participant 2: 8, missed some operators, could fix it fairly easily, missed BYE, “boolean” is a bit too long, would rather use “bool”.
- Participant 3: 6, BYE is useless, otherwise fairly smooth
- Participant 4: 9
- Participant 5: 9, pretty smooth, easy and logical ret smooth

From 1 to 10, how difficult did you find the assignments?

- Participant 1: 1
- Participant 2: 3
- Participant 3: 1
- Participant 4: 1
- Participant 5: 2

Inspect the generated code in “output.txt”. How would you describe it compared to the code you entered?

- Participant 1: “Difficult. Unintelligible. You could try to make sense of it, but I don’t want to.”
- Participant 2: “Really ugly. Unreadable. Impressive that it can be translated to this. The two languages are incomparable. Would take a lot longer time to write 80 lines of this, instead of 15 lines of your language. I wouldn’t even try.”
- Participant 3: “I can’ read assembly, I don’t know if any normal person can. Unreadable and unintelligible.”
- Participant 4: “The two languages seems completely unrelated”, finds one line of code the could look like EOF. “That i can’t interpret. Has nothing to do with me”
- Participant 5: “Not very easy to read. Not very logical. Not very comprehensible. Long. Messy. Confusing.”

For a better code-writing experience, if you had to modify the source language, what would you change?

- Participant 1: feature multiplicity for compound assignments
- Participant 2: add operations, missing compound, forget BYE or replace with END, that floats can’t be whole numbers i.e. convert 1 to 1.0, you could force indentation - warnings, equal signs in an if-statement are comparisons
- Participant 3: add compound assignment, remove BYE, liked the semicolon
- Participant 4: (add object oriented lol) text types, char, string, array, pointer
- Participant 5: add floats and integers together, nice and simple, cool with BYE kinda like return 0, otherwise not very intuitive

