Buff

A math inspired, beginner friendly, nearly pure functional programming language

Christopher C. Jensen, David S. H. Christiansen, Jacob T. Rasmussen, Laurits C. B. Mumberg, Lasse D. Skaalum

Software, SW414F21, 2020-05

AALBORG UNIVERSITY
STUDENT REPORT

Software
Aalborg University
https://www.aau.dk/

Title:

Buff

Project:

P4-project

Project period:

February 2021 - June 2021

Project group:

Group SW414f21

Participants:

Christopher C. Jensen
David S. H. Christensen
Jacob T. Rasmussen
Laurits C. B. Mumberg
Lasse D. Skaalum

Supervisor:

Henrik Sørensen

Page Numbers: 83

Date of Completion:

June 25, 2021

Abstract

This paper dives into the implementation of a mathematical domain specific programming language called Buff. It is designed for easy adoption by programming beginners who have knowledge of high school mathematics. The paper and the project has been developed by a fourth semester student group at Aalborg University studying a bachelor's degree in software engineering.

Buff is a functional programming language that, to a very high extend, shares its syntax with mathematics. In most aspects, the language is a purely functional language, meaning that it does not include side effects. The design and implementation of Buff found inspiration through research papers and interviews regarding difficulties that beginners might encounter when learning programming.

An accompanying compiler has been implemented for compiling Buff to JavaScript. The parser creation tool ANTLR has been used in the development of Buff's compiler.

Preface

We are a fourth semester project group studying a bachelor in Software Engineering at Aalborg University. We have accumulated programming experience through the previous semesters, hobby projects and jobs. In our first semester we were introduced to the C programming language as well as some coding guidelines and conventions. Since then, we have also worked with JavaScript and C#. This has led us to reflect on the different languages and their strengths and weaknesses. We find it very interesting what factors made the languages easier or harder to learn, especially as we did not agree completely on this in the group. However, we can agree on the fact that learning to program for the very first time was harder than learning a second or third programming language, regardless of which programming language was the first. This thought process made us more curious about the difficulties related to beginners learning to code. As a result, we started wondering if some programming languages are better suited for beginners than others.

This semester includes 3 courses: "Computer architecture and operating systems" (CAOS), "Syntax and semantics" (SS) and finally "Languages and compilers" (SPO). The courses will play a crucial role in the process of this project as they will provide us with the knowledge for implementing a language and a compiler from scratch. CAOS will teach us the underlying theory and inner workings of computers and operating systems. SS will provide us with the mathematical background and theory behind computation and programming languages. SPO will be very important for the practical implementation of a language and a compiler.
This report has been written as a fourth-semester project at Aalborg University's Institute of Computer Science in 2021 by Group SW414f21.

The compiler for the programming language has been implemented in Java, as this is the language, which our course, SPO, encouraged us to learn and use.

We would like to express our gratitude to Henrik Sørensen, our project supervisor, for his unwavering support during the project.

Furthermore, we express our gratitude to Kurt Nørmark for participating in an interview leading to new findings and backing of others.

Contents

1 Introduction

Learning to program can be a cumbersome and tiring process. Everything from understanding the fundamental structures of programming languages to deciphering the syntax may seem overwhelming at first - even understanding the various scenarios in which you can use characters like '=' and '+' can be daunting when you are starting out.
Therefore, this report addresses the problem of programming having a steep learning curve. Firstly the common difficulties that exist when learning your first programming language are investigated, followed by how a new programming language can accommodate these difficulties, having a positive effect on beginner's ability to learn programming. Part of this process will include investigating which programming paradigm is suitable for a beginner friendly programming language.

The analysis considers whether a mathematical approach and the functional programming paradigm can be a good option for a beginner programming language. The findings of the analysis create the foundation for the design of a mathematically domain specific programming language called Buff. The design process presents the formal definition of Buff in terms of syntax and semantics and results in implementation specific requirements. An implementation and test section presents interesting code snippets and the decision making behind. Afterwards, a discussion chapter sums up the important decisions of the project and discusses the outcomes that emerge as a result of those decisions. Furthermore, the discussion also contains a future works section which addresses some shortcomings and desirable features, that should be implemented in future development. Finally, the report comes to an end with a conclusion that presents whether or not the problem statement is considered to be fulfilled.

Buff is implemented in Java with the help from a parser generator called ANTLR. The source code can be found in appendix A, and is discussed in chapter 4. The relating code documentation can be found in appendix B. Finally, to ensure correctness of our language and compiler, software testing has been conducted which is described throughout chapter 5.

2 Problem analysis

This chapter will include an analysis of the problem domain for our project. It takes offset in an initial problem that provides a basis for the following exploration and research. The main part of the analysis consists of research and arguments that help us steer the project in the most interesting direction. The analysis culminates to a narrow problem and its accompanying problem statement, which we will attempt to find a solution to during the project. Figure 2.1 gives a visual representation of the problem analysis process, and displays the sub-conclusions, indicated by the light bulbs, that were made along the way.

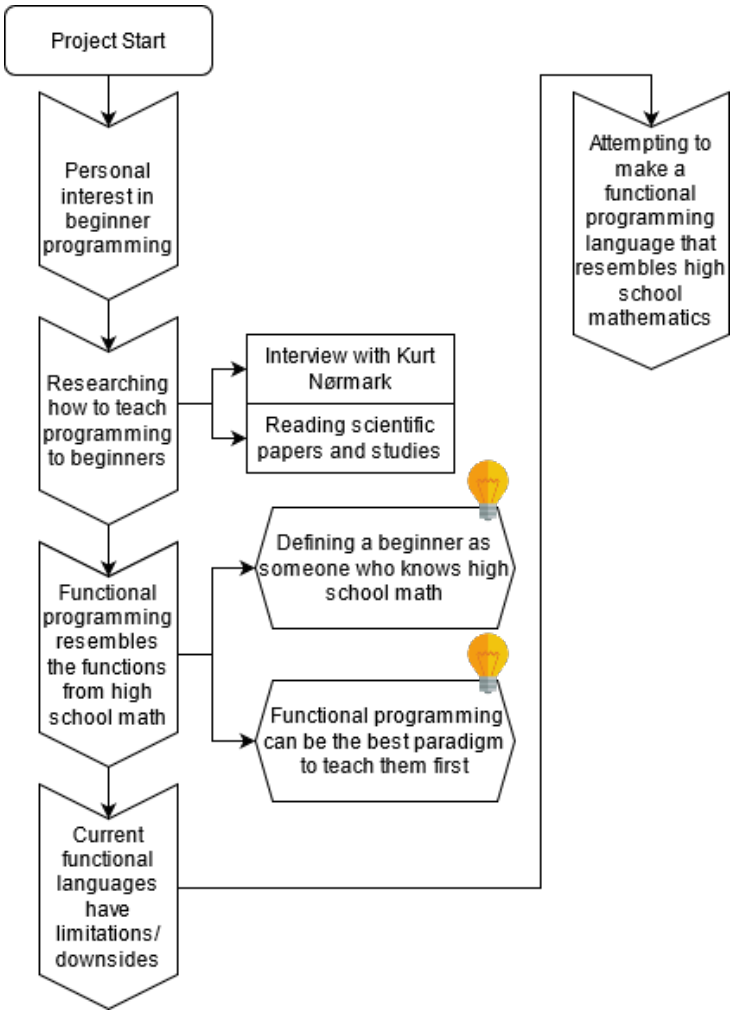


Figure 2.1: A flowchart that shows the process of the problem analysis

2.1 Initial problem

Considering the points of interest mentioned in the preface, we start our problem analysis with the following initial problem: **What is the best way of teaching programming to beginners?**

2.2 Difficulties related to learning programming

Learning to program as a beginner is for many people not an easy task. To solve programming problems you need an understanding of programming techniques and concepts, which might not come naturally to everyone. As a result of the relevance of programming and the rise in the amount of new programmers, a large amount of research has been conducted on the subject of difficulties related to learning programming.
A fairly extensive literature review was made in 2017, attempting to gather and combine the research and findings of the last 40 years on the subject [1]. The study sets out to identify the difficulties and misconceptions of beginner programming students, and categorizes them into the following three categories:

- Syntactic knowledge
- Conceptual knowledge
- Strategic knowledge

Problems with **syntactic knowledge** commonly relate to misuse of parentheses and incorrect understanding of operators, for example the implementing the `for-loop` in C, where one might not know the correct order of initialization statement, test expression and update statement. The study found that complex tasks can increase the cognitive load and cause beginners to forget basic syntax. How substantial of an issue this is, is also dictated by how much help the beginner gets from their editor. Advanced editors with good highlighting can help prevent these problems, as it can reduce this cognitive load.

Most errors that beginners make are minor misunderstandings related to syntactic knowledge. These errors are superficial and can easily be fixed. Problems with **conceptual knowledge** are more significant, as it relates to a person's mental model of a program, and how it should execute. An example of this, is that prior math knowledge might confuse the student, as many of the symbols used in most programming languages have a different meaning in math. One such example is the equals sign used in variable assignments, which means something different in most programming languages. For example is the expression "`x = x + 1`" considered legal in many programming languages.
Another example of a difficulty that results in conceptual misunderstanding, is when programming languages use words from natural language in ways that might be unintuitive for

beginners. The study gives the example of the word "and", which in some languages is used as a boolean operator, while some beginners understand it as a conjunction.

Strategic knowledge is the final category and refers to the understanding of what strategy or pattern should be used to solve a problem. A beginner might have good syntactic and conceptual knowledge, but still not know how to combine these into a coherent plan for a solution. An example of a difficulty this could create, is the inability to see what loop structure would be the right one for a given situation.

The study concludes that there are many sources for misconceptions and difficulties for beginners, and that many sources for these difficulties have connections to the beginners prior knowledge [1].

2.3 Predictors of succes

Besides difficulties related to learning programming it is also important to consider what predictors of success come into play.
The predictors of success in learning programming is a subject of which a lot of research has been conducted. Some of the predictors found are the student's level of:

- Ability of logical thinking and problem-solving
- Attention to detail
- Consideration of details
- Mathematics
- Knowledge of programming
- Ability to read

[2]

As it would cause great complexity to incorporate all of these features within the scope of this project only a select few are chosen. The factors that we find the most interesting are *ability of logical thinking and problem-solving*, *mathematics* and *ability to read* [2]. The factors *ability of logical thinking and problem solving* and *mathematics* are of interest, as both logic and mathematics are the basis of programming. This would also support a common ground in mathematics which is supported by Buff. The factor *ability to read* is interesting as Buff aims to remove symbols that are not known from math, such as "&&", "==" and "||" and replace them with words that represent their meaning. An example could be the "==" operator being represented by the word "IS". This is done because most of the boolean operators known from mathematics

like ¬, ∧ and ∨ do not exist on a standard keyboard.

Ability of logical thinking and problem solving and *mathematics* are both covered by the functional programming paradigm, as functional programming builds on assertions and mathematical expressions, hence striving towards a 1:1 correlation with mathematics [3]. The predictor *ability to read* can at first glance seem somewhat hard to implement in a programming language, but this is what Python have tried to do with their minimalist syntax design. An example of Python syntax can be seen in code snippet 1. The snippet shows the syntax for iterating over a list in python and can almost be read as plain English. One might read the code aloud in their head as *"For every car in car_brands_list print that car"*. As functional language encapsulate the first two principles, it would be advantageous to combine a language such as python because of its readable syntax with a language like Haskell which claims to be a pure functional language, but has presented some problems related to readability. Some of the downsides of the syntax of Haskell is further explored in section 2.6.1.

```python
################### For loop ###################
car_brands_list = ['VW', 'Alfa Romeo', 'Range Rover', 'Tesla']

for car in car_brands_list:
    print(car)

################### Function definition ###################
def addTwo(num):
    return num + 2

addTwo(4) #Returns 6

################### List comprehension ###################
even_numbers = [x for x in range(101) if x % 2 == 0] #A list with even numbers
↪  from 0 to 100

################### If conditions ###################
if 2 + 2 == 4:
    print("Math")
else:
    print("Weird")
```

Code Snippet 1: Example of python code

2.3.1 Interview findings

In addition to investigating research papers, we conducted an interview with Kurt Nørmark who is associate professor at Department of Computer Science at Aalborg University with

focus on the principles of good programming and the difficulties new programmers meet. Nørmark has 25 years of experience in teaching beginner programmers where he inevitably has encountered problems. The interview followed the semi-structured model allowing us to pursue interesting topics that we encountered during the interview. From the interview the key points are:

- Beginners have trouble understanding the assignment "=" operation as it differs from their mathematical understanding of the equals "=" character.
- Understanding of the mathematical function is of utmost necessity to understand the concept of a function in programming languages
- It is difficult to transition from imperative to functional programming and vice versa

Furthermore Nørmark stated that if he were to decide the curriculum, for the first semester programming course that he teaches, he would start out by teaching a pure functional language as it is "the purest form of programming", meaning it resembles mathematics the most. Looking back at the findings it was stated that a beginner programmer's mathematical capabilities was a predictor of success in learning programming, which supports Nørmark's findings. This indicates a notion that a functional programming language might serve as a good first language for beginners. Beyond the problems encountered throughout his teachings, Kurt Nørmark also mentioned some programming principles one should follow when developing software. However he did not insist on forcing a programmer to obey to the principles, if a given situation made it more favorable to disregard the principles. From the interview the following principles were discussed:

- Abstraction (divide and conquer) as no one understands 1000 lines of code, but rather small individual bits that constitute a whole.
- Avoid jump commands, as it leads to confusion.
- Do not be overly smart, as no one will understand the code. This can be translated to the principle "Keep It Simple, Stupid" (KISS).

Following the interview we discussed the principles with a focus on how they best can be taught to a beginner. Additionally, we considered if the design of a programming language can be developed to naturally embed these principles into the beginner by simply using the language.
It was found that such principles are hard to enforce in a compiler, hence it will not be the focus of this project, however the language will still strive towards some way of enforcing these principles. A principle that could be strived towards is **divide and conquer**, by creating a functional language as it does not allow for mutable data, hence forcing the programmer to use functions to abstract over data instead of variables [4].

The second, **avoid jump commands**, is easily avoided by not including the functionality in the language. It is also possible to go one step further like Java and reserve the keywords such as `goto`, but give them no meaning ensuring one cannot misuse it by mistake, as the parser would detect the keyword and emit a syntax error because the token does not belong in that context [5][6].

The third, **do not be overly smart**, does not suggest any means of solving this issue through a programming language, but rather the mindset of the programmer should be changed. This mindset can however be somewhat influenced by not including the functionality that allows one to be overly smart such as expressions like `a[i++] = a[--j]` where a is an array. These functionalities correlate with the discussion of orthogonality and simplicity found in section 3.2 hence it will not be further discussed here.

On the basis of this section, the functional programming paradigm is considered to be an interesting option for a beginner language, with a focus on previous knowledge and skills within mathematics. As a result, Buff will be a functional programming language.

2.3.2 Beginner definition

From section 2.3 it was found that mathematical skills and understanding was a predictor of success in learning programming. Furthermore, Kurt Nørmark stated that to understand the programming concept of functions one first needs to understand the mathematical concept of a function. Derived from this, our definition of a beginner is presented in definition box 2.3.1 and carrying on from now this will be the definition of a beginner that the report refers to.

Definition box 2.3.1: Beginner

Someone who has completed at least the highest level of high school mathematics and has no prior experience or knowledge in programming

With this definition of a beginner in mind we think that it would be interesting to further explore how best to teach programming to an individual with the stated level of mathematical knowledge. The interview with Kurt Nørmark combined with our additional findings point towards functional programming as a good paradigm for beginners as it resembles their mathematical knowledge the most.

Our definition of mathematical syntax known from high school can be seen in definition box 2.3.2.

Definition box 2.3.2: High school Mathematics syntax

Function: f(x) = x
Plus: x + y
Minus: x - y
Multiplication: x * y
Division: x / y
Power: x^y
Square root: $\sqrt{x}$
Greater than: x > y
Greater than or equal to: $x \geq y$
Smaller than: <
Smaller than or equal to: $x \leq y$
Not equal to: $x \neq y$

2.4 Purity of the language

If a functional programming language is said to be *pure*, it means that every computation is evaluated as a mathematical expression, operations have no side effect and data is immutable [7]. However, if all of this were to be fulfilled in a language, programs written in the language would not do anything but heat up your PC. The reason being that any side effect such as writing to the screen, reading input data, logging files or other things would not be permitted, while at the same time the program could not alter any data as all data is immutable in such a language [7]. The result is a program that throws all data away at termination consequently only putting extra load on your PC. Therefore, the discussion of whether or not a language is pure is somewhat controversial. One approach to achieve a pure language, is to state that it must wrap all impure functionalities in a special language construct, such that it cannot interact with basic datatypes even tough they might be similar. A language that makes use of this is Haskell [7]. This ensures that no impure data is mixed with pure data, which is depicted in code snippet 2, where it is attempted to concatenate a regular string with a string received from the user, which results in an exception being thrown. Another approach is simply to allow for some impurity in your language, but limiting it to the absolute necessary and no more [7]. The latter will be the case for Buff.

```
"Hello: " ++ "World!" -- Correct: string + string
"Name: " ++ getLine -- Exception: string + IO string
```

Code Snippet 2: Haskell language constructs. *Haskell* [8, HaskellDoc]

2.5 Teaching functional programming to beginners

Using a functional language to teach programming to beginners is not an original idea. Functional programming is a well established programming paradigm and has already been taught at some universities as an introduction to programming. In 2004 a study was made at the University of New South Wales, with the goal to evaluate the risks and benefits of teaching purely functional programming to first year students [9].

The study states that a freshman course in programming should not focus on a specific paradigm, but should teach elemental programming concepts and programming techniques. The study argues that the light syntax, clean semantics and high level of abstraction found in purely functional languages allows the focus to not be on the language, but instead on these concepts and techniques. It is boldly claimed that first year students should be *"liberated from the tyranny of syntax"*.
Instead, three principles that the authors believe an introductory course in programming should aim for is presented. They are as follows:

- Convey the elementary techniques of programming (the practical aspect).
- Introduce the essential concepts of computing (the theoretical aspect).
- Foster the development of analytic thinking and problem solving skills (the methodological aspect).