Appendix E

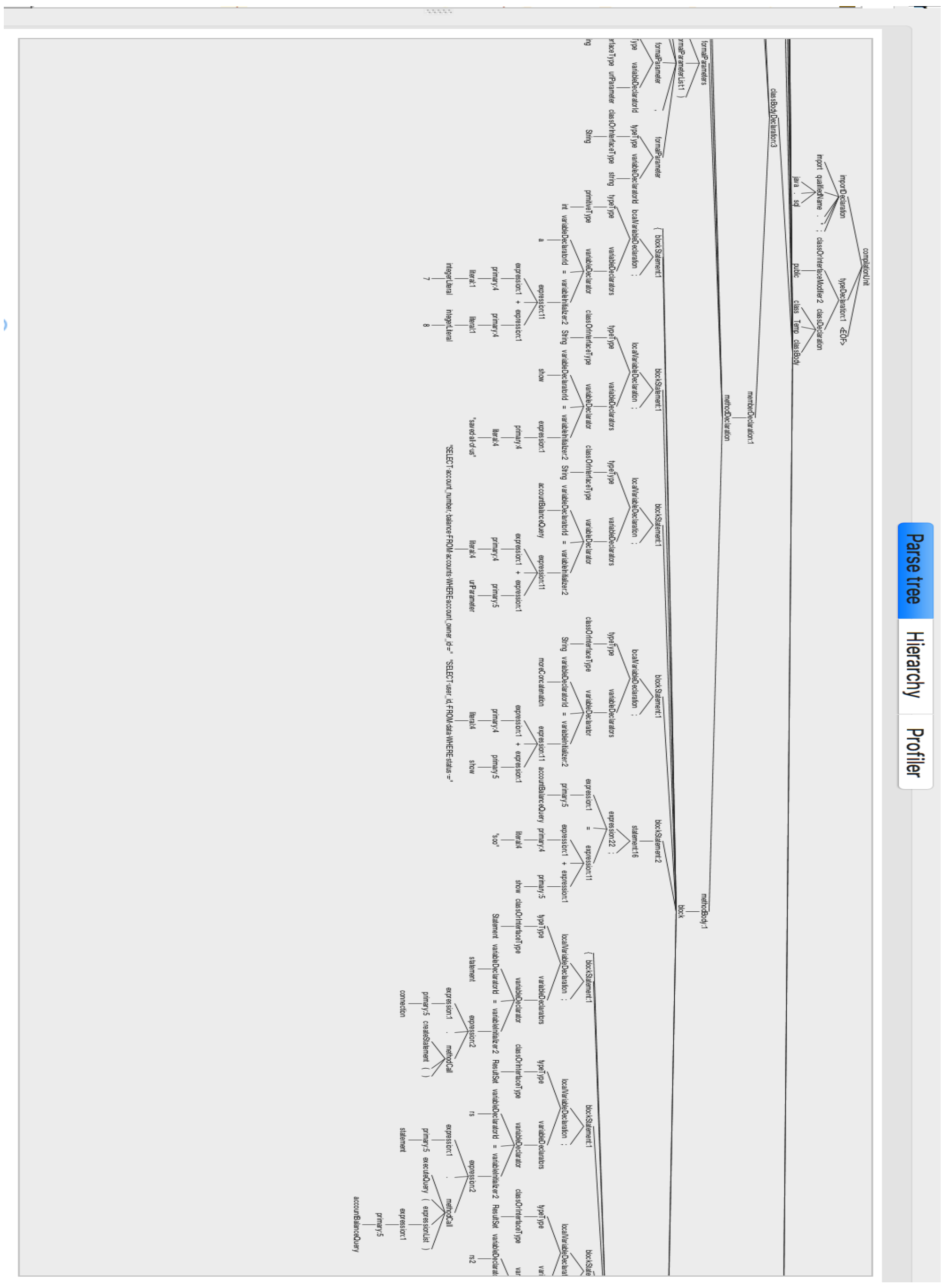


Figure 2: An example of a parse tree made by the parser in ANTLR [6]