These three cover much of the same things as the predictors mentioned in section 2.3. In the study, it is proposed that functional programming supports these three aims well, as clean and simple syntax allows the focus to be on good practice early on. A tight interplay between theory and practice helps with the first two aims, and functional programming is expressive enough to allow complicated problems to be tackled early on.

2.5.1 Learning good programming concepts with functional programming

In the book *Get Programming With Haskell* [10] the author Will Kurt argues that learning to program in Haskell, because of its pure functional nature, forces the programmer to adapt healthy programming concepts. The book emphasizes that thinking in Haskell will improve your abstraction capabilities, and can help you avoid bugs.

In the project group we see this as a strong benefit for new programmers. The statements from *Get Programming With Haskell* [10] align well with the viewpoints from Professor Kurt Nørmark that were mentioned in section 2.3.1. Both sources focus on the concept of abstraction as being a key principle in programming, which encourages us to highlight the notion that functional programming can stimulate the understanding of abstraction with regards to

programming. However as mentioned in section 2.4 the syntax of Haskell can be difficult to read and understand which we will elaborate on in section 2.6.1.

2.6 Issues with functional programming for beginners

The study mentioned in section 2.5 [9], also highlights some of the pitfalls they experienced related to teaching functional programming to beginners. An issue arises from the fact that concepts like lambda expressions and higher-order functions are seen as very integral to the paradigm, but can be hard for beginners to understand. The author's solution was to restrict themselves to very basic higher-order functions such as `map` and `filter`, and classify them as an advanced topic.
Another big issue is on the topic of recursion. When introducing beginners to other paradigms, simpler forms of iteration such as loops can be presented first, but that is not an option in a purely functional language, as these control structures aren't part of the paradigm [9].
These issues are however not regarded as important because the level of mathematics that Buff requires, does not call either for recursion, lambda or high order functions. These concepts will therefore only be explored by the programmer if greater interest of the programming capabilities is in place. It should be noted that using recursion would help enforce the principle of abstraction given as an important principle to follow in section 2.3.1.

2.6.1 Haskell

Haskell is one of the biggest pure programming language and is being taught at a few universities in introductory courses to programming [11]. A study from 2015 was made to investigate the types of errors new programmers makes when they are taught Haskell [12]. Haskell is one of the most popular purely functional languages. The study identified various issues related to the flexible nature of the Haskell syntax. In some cases, the study recommends sticking to one specific Haskell syntax to remedy the issues.
One such example is on the topic of giving one or more arguments to functions as parameters. Instead of comma separated arguments like what is used in C, Haskell allows multiple arguments to be separated by white space (code snippet 3).

```
-- Single tuple with multible arguments
sortBy(comparing(length), group(xs))

-- Space seperated arguments
sortBy (comparing length) (group xs).
```

Code Snippet 3: Example given in mentioned study for different approaches for multi-parameter functions. The two statements are equivalent.

The programmer is also allowed to omit the parenthesis in favor of using white space if the parenthesis are unnecessary when executing a function call (code snippet 4).

```
-- Function call with arguments encapsulated in parentheses
(f a b)

-- Function call with no encapsulation of arguments
f a b
```

Code Snippet 4: Example of omitting parenthesis in function calls in Haskell. The two statements are equivalent.

The study found that the syntax on code snippet 3 caused trouble with precedence, parenthesis placement and telling the arguments apart. The study suggests that sticking to the the C-syntax would be preferable for beginner programmers. It is recognized that Haskell's flexibility is useful for advanced programmers, but some features should be avoided until the programmer is more familiar with the language. This further supports the findings from section 2.3 which stated that Haskell presented problems regarding its readability.

2.6.2 Functional programming as a beginner's paradigm

Whether or not the functional paradigm is the best paradigm for a beginner's language is difficult to give a straight answer to, as it is subject to very contradicting opinions. On one side the functional programming paradigm has some very enthusiastic users, who strongly believe that functional programming is the best way to get started with programming [13] [14]. As the interview with professor Kurt Nørmark uncovered, he would prefer to teach a functional language as the first language in the computer science curriculum that he is responsible for. There are programmers that believe that because functional languages achieve everything with just one construct, the function call, they allow for the highest amount of simplicity [15, p. 35]. Because of the high simplicity that functional languages offer, some language researches find them highly interesting, and consider them to be a strong alternative to languages like Java that are not functional[15, p. 35]. Another educational institution who believes that functional programming has a place as the first paradigm is the University of Oxford. Here the MSc degree in Computer Science has a course called Functional Programming that takes place in the first semester and teaches Haskell. In the course description the University presents a benefit of teaching Haskell. Here they state that programs in Haskell are similar to mathematical functions. This gives the language a lot of strength, allowing the students to write programs that would be more challenging or cumbersome in other languages[16]. This claim is similar to the findings from section 2.5[9] which stated that a beginner course should focus on teaching elemental programming concepts and techniques, which they considered to be better taught in a pure functional language.

It has proven difficult to find solid arguments against learning a functional language as a first language. However, more frequently, viewpoints are presented that argue for other paradigms as being the best beginner paradigms. This gives the impression that there is not necessarily anything wrong with starting with a functional language, but there are simply other options that might suit beginners better. Most literature that considers what programming languages to learn first is found on IT blogs and forums which are subject to a high level of personal opinion. The standpoints that we have encountered, that do argue against functional programming, originate from different internet forums like stackoverflow.com and quora.com as well as the course material for the SPO (Languages and Compilers) semester course. Here some of the key points in the argumentation have been the very small popularity and real world use that functional programming has [17] and the claim that the human mind's natural way of thinking leans more towards the imperative paradigm [18] [15, p. 43]. It is worthwhile to take the viewpoints from the forums into consideration as they have been presented by programmers and IT-proficient people who have put thought and previous experience into their argumentation. However, as almost anyone is able to share their opinion on these internet forums, the arguments lack the amount of ethos that the view point of respected universities have.
In the end, it can be said that there is not necessarily a clear answer to, where the functional programming paradigm should or should not be the standard first language choice. There are many who think that other programming paradigms like the object oriented- and the imperative paradigm are the best choices for beginners. Yet, there are also some strong arguments presented by university professors, that point toward the functional programming paradigm as a very good candidate for a beginner's language.

2.7 Summary

From section 2.3 some predictors of success in the learning of programming were introduced with previous knowledge of *mathematics*, *ability to read* and *ability of logical thinking and problem-solving* being the three predictors this project will focus on. These predictors were further supported by the findings from the interview with Kurt Nørmark. Additionally the meeting also revealed some principles one should follow to become a better developer. All together the results of the interview and the findings from different studies pointed towards functional programming as being beneficial when teaching programming to beginners.
Afterwards, section 2.5 further discussed why functional programming could be a good choice for a beginner language introducing three principles a beginner course should follow. All of them were included in the functional paradigm further supporting the decision of making a functional language.

Because of the difficulties that were identified in section 2.2, specifically the ones that relate to conceptual knowledge regarding math and natural language and the issues identified with teaching purely functional languages and Haskell to beginners in section 2.6, we find it interesting to explore this topic further. With the previously mentioned predictors of success in mind, designing a new functional language that could address some of these pitfalls that are present in existing purely functional languages might make it easier to learn programming.

Some of these pitfalls include previous math knowledge and natural language resulting in a wrong understanding of how code executes, high-order functions and the concept of recursion. Studying Haskell, it was found that some features should be avoided in the beginning, and following a C like syntax could in some cases be preferable. The analysis also shortly examined the concept of purity. Lastly, it was proposed that not enough strong arguments could be found, that argue against learning a functional language to begin with, in order to discourage the development of a functional beginner language. Based upon the advantages of teaching a purely functional language, discussed in section 2.5, combined with the predictors we want to focus on, creating a purely functional language seems like a good fit.

2.8 Problem statement

As a result of the problem analysis we have pinpointed a narrow problem regarding teaching programming to a beginner and being limited to using the currently available programming languages. This has led to the formulation of the following problem statement:

How can a functional language which resembles mathematical syntax and an appertaining compiler be implemented that has the potential to serve as the first programming language for beginners?

3 Language design

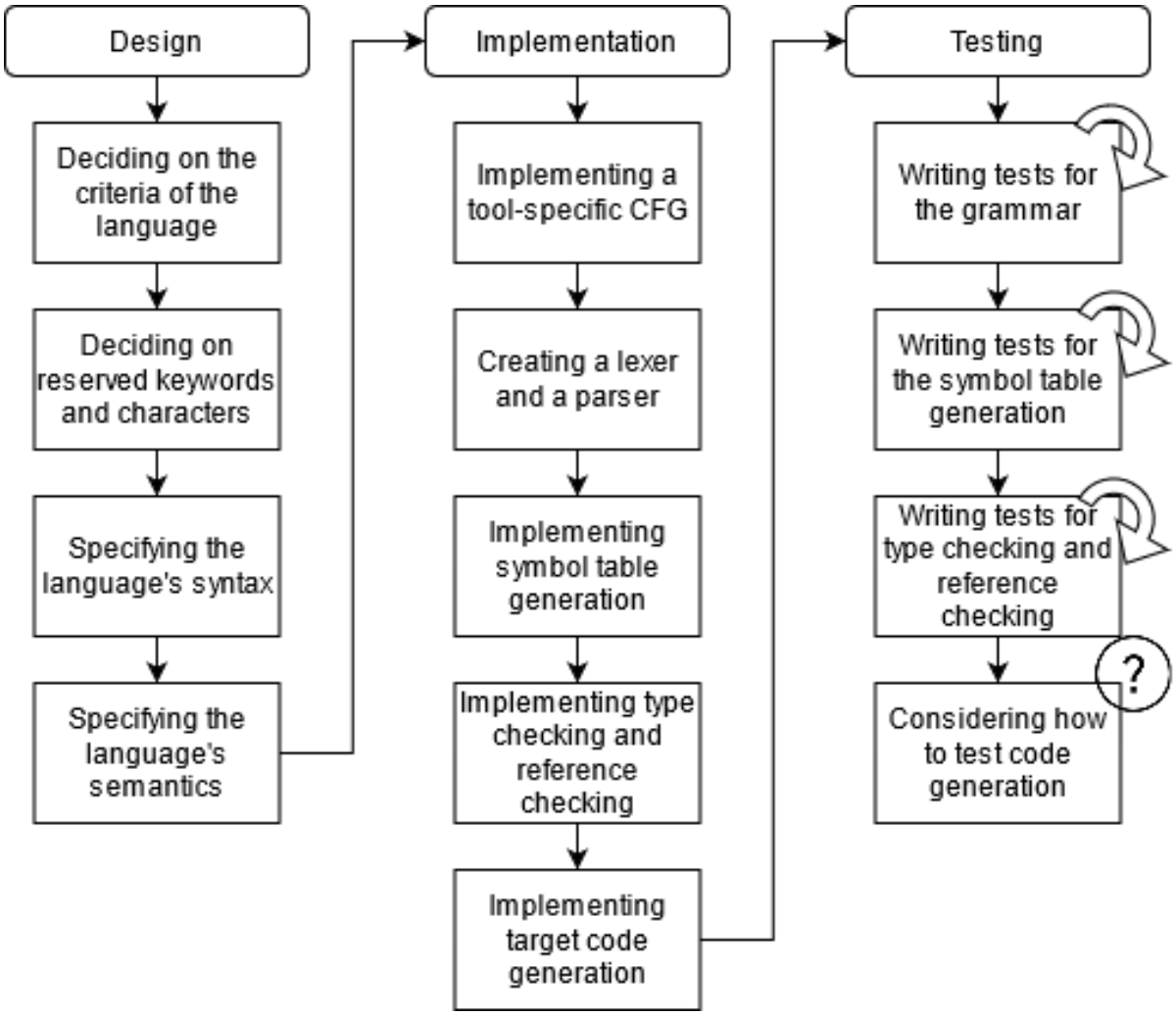


Figure 3.1: An overview of the problem solving process

This chapter is the first to describe the problem solution phase of the project, and figure 3.1 is included to give an overview of the process. The rotating arrows seen in the test column symbolize and iterative an ongoing workflow where additional tests are added at different times. Even though the figure shows a linear process, it must be said that previously completed steps were revisited when bugs emerged or when ideas for improvement came to mind. The last box in the testing column is marked with a small question mark because no optimal implementation for these tests was carried out. A potential method for testing the code generation is discussed in section 6.5.3

3.1 Purpose of Buff

After concluding the research and problem analysis for this project it has become clear to us that there are strong potential benefits associated with learning a functional language as the first programming language for a beginner because of characteristics such as the resemblance of high school mathematics, and the fact that it reinforces abstract thinking and it implicitly teaches good programming practices/principles as impure actions are not allowed,

However, current functional programming languages are not perfect. In particular we have evaluated Haskell and its syntax, where we found some aspects to be strange and difficult to understand. Our viewpoint on the syntax is also shared in the conclusions of research studies and programming books written about Haskell [12][11]. As a result we see potential for a new programming language that follows the functional programming paradigm and which syntax is intuitive as well as beginner friendly. It can be difficult to define what intuitive and beginner friendly means, and also hard to prove that those criteria are actually met. However, for this language we will try to accomplish this by making the language's syntax resemble mathematical syntax as close as possible, as well as to use reserved keywords which have actions that correspond with their wordings from the English language. The language will be purposeful for coding programs that perform mathematical computations, and as a result we will primarily focus on introducing arithmetic functionalities into the language. The main focus of the language is not to ease a beginner into the general world of programming by acting as a transition between high school mathematics and general purpose programming languages, but rather to ease the process of learning programming by resembling mathematics. To summarize the decisions so far regarding Buff a list is presented with the requirements found throughout chapter 2:

- Replace confusing characters such as "&&", "==" and "||" with words that represents their meaning or mathematical characters if such character exist (section 2.3 and subsection 2.6.1)
- Buff will follow the functional paradigm (section 2.3.1)
- Buff will allow for some impurity to support meaningful operations like printing (section 2.4)
- Recursion will be possible, but not expected to be understood by a beginner (section 2.6)
- The goal of Buff is to ease the process of learning programming by resembling mathematics (section 2.3)
- Buff will attempt to support the three principles an introductory course should aim for (section 2.5)

3.2 Criteria

To define and develop a language it is necessary to determine what criterion is of great importance and what is of lesser importance. To examine and determine that, this section uses Sebesta's book as a guide [15, p. 30-41]. Sebesta uses a table to indicate what factors have an impact on set criteria. The table is presented in figure 3.2.

Beyond the criterion seen in the table Sebesta also discusses monetary *cost* as a criteria, however as this project is developed by students, cost is not a factor to consider and it will therefore not be discussed further.

The criterion important to this project are readability and writability. Readability is an important factor as Buff aims to be as simple as possible to understand if one already knows mathematics to a beginner's level. Writability is important as this simplifies the process of learning the language. Lastly the criteria Simplicity is regarded as the most important feature in Buffas the target group is beginners who have never written code before.

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

Figure 3.2: Criteria and their characteristics [15, p. 31]

3.2.1 Readability

Readability is a criteria which has gotten more attention over time. Efficiency previously relied more on software than hardware because hardware was inefficient and expensive, but as hardware has become exponentially better and cheaper and developers have become increasingly more expensive the importance of readability has changed to be one of the determining factors of a good language with one reason being that the developer who starts a project is rarely one of the developers who finishes it. It is beneficial that new developers spend as little time as possible familiarizing themselves with the code. From table figure 3.2 it is evident that the four factors effecting readability are *simplicity*, *orthogonality*, *data types* and *syntax design*,

all of which will be discussed in the following subsections

Simplicity

The simplicity of a program revolves around the amount of constructs in a language, the feature multiplicities and operator overloading. The first aspect, the amount of constructs, suggests that too many language constructs decreases the simplicity and learnability of a language hence a program for beginners should contain few language constructs. Feature multiplicity means that there exist multiple ways to perform the same action. An example of this could be incrementing a variable by 1 which can be performed in four different ways in Java. Lastly the operator overloading suggests that having different meanings for an operator depending on the context leads to less simplicity. An example would be to add two numbers and adding two strings in C#. Both are done by the plus sign, but have different outcomes. From this discussion it is chosen that Buff should only have two data types, one being `number` to represent the types; integer, float, double and char. `number` will be interpreted as a 64-bit floating point number by the compiler. The second is `boolean` which will encapsulate the boolean characters `true` and `false`. Regarding operator overloading it is chosen that the `+` operator can only be used to sum `numbers`. Lastly, using the `+` operator on a `number` and a `boolean` will cause a syntax-error. This will be further elaborated upon and formalized in section 3.4.4.

Data types

The data types impact the simplicity of the language. More data types will lead to higher complexity, but also more control. Fewer data types will lead to greater simplicity, but less control. As Buff targets beginners greater simplicity is favored, hence fewer data types are implemented. The data types available in Buff can be seen in table 3.1.

Orthogonality

Another aspect of readability is orthogonality which "*means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language*" [15, p. 33]. This means that one should be able to combine the few data structures in a given language in such a way that other desirable data structures such as structs or classes is achieved. The reason why one should strive for this is that less orthogonality will lead to deviations in the syntax of the language which might confuse the developer.

Syntax design

The design of the syntax greatly affects the readability and simplicity of the program. Therefore it is important to determine for what purpose and in what context the language will be

used, as a language build for an experienced developer might favor complex syntax and constructs that in return gives more capabilities to the developer. For Buff the purpose is to create a language that represents mathematics in the closest way possible. Furthermore, the users will be beginners, therefore, both readability and simplicity are key features. Another important remark is the fact that Buff does not strive towards being an introductory programming language that eases the journey towards more complex concepts and languages, but rather aims to be a language resembling mathematics and thereby easing the learning process. From this it is chosen that keywords need not represent or closely relate to those of other languages such as C, but instead focus on resembling the mathematical syntax.

3.2.2 Writability

The writability of a program is closely related to the readability as writing a program calls for the developer to reread code multiple times. Another thing to consider when discussing writability, is what the context of the target problem is, as different problems call for different methods.

Orthogonality

Continuing the discussion on orthogonality, having a larger amount of constructs and data types will lead to more precision in the language, but will in turn also lead to greater cause for misuse of constructs. Therefore, high orthogonality, meaning a low amount of constructs and data types, will lead to simplicity when writing the code.

Expressivity

Expressivity most commonly describes the convenience of operations in a programming language. An example would be writing `count++` instead of `count = count + 1`. Moreover shortcircuiting is another convenience feature which also describes the expressivity. This is not of greater concern in this project, as it was chosen that the expressivity of Buff is defined by the expressivity of the mathematical language as it strives for a close-to 1:1 correspondence.