Appendix F

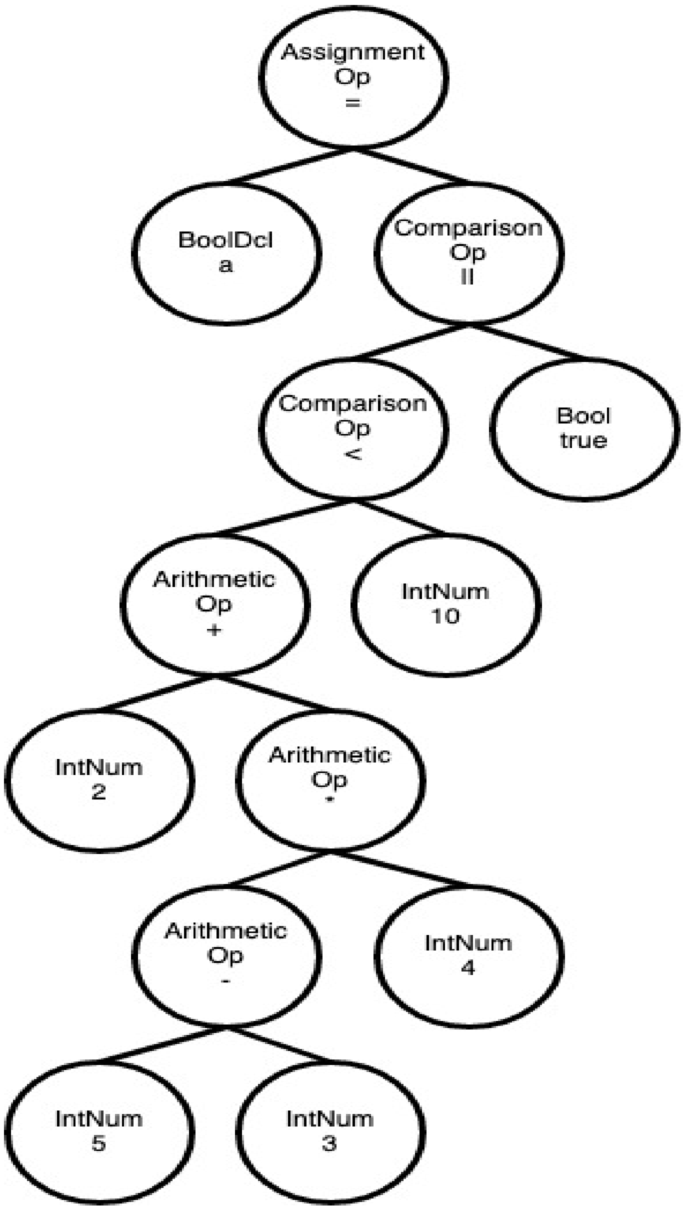


Figure 3: AST before constant folding

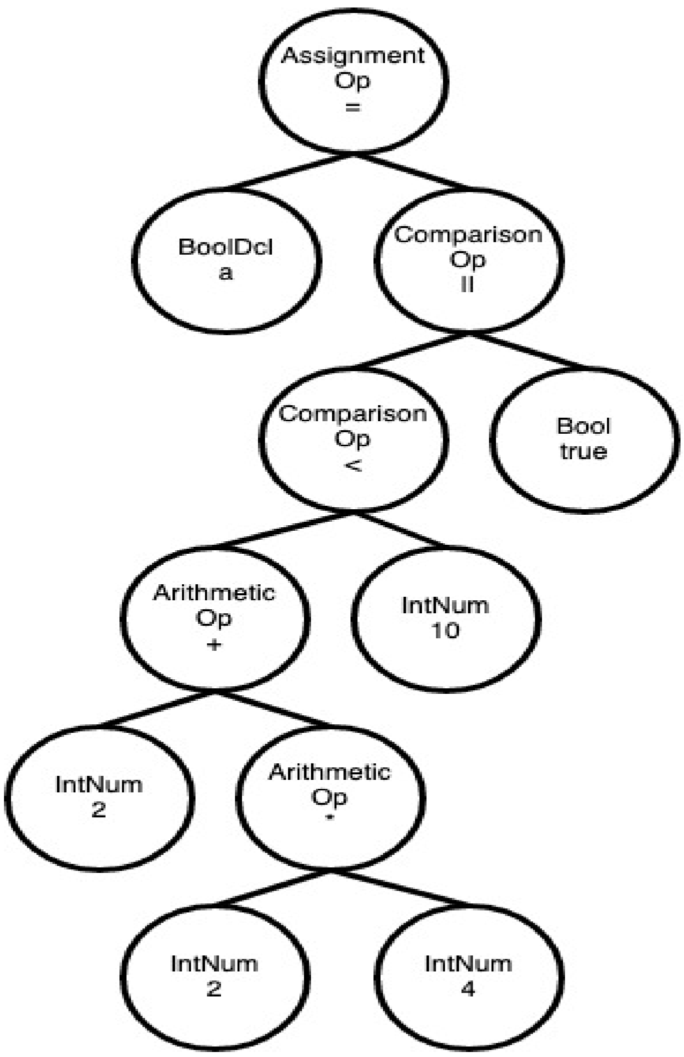


Figure 4: Constant folding: step 1

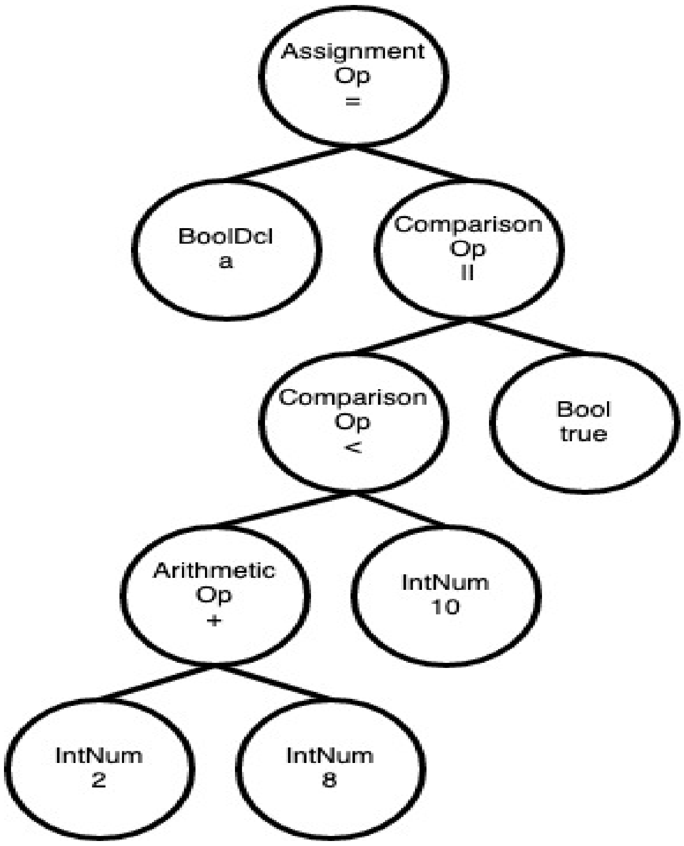


Figure 5: Constant folding: step 2

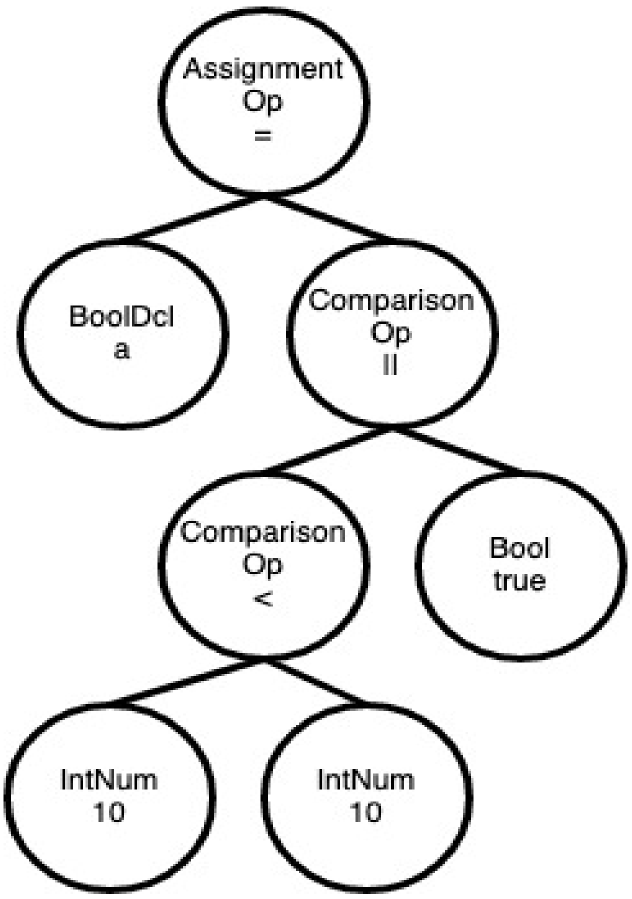


Figure 6: Constant folding: step 3

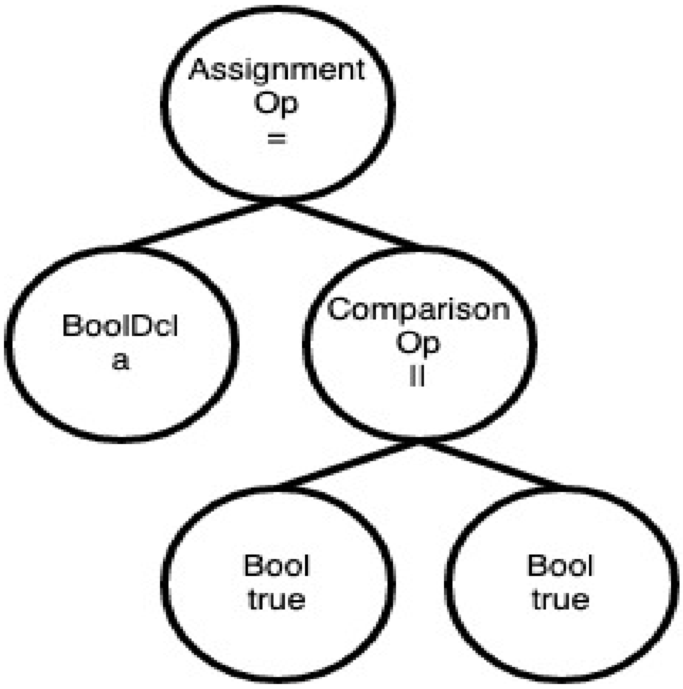


Figure 7: Constant folding: step 4

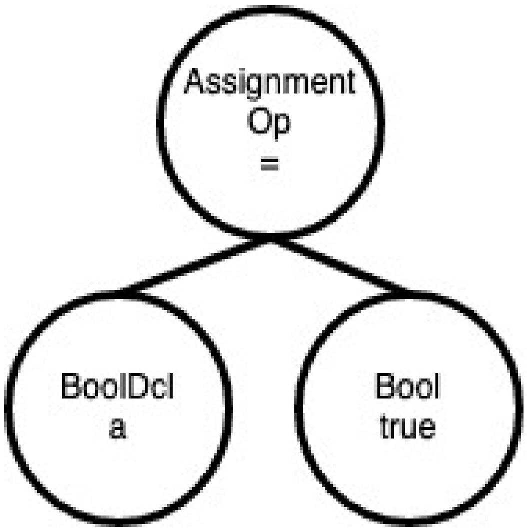


Figure 8: AST after constant folding