3.2.3 Reliability

Lastly, Sebesta delves on the topic of reliability which means that a program "*is said to be reliable if it performs to its specifications under all conditions*" [15, p. 37]. Sebesta discusses 4 topics in the book: Type checking, Exception handling, aliasing and readability and writability. As exception handling is a rather extensive feature to implement, the project at hand will not concern itself with it therefore, it will not be further discussed.

Type checking

Type checking is of utmost importance in ensuring reliability. Type checking can either happen on compile-time or at run-time. Compile-time type checking happens during compilation and is used by a large number of languages such as C, C++ and Java. compile-type checking is also referred to as static type checking as the types have to be declared before compilation. The advantage of static type checking is that all code is checked before execution, meaning that errors are encountered before execution rather than during. The downside is that all code, also some that might not be used, will be checked. Causing more load to the compiler and therefore making it more inefficient. Furthermore, it is not be possible to check all values during compilation if e.g. a program fetches data from a database during run time. This calls for run time type checking also known as dynamic type checking. Dynamic type checking happens during the execution and is used in languages such as JavaScript and Python. Dynamic type checking exist in many statically typed language, as it solves some issues regarding static type checking. One of those issues is with inheritance. This issue arises as the static type checker is not able to determine, what specific type an inherited object is. For example if you have a class `A`, and you have a class `B` which is a subclass of `A`. If you then instantiate a new object `c` the following way: `A c = new B();`, the static type checker will not be able to tell, if object `c` is of type `B`, as it is instantiated as type `A` [19].
As Buff does not contain inheritance or other features that calls for dynamic type checking, it will only use static type checking.

Aliasing

Aliasing describes the concept of having distinct names that are used to access the same memory cell. As of now aliasing is accepted as a dangerous feature, but some constructs exist that exploit this feature. A pointer in C is one example, but since neither pointers nor any other type of aliasing will not be part of Buff aliasing is not further discussed.

Readability and Writability

Again delving on the topic of readability and writability, both are important characteristics of a reliable program, as if there is no natural way to express an algorithm, it becomes more prone to errors because of the unnatural approaches one must take.

3.2.4 Summary

From the criterion discussed, some choices regarding Buff have been made. One of them being only having two types; `number` and `boolean` for the sake of simplicity. It was also stated, that the expressivity of Buff was highly limited by the mathematical language. Lastly, it was determined that Buff should use static type checking as it will not support any functionality that calls for dynamic type checking.

3.3 Reserved words

From the previous discussion, a number of reserved words were found all of which are depicted in table 3.1. Some reserved words in table 3.1 have not been discussed earlier, as they were given by the syntax of mathematics. An example of this is the "-" operator.

Data types	Operators	Statements	Other
number	+ (add)	if	end
boolean	- (subtract)	return	true
	* (product)		false
	/ (division		
	^ (power of)		
	= (assignment)		
	> (greater than)		
	< (lesser than)		
	>= (greater than or equal)		
	<= (lesser than or equal)		
	IS (is equal)		
	NOT (logical NOT)		
	IS NOT (is not equal)		
	AND (logical AND)		
	OR (logical OR)		

Table 3.1: Reserved words

In table 3.1 two data types and an assignment operator are shown, which could make it seem like we contradict our statement from section 2.3.1 stating that beginners did not understand the concept of the assignment character '=' when learning C-programming. However, in Buff the '=' character is used in the same context as in mathematical functions. This is also the reason behind using the token "IS" instead of "==". An example of the usage of the '=' character can be seen on code snippet 5.

```
number add(number a, number b) = a + b;
```

Code Snippet 5: Usage of the '=' character in Buff

3.4 Language Specification

In this section, the specifications for Buff is presented in the form of a lexical grammar, context-free grammar and semantics. This will provide a clear definition of the syntax, the semantics and evaluation at execution.

3.4.1 Lexical grammar

The lexical grammar describes the syntax of the language regarding tokens. It does not concern itself with the order that the tokens are entered, but only whether or not a given input is lexically valid. Table 3.2 presents the tokens and their corresponding regular expression. A more in-depth description of the tokens will follow in table 3.3.

Tokens	Regular Expression
NUMBERTYPE	"number"
BOOLTYPE	"boolean"
NUMLITTERAL	$(0..9)^+ \mid (0..9)^+ . (0..9)^+$
BOOLLITTERAL	"true" | "false"
END	"end"
RETURN	"return"
IF	"if"
ID	[A-Za-z][A-Za-z_0-9]*
LPAREN	"("
RPAREN	")"
LOGOR	"OR"
LOGAND	"AND"
LOGEQ	"IS"
NEGATE	"NOT"
LOGNOTEQ	"IS NOT"
LOGLESS	"<"
LOGGREATER	">"
LOGLESSOREQ	"<="
LOGGREATEROREQ	">="
ASSIGN	"="
SEMICOLON	";"
PLUS	"+"
MINUS	"-"
MULTIPLY	"*"
DIVIDE	"/"
POW	"^"
COMMA	","
PRINTCHAR	"?"

Table 3.2: Definition of terminals

№	Token	Description
№ 1	ID	An identifier which the user of the programming language specifies. For example: "foo", "bar", "someName" etc. The identifiers are used to identify functions and their parameters. The rules are based on C's lexical grammar for identifiers
№ 2	LPAREN/ RPAREN	A left and right parenthesis. They are used to define and call functions and as for boundaries for conditions in if statements. Furthermore they can be used to control precedence in expressions
№ 3	ASSIGN	The assignment character. The assignment character is only used in the context of function declarations.
№ 4	RETURN	The return keyword, which is used to return a value from a function.
№ 5	END	Defines the end of a function definition's scope.
№ 6	NUMBERTYPE/ BOOLTYPE	Used to define the type of a literal as well as the return type of a function.
№ 7	COMMA	Specifies the comma character. It is used to separate arguments in function definitions and function calls.
№ 8	IF	Marks the start of an if statement.
№ 9	SEMICOLON	The semicolon character. It is used to mark the end of a statement.
№ 10	LOGOR/ LOGAND/ LOGEQ/ LOGNOTEQ/ LOGLESS/ LOGGREATER/ LOGLESSOREQ/ LOGGREATEROREQ	The set of logical operators in the language. They are used in the context of evaluating conditional statements.
№ 11	NEGATE	Logical NOT. Negates a boolean expression. Cannot be used on numbers.
№ 12	PLUS/ MINUS/ MULTIPLY/ DIVIDE/ POW	The set of arithmetic operators in the language. Used for mathematical calculations.
№ 13	PRINTCHAR	A token that specifies that the name of the function, the parameters given and the evaluated value from the function call must be printed.
№ 14	NUMLITERAL	The set of integer and floating point literals. For example "11", "12.3", "0", "-432" etc.
№ 15	BOOLLITERAL	The boolean literals "true" and "false".

Table 3.3: Token description

Interesting lexemes

The `<POW>` token is used as a way to perform the *power of* arithmetic operation. In many other programming languages this would be a function call to a function called `pow()`. We chose to include the `<POW>` token in Buff as the "`^`" lexeme, because this is a widely used representation of the *power of* operation in programs like Geogebra and Word's equation mode.

Example usage: `x^y`

The `<PRINTCHAR>` token is used to print output to the user's screen. Whether or not one should be able to print outputs was also widely discussed. On one hand mathematics provide no way of printing in an expression, but one could argue that writing out the equation and result on a paper or in a IDE is equivalent to printing i a programming language. Moreover even though mathematics do not suggest printing in an expression, it is highly favorable when programming for quick debugging purposes as code might be syntactically and semantically correct, but not solve the problem it set out to. Furthermore as Buff is a mathematical language it is desirable to print the result if it used as aid in an assignment or other meaningful context. It was therefore chosen that the user can print the input and output of a function call, by putting the `<PRINTCHAR>` token as a suffix to the function call. The lexeme was chosen to be a "`?`" as this would symbolize the reasoning: "*What happens here?*". Placing this token would produce an output printing the name of the function that is called, the input provided and the result of that function call with that specific input.

Example usage: `plus(1, 2)?`
Example output: "`plus(1, 2) => 3`"

The `<ASSIGN>` token is used as part of the function definition. Again, we want to stay true to the mathematical syntax. Therefore, we chose to use the `<ASSIGN>` token as the "`=`" lexeme to denote the beginning of a function definition's scope. This raises the issue of identifying where the function's scope ends if it contains multiple statements. To solve this problem, we introduced the `<END>` token, as a way of indicating this. If the function only contained a single statement the developer has the opportunity to write it as a one line function which highly correlates with the mathematical syntax for a function definition.

Example multiple statements usage:

```
number plusGreaterThan(number a, number b) =
    if (a > b)
        return a + b;
```

```
    return b;
end
```

Example one line function usage:

```
number plus(number a, number b) = a + b;
```

3.4.2 Context-free grammar

A context-free grammar (CFG) can be used to describe the syntax of a programming language. It is a finite and compact representation of a language [20], which is going to play a central role in the construction of our parser. We decided to build the first version of our CFG as a small subset of the full language to minimize the complexity of it. This would allow us to get experience with the whole implementation process in a shorter time, which we considered to be the best strategy for this project. The initial CFG can be found in appendix C.

The tokens described in section 3.4.1 will be used as terminals in the CFG, and is written in *ALL-CAPS*. The non-terminals will be written with the *camelCase* notation.

We will use the initial version of our CFG (grammar 1, appendix C) to choose our parsing strategy for our final language. The parsing strategy is discussed in section 4.1. After choosing the strategy by trying out the available tools, a parser, based on our final CFG (grammar 3.4), were constructed. It should be noted that the grammar in grammar 3.4 is *adaptive* LL(*) (ALL(*)) which is developed by Terence Parr and Sam Harwell [21, p. 2]. ALL(*) performs grammar analysis dynamically rather than statically and is described as follows: "*Because ALL(*) parsers have access to actual input sequences, they can always figure out how to recognize the sequences by appropriately weaving through the grammar*"[21, p. 2].

⟨*prog*⟩ → ⟨*code*⟩* ⟨*EOF*⟩

⟨*code*⟩ → ⟨*funcDef*⟩
| ⟨*stmt*⟩

⟨*funcDef*⟩ → ⟨*typeAndId*⟩ ⟨*LPAREN*⟩ ⟨*funcDefParams*⟩? ⟨*RPAREN*⟩ ⟨*ASSIGN*⟩ ⟨*stmts*⟩* ⟨*returnStmt*⟩ ⟨*END*⟩
| ⟨*typeAndId*⟩ ⟨*LPAREN*⟩ ⟨*funcDefParams*⟩? ⟨*RPAREN*⟩ ⟨*ASSIGN*⟩ ⟨*stmt*⟩

⟨*returnStmt*⟩ → ⟨*RETURN*⟩ ⟨*stmt*⟩

⟨*type*⟩ → ⟨*NUMTYPE*⟩
| ⟨*BOOLTYPE*⟩

⟨*funcDefParams*⟩ → ⟨*typeAndId*⟩ (⟨*COMMA*⟩ ⟨*typeAndId*⟩)*
⟨*typeAndId*⟩ → ⟨*type*⟩ ⟨*ID*⟩
⟨*stmts*⟩ → ⟨*IF*⟩ ⟨*LPAREN*⟩ ⟨*expr*⟩ ⟨*RPAREN*⟩ ⟨*returnStmt*⟩
⟨*stmt*⟩ → ⟨*expr*⟩ ⟨*SEMICOLON*⟩
⟨*expr*⟩ → ⟨*LPAREN*⟩ ⟨*expr*⟩ ⟨*RPAREN*⟩
| ⟨*funcCall*⟩
| ⟨*funcCall*⟩ ⟨*PRINTCHAR*⟩
| ⟨*NUMLITERAL*⟩
| ⟨*BOOLLITERAL*⟩
| ⟨*ID*⟩
| ⟨*NEGATE*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*POW*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*DIVIDE* | *MULTIPLY*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*PLUS* | *MINUS*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*LOGLESS* | *LOGGREATER* | *LOGLESSOREQ* | *LOGGREATEROREQUAL*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*LOGEQ* | *LOGNOTEQ*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*LOGAND*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*LOGOR*⟩ ⟨*expr*⟩
⟨*funcCall*⟩ → ⟨*ID*⟩ ⟨*LPAREN*⟩ ⟨*exprParams*⟩? ⟨*RPAREN*⟩
⟨*exprParams*⟩ → ⟨*expr*⟩ (⟨*COMMA*⟩ ⟨*expr*⟩)*

Grammar 3.4: Our final context-free grammar. The context-free grammaris written in ANTLR syntax.

ANTLR grammar vs. Extended Backus–Naur form

Grammar 3.4 does not follow the syntax of extended Backus-Naur form (EBNF), but rather the syntax defined by ANTLR. The reasoning behind is that using a syntax in the report that differs from that of the code will lead to confusion when comparing both. This of course leads to the drawback that if one only wants to read the report and the findings, the syntax will be confusing as it differs from that of EBNF. Therefore, the following sections shortly introduce the syntax of grammar 3.4.

In ANTLR the `"()"` groups a set of non-terminals and terminals, the `"*"` operator states that the previous rule or rule group must be present 0 or more times, the `"?"` operator states that the previous rule or rule group must be present 0 or 1 times and the `"+"` operator states that the previous rule or rule group must be present 1 or more times. The precedence in the grammar also differs from that of a normal EBNF, as all of the `<expr>` productions would be

given the same precedence in standard EBNF and other tools. However, ANTLR gives highest precedence to the topmost production, then the second and so forth. The precedence of the arithmetic operators will directly depict that of algebra precedence; PEMDAS (paranthesis, exponent, multiplication, division, addition, subtraction) [22]. As there are not any set and broadly agreed upon precedence between the logical operators and arithmetic operators, we have chosen to stick to the operator precedence known from the programming language C, as the precedence rules for C have been reviewed and reevaluated throughout its existence by the creator Dennis M. Ritchie [23]. The only change which according to Ritchie; "seems that it would have been preferable", is to move the comparison operators below the bitwise operators [23]. However, as bitwise operators are not included in our language design it seems reasonable to stick to the well defined and wide spread rule set of the C programming language.

The two versions of `<funcdef>`

In the `<funcdef>` production, it is worth noticing, that the grammar enforces a return statement if it contains multiple statements. Therefore, an alternative function definition was implemented, that would almost directly depict that of mathematics.

$\langle funcdef \rangle \rightarrow \langle type \rangle \langle ID \rangle \langle LPAREN \rangle \langle funcDefParams \rangle \langle RPAREN \rangle \langle ASSIGN \rangle \langle stmt \rangle$

In this type of function definition the result of the `<stmt>` production is equivalent to the `<RETURN> <stmt> <END>` part of grammar 3.4. The concise function definition grammar allows us to define a simple plus function like so:

```
number add(number x, number y) = x + y;
```

Compared to the more verbose version:

```
number add(number x, number y) =
    return x + y;
end
```

Exploitation of a Computing Machine

In the Languages and Compilers course Bent Thomsen states that to provide the programmer full power of a Computing Machine, we must provide the following constructs in our programming language [24]:

- Sequential operations
- Conditional selection

- Looping construct

In the final version of the CFG (grammar 3.4), we have all of these constructs included in the following ways:

- **Sequential operations:**
  1. **Assignment** is included only as a way to define the behavior of a function.
  2. **Sequencing** is included both by the `<SEMICOLON>` token and the `<COMMA>` token.
  3. **Blocks** are included only in the context of functions by `ASSIGN` and `<END>` which encapsulate the function body.
- **Conditional Selection** is included only as an if statement. We deliberately chose not to include the *else* language construct, as the only type of statement which would make sense to execute in the context of a conditional selection would be a return statement. The `return` statement will terminate the function anyways which eliminates the need for the *else* construct.
  You might still ask why we chose not to allow statements without `return` within an if sentence. In a purely functional programming language a function cannot have any side effects. Also, we have chosen not to include variables in our language. This means that if a statement was to be executed, the result of a statement would go into the void and not contribute to anything useful. Therefore, the only statement, which makes sense in the context of an if sentence would be a `return` statement.
- **Looping Construct** is included only as recursion as other looping constructs like *while*, *do-while* and *for* does not belong in a pure functional programming language.

As all of the constructs, which Thomsen has mentioned, are present in Buff, we can conclude that Buff gives the programmer full power of a Computing Machine.

Overall, we can conclude, that we have done our best to create a CFG, which has a restrictive yet readable syntax, as it mimics the syntax known from mathematics. Precedence levels are as intuitive as they can be, and the CFG gives the programmer full power of a Computing Machine.

3.4.3 Scope rules

When scope checking in general it is required for each function to remember the statement it contains. As Buffis statically scoped, it is also necessary to remember what other functions are accessible at its declaration and the expected parameters. The definition for the function environment for Buff can be seen below:

$$\textbf{EnvF} = \textbf{Fnames} \rightharpoonup \textbf{Stm} \times \textbf{EnvF} \times \textbf{Params}$$

The *params* mentioned above are defined as follows:

$$\textbf{Params} = T\ x_1, ..., T\ x_k \qquad \text{where } k \in \mathbb{Z}^+$$

The actual arguments given in a function call are saved in a variable environment. To describe this, the environment-store model is used, which is a pair of two mappings that describes a program state [25]: a variable environment env_V taken from the set:

$$\textbf{EnvV} = \textbf{Var} \cup \{\text{next}\} \rightharpoonup \textbf{Loc}$$

and a store *sto* taken from the set:

$$\textbf{Sto} = \textbf{Loc} \rightharpoonup \mathbb{Q} \cup tt \cup ff$$

Lastly, *Stm* is defined as follows:

$$\textbf{Stm} = stmts\ \texttt{return}\ stmt\ \texttt{end}$$

Functions in Buff support recursive calling of themselves. In table 3.5 [CALL_{BSS}] , env'_F is a function environment in which the content of $env_F f$ has been added. env'_F is then used when evaluating the statement body of f, thus allowing recursive calls. This can be seen in the final rule in table 3.5.