Appendix G

Bootstrapping was not researched beyond its fundamental principles, nor implemented at any level. Bootstrapping would, if the target system's memory capacity was larger, enable compilation on the machine. This would also mean that the 6502 processor could compile and execute the language from this project independently.

The steps of bootstrapping are the following, producing a new language L for machine A [27]:

1. Create a ${}^{S}C_{A}{}^{A}$ compiler for subset S of the desired language, L using language "A" and that compiler runs on machine A.

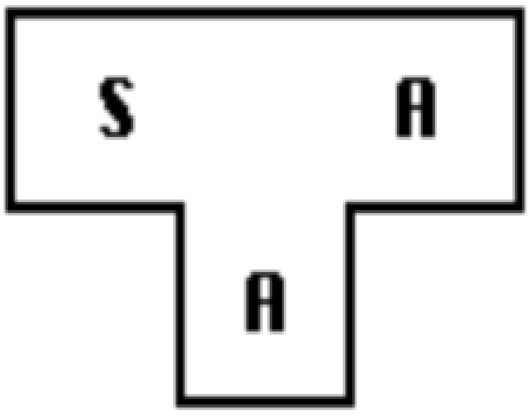


Figure 9: ${}^{S}C_{A}{}^{A}$ compiler[27].

2. Create a ${}^{L}C_{S}{}^{A}$ compiler for language L written in a subset of L.

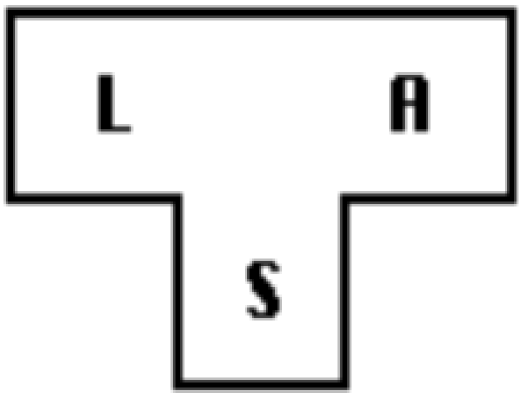


Figure 10: ${}^{L}{C_S}^{A}$[27].

3. Compile ${}^{L}{C_S}^{A}$ using the ${}^{S}{C_A}^{A}$ compiler to obtain ${}^{L}{C_A}^{A}$. ${}^{L}{C_A}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.