$$[\text{DEF}_{\text{BSS}}] \quad \frac{\langle funcDef,\ env_F[f \mapsto (Stmt,\ env_F,\ params)]\rangle \rightarrow_{funcDef}\ env'_F}{\langle T\ f(params) = Stm\ code,\ env_F\rangle \rightarrow_{funcDef}\ env'_F}$$
if $code = funcDef$

$$[\text{DEF-STMT}_{\text{BSS}}] \quad \langle code,\ env_F\rangle \rightarrow_{funcDef}\ env_F$$
if $code = stmt$

$$[\text{DEF-EMPTY}_{\text{BSS}}] \quad \langle code,\ env_F\rangle \rightarrow_{funcDef}\ env_F$$
if $code = \epsilon$

$$[\text{CALL}_{\text{BSS}}] \quad \frac{env_V[params' \mapsto l][next \mapsto l']\ env''_F \vdash \langle Stm,\ sto[l \mapsto \{v_1, ..., v_k\}]\rangle \rightarrow sto'}{env_V, env_F \vdash \langle f(params),\ sto\rangle \rightarrow sto'}$$
where $env_F f = (Stm,\ env'_F,\ params')$
and $env''_F = env'_F[f \rightarrow (S,\ env'_F,\ params')]$
and $env_V,\ sto \vdash params \rightarrow \{v_1, ..., v_k\},$ where $k \in \mathbb{Z}^+$
and $l = env_V next$
and $l' = new\ l$

Table 3.5: Sope rules

3.4.4 Type rules

In this section, the type rules for Buff is presented. Buff contains two different types: *Booleans* and *numbers*. Because Buff strives for low orthogonality, most operators can only operate on one of the types, with the exception being [EQUALS$_{\text{EXP}}$] and [NOT-EQUALS$_{\text{EXP}}$] . In these two cases, the symbol T is used to represent a generic type: $T \in \{numbers, boolean\}$. The type rules for expressions are separated into two tables, arithmetic expressions in table 3.6 and logical expressions in table 3.7.

$[\text{ADD}_{\text{EXP}}]$	$\dfrac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 + e_2 : \text{number}}$
$[\text{SUBS}_{\text{EXP}}]$	$\dfrac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 - e_2 : \text{number}}$
$[\text{MULT}_{\text{EXP}}]$	$\dfrac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 * e_2 : \text{number}}$
$[\text{DIV}_{\text{EXP}}]$	$\dfrac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 / e_2 : \text{number}}$
$[\text{POW}_{\text{EXP}}]$	$\dfrac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 \text{^} e_2 : \text{number}}$
$[\text{PARENTH}_{\text{EXP}}]$	$\dfrac{E \vdash e : T}{E \vdash (e) : T}$
$[\text{NUM}_{\text{EXP}}]$	$E \vdash n : \text{number}$

Table 3.6: Type rules for arithmetic expressions

$$[\text{EQUALS}_{\text{EXP}}] \quad \frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 \ IS \ e_2 : \text{boolean}}$$

$$[\text{NOT-EQUALS}_{\text{EXP}}] \quad \frac{E \vdash e_1 : T \quad E \vdash e_2 : T}{E \vdash e_1 \ IS \ NOT \ e_2 : \text{boolean}}$$

$$[\text{GREATER-THAN}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 > e_2 : \text{boolean}}$$

$$[\text{LESS-THAN}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 < e_2 : \text{boolean}}$$

$$[\text{GREATER-THAN-EQUALS}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 >= e_2 : \text{boolean}}$$

$$[\text{LESS-THAN-EQUALS}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 <= e_2 : \text{boolean}}$$

$$[\text{NEG}_{\text{EXP}}] \quad \frac{E \vdash e : \text{boolean}}{E \vdash \ NOT \ e : \text{boolean}}$$

$$[\text{AND}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \ AND \ e_2 : \text{boolean}}$$

$$[\text{OR}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \ OR \ e_2 : \text{boolean}}$$

Table 3.7: Type rules for logical expressions

Functions in Buff have an explicit return type. It is therefore a requirement that the body of a function returns a value of that type. This is made sure of in [FUNC_{DEF}] which can be seen in table 3.8, where e is checked to return T', which is the type given at the start of the declaration.

$$[\text{FUNC-1}_{\text{DEF}}] \quad \frac{E \vdash stmts : \text{ok} \quad E \vdash e : T' \quad E[f \mapsto (x_1, ..., x_k : T_1, ..., T_k \to \text{ok})] \vdash \text{FuncDef} : \text{ok}}{E \vdash T' \ f(T_1 \ x_1, ..., T_k \ x_k) \ = stmts \ \texttt{return} \ e; \ \texttt{end} : \text{ok}}$$

where $k \in \mathbb{Z}^+$

$$[\text{FUNC-2}_{\text{DEF}}] \quad \frac{E \vdash stmts : \text{ok} \quad E \vdash e : T}{E \vdash T \ f() \ = stmts \ \texttt{return} \ e; \ \texttt{end} : \text{ok}}$$

Table 3.8: Type rule for functions definitions

Last are the type rules for statements, which are pretty straight forward. Most interesting

is the rule for a function call, in which each given argument needs to be checked with the expected argument type of the function. As the amount of arguments provided can vary widely from function to function, this is represented as an amount of arguments from 1 to k. This can be seen in [$\text{CALL}_{\text{STMT}}$] at the bottom of table 3.9.

[IF_{STMT}]	$\dfrac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : T}{E \vdash \texttt{if}(e_1)\ \texttt{return}\ e_2 : \text{ok}}$	
[RET_{STMT}]	$\dfrac{E \vdash e : T}{E \vdash \texttt{return e} : \text{ok}}$	
[$\text{CALL-1}_{\text{STMT}}$]	$\dfrac{E \vdash e_1, ..., e_k : T_1, ..., T_k \quad E \vdash f : (x_1, ..., x_k : T_1, ..., T_k \rightarrow \text{ok})}{E \vdash f(e_1, ..., e_k) : T'}$	where $k \in \mathbb{Z}^+$
[$\text{CALL-2}_{\text{STMT}}$]	$E \vdash f() : T$	

Table 3.9: Type rules for statements

3.4.5 Operational Semantics

This section presents the formal definition of the operational semantics for Buff. The semantics have been split into multiple tables:
In table 3.10 the semantics for arithmetic expressions can be found. In table 3.11 and table 3.12 the semantics for logical expressions can be found. The semantics will probably seem straight forward, as they conform with semantics known from many other programming languages like C and JavaScript.

$[\text{PLUS}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1 \quad a_2 \to v_2}{a_1 + a_2 \to v}$	where $v = v_1 + v_2$
$[\text{MINUS}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1 \quad a_2 \to v_2}{a_1 - a_2 \to v}$	where $v = v_1 - v_2$
$[\text{MULT}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1 \quad a_2 \to v_2}{a_1 * a_2 \to v}$	where $v = v_1 \cdot v_2$
$[\text{DIV}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1 \quad a_2 \to v_2}{a_1 / a_2 \to v}$	where $v = \dfrac{v_1}{v_2}$
$[\text{POW}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1 \quad a_2 \to v_2}{a_1 \hat{} a_2 \to v}$	where $v = v_1^{v_2}$
$[\text{PARENTH}_{\text{BSS}}]$	$\dfrac{a_1 \to v_1}{(a_1) \to v_1}$	
$[\text{NUM}_{\text{BSS}}]$	$n \to v$	if $\mathbb{Q}[\![n]\!] = v$

Table 3.10: Big-step transition rules for arithmetic expressions

[EQUALS-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 \ IS \ a_2 \rightarrow_b tt}$	if $v_1 = v_2$
[EQUALS-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 \ IS \ a_2 \rightarrow_b ff}$	if $v_1 \neq v_2$
[NOT-EQUALS-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 \ IS \ NOT \ a_2 \rightarrow_b ff}$	if $v_1 = v_2$
[NOT-EQUALS-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 \ IS \ NOT \ a_2 \rightarrow_b tt}$	if $v_1 \neq v_2$
[GREATER-THAN-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 > a_2 \rightarrow_b tt}$	if $v_1 > v_2$
[GREATER-THAN-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 > a_2 \rightarrow_b ff}$	if $v_1 \ngtr v_2$
[LESS-THAN-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 < a_2 \rightarrow_b tt}$	if $v_1 < v_2$
[LESS-THAN-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 < a_2 \rightarrow_b ff}$	if $v_1 \nless v_2$
[GREATER-THAN-EQUALS-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 >= a_2 \rightarrow_b tt}$	if $v_1 \geq v_2$
[GREATER-THAN-EQUALS-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 >= a_2 \rightarrow_b ff}$	if $v_1 \ngeq v_2$
[LESS-THAN-EQUALS-1$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 <= a_2 \rightarrow_b tt}$	if $v_1 \leq v_2$
[LESS-THAN-EQUALS-2$_{\text{BSS}}$]	$\frac{a_1 \rightarrow_a v_1 \quad a_2 \rightarrow_a v_2}{a_1 <= a_2 \rightarrow_b ff}$	if $v_1 \nleq v_2$

Table 3.11: Big-step transition rules for logical expressions 1

$$[\text{NOT-1}_{\text{BSS}}] \quad \frac{b \rightarrow_b tt}{NOT\ b \rightarrow_b ff}$$

$$[\text{NOT-2}_{\text{BSS}}] \quad \frac{b \rightarrow_b ff}{NOT\ b \rightarrow_b tt}$$

$$[\text{AND-1}_{\text{BSS}}] \quad \frac{b_1 \rightarrow_b tt \qquad b_2 \rightarrow_b tt}{b_1\ AND\ b_2 \rightarrow_b tt}$$

$$[\text{AND-2}_{\text{BSS}}] \quad \frac{b_i \rightarrow_b ff}{b_1\ AND\ b_2 \rightarrow_b ff} \qquad i \in \{1,2\}$$

$$[\text{OR-1}_{\text{BSS}}] \quad \frac{b_i \rightarrow_b tt}{b_1\ OR\ b_2 \rightarrow_b tt} \qquad i \in \{1,2\}$$

$$[\text{OR-2}_{\text{BSS}}] \quad \frac{b_1 \rightarrow_b ff \qquad b_2 \rightarrow_b ff}{b_1\ OR\ b_2 \rightarrow_b ff}$$

Table 3.12: Big-step transition rules for logical expressions 2

3.5 Single Pass vs Multi Pass compiler

When designing a compiler, you have to decide early on whether it is going to be a single pass or a multi pass compiler. Both approaches have advantages and disadvantages which should be considered, as it affects the subsequent design choices. Therefore, the following sections discusses pros and cons of different approaches.