$${}^{L}{C_S}^{A} -> {}^{S}{C_A}^{A} -> {}^{L}{C_A}^{A}$$

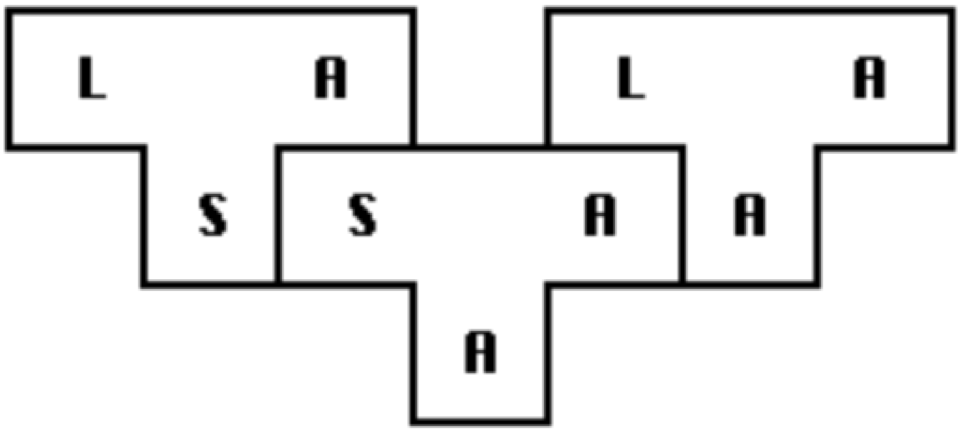
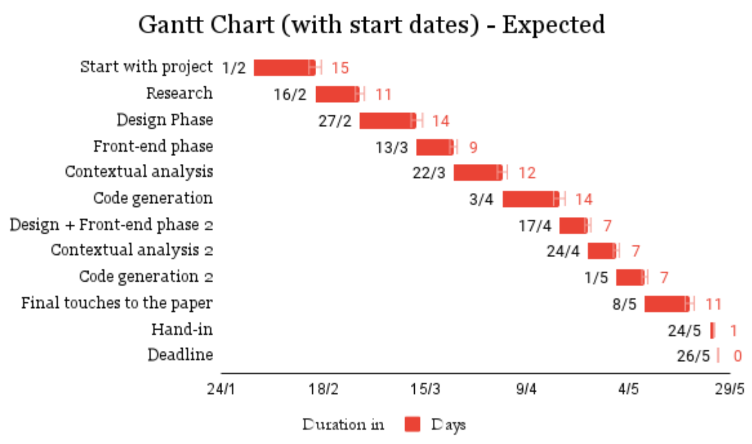


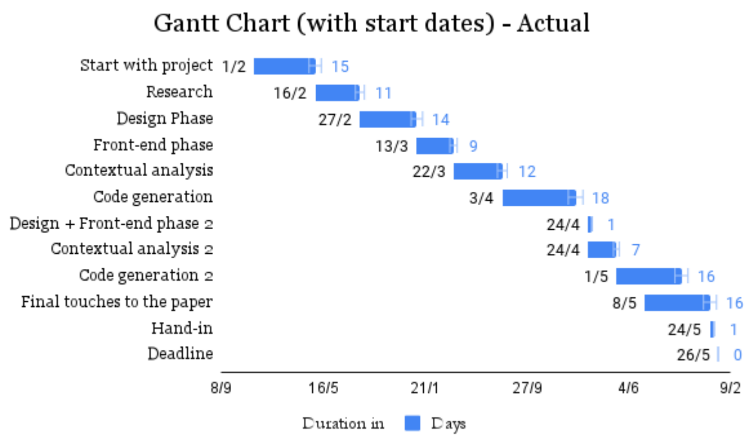
Figure 11: Compiling ${}^{L}{C_S}^{A}$ using the ${}^{S}{C_A}^{A}$ compiler to obtain ${}^{L}{C_A}^{A}$ [27].

Implementing the compiler in a subset of its own language makes one less dependent on the target platform. This leads to more portable implementation.

Appendix H



Appendix I


Gantt Chart (with start dates) - Actual
Start with project 1/2 15
Research 16/2 11
Design Phase 27/2 14
Front-end phase 13/3 9
Contextual analysis 22/3 12
Code generation 3/4 18
Design + Front-end phase 2 24/4 1
Contextual analysis 2 24/4 7
Code generation 2 1/5 16
Final touches to the paper 8/5 16
Hand-in 24/5 1
Deadline 26/5 0
8/9
16/5
21/1
27/9
4/6
9/2
Duration in
Days