3.5.1 Single Pass Compilers

Single pass compilers do as their name implies, only run through the source code a single time. A defining feature of a single pass compiler, is that the parser drives the other phases. Code for the target language gets generated while the parser reads through the source code.

Advantages:
Using a single pass compiler makes code generation easy for languages designed to be parsed using a recursive decent parser. It also does not use as much memory as a multipass compiler, because an abstract syntax tree is not needed. Moreover it is faster than a multipass compiler

Disadvantages:
It becomes harder to differentiate the different phases, as they happen very close together in your code. Furthermore, it makes it syntactically impossible in the source code language to

use a function before its definition is declared in the file, which means that the language must either completely disallow this, or force a prototype-like syntax similar to what is used in C.

3.5.2 Multi Pass Compilers

Different from single pass compilers, multi pass compilers allow two or more parses through the code, gradually moving closer to generating the target code after each iteration. It separates the syntactic analysis, contextual analysis and code generation into their own pass, each time running on the output of the previous step. Today, most modern compilers are multi pass compilers.

Advantages:
In general, doing multiple passes through the code allows a more creative syntax. Using a multi pass compiler fixes the previously mentioned issue of not being able to call a function before it is defined. It also enables the code for the compiler to be put in groups and better organized, which can help during development especially in regards to optimization of code generation.

Disadvantages:
If you have a language that could easily be implemented by a single pass compiler, making a multi pass compiler could be unnecessarily complicated. Furthermore it uses more memory and is slower compared to a single pass compiler

In our project, we choose to make a multi pass compiler as memory and efficiency is not an issue. Moreover it is desirable to separate functionality between phases to increase readability and optimization possibilities.

3.6 Symbol table

A symbol table is a place where information about identifiers and variables is saved and retrieved when needed. It can be incorporated in a few of the compilation phases, but is most relevant during type checking, as it is in the symbol table that information about a variable's type is stored. As Buff does not have the type of variables you would find in imperative languages but instead functions and function parameters, does the word "variable" in this section refer to these instead.

3.6.1 Symbol table strategy

As our language is block-structured language with scopes, we need a way to differentiate which variables are available at a current scope. Two approaches to this are presented in

Crafting a Compiler [20], making a symbol table for each scope, or saving the variables for all scopes in a single table, both have advantages and disadvantages which will be discussed below.

Individual symbol tables for each scope

When using multiple symbol tables, the compiler needs to keep track of when a new scope is entered or exited, and open or close the symbol tables accordingly. *Crafting a Compiler* [20] recommends using a stack for this, as scopes are opened and closed in a last-in, first-out manner. The disadvantage of this approach comes from the fact that if you want to do a look up for a variable, you need to check multiple symbol tables, which is more timely compared to checking in a single table. How big of a problem this is, depends on the importance of speed in the given scenario.

Single symbol table for all scopes

When only using a single symbol table for all scopes, each symbol needs to save information about what scope they were declared in. As a benefit to this approach compared to using multiple tables, it is not needed to look through a chain of tables to check if a symbol is defined, which means that using a single table is often faster.

Common for both approaches, is that they both involve inserting and retrieving information based upon the name of the symbol (usually in the form of a string). This can also be done in a variety of different ways, some of which is presented below:

Unordered and ordered list

A list is one the the most simple solutions to this problem. Optimally, it would be using a list structure with dynamic sizing, so that symbols could be added and removed without worry. The drawback of this approach is speed, as it becomes inefficient with too many symbols. *Crafting a Compiler* [20] claims that an unordered list becomes impractically slow except for the smallest symbol tables, as it takes too long to find the entry for a given symbol. An ordered list fixes the issue of the inefficient retrieval and is able to find a symbol in $O(log(n))$ time, but requires that symbols are entered at the correct positions in the table, which also requires more time depending on the amount of symbols. This approach works best for if the total count of possible variables is known in advanced, which is not the case in Buff.

Binary search (balanced tree)

Binary search trees provide a good middle ground between insertion and retrieval speed, as they both have a time complexity of $O(log(n))$. But in *Crafting a Compiler* [20] it is also stated that the average case performance is not optimal, as programmers do not choose their symbol names randomly. Using a balanced tree can help avoid the worst case scenarios of a normal binary search tree, but might take a bit more time to maintain. One example of such tree is a red-black tree.

Hash tables

Hash tables is the most common choice for symbol tables, because of its performance. *Crafting a Compiler* [20] states that with a large enough table, good hash function and collision handling, insertion and retrieval can be done in constant time.

3.6.2 The choices for our language

A hash table is chosen as the mechanism for our symbol table, as it seems like the best choice out of the options above. This avoids having to deal with the low performances of lists, and having to maintain a balanced tree.

3.7 Summary

To summarize all the design decision made regarding Buff, this section presents a list of all requirements for the language. The following list extends the requirements presented in section 3.1.

- Buff will contain two data types: `number` and `boolean` (subsection 3.2.1).
- Keywords do not need to relate to the syntax of other languages, but must instead represent the syntax of mathematics as closely as possible (subsection 3.2.1).
- Arithmetic operators will not be allowed to operate on *booleans* vice versa, and such action will therefore result in an exception being thrown, with the exception of the EQUALS operation (subsection 3.4.4). Furthermore, binary operations between two different types will be illegal and result in an exception being thrown.
- Custom print statements will not be possible, however one can print the result of a function call by suffixing the call statement with the "`?`" character (subsection 3.4.2).
- Buff will only contain the *if-then* language construct and not the *else* language construct (subsection 3.4.2).

- The compiler will be a multi pass compiler (subsection 3.5).
- The symbol table will be implemented as a hash map (subsection 3.6.2).

4 Implementation

To build the compiler we will need to go through three phases: The syntax analysis phase, the contextual analysis phase and the code generation phase. The three phases and their connection are depicted on figure 4.1.

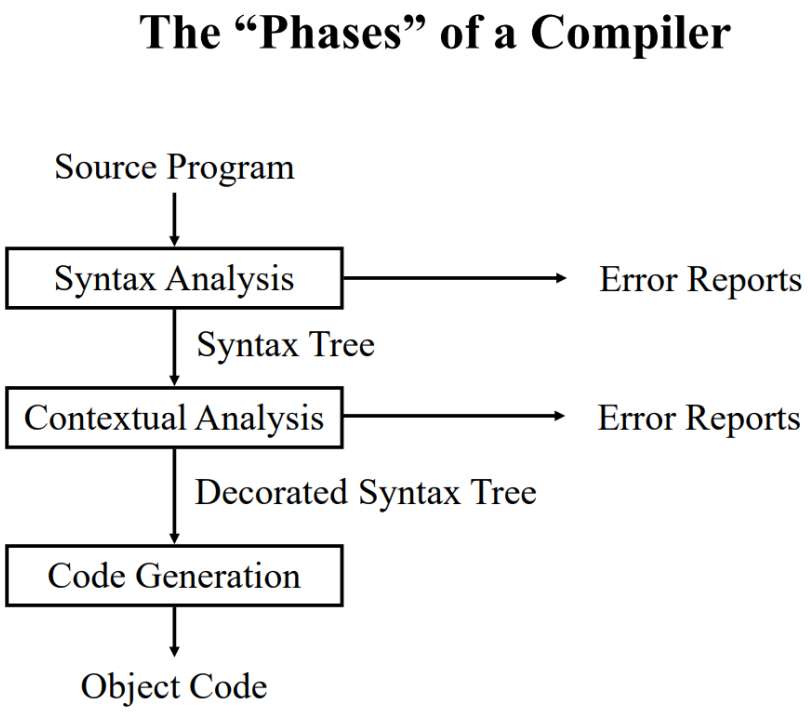


Figure 4.1: The three phases of a compiler.
Inspired by Bent Thomsen's slides [24].

On figure 4.2 a more detailed version of figure 4.1 is portrayed. In this version, the *Scanner* and *Parser* correspond to the *Syntax Analysis* phase on figure 4.1, the *Type Checker* corresponds to the *Contextual Analysis* phase on figure 4.1 and the *Translator*, *Optimizer* and *Code Generator* correspond to the *Code Generation* phase on figure 4.1.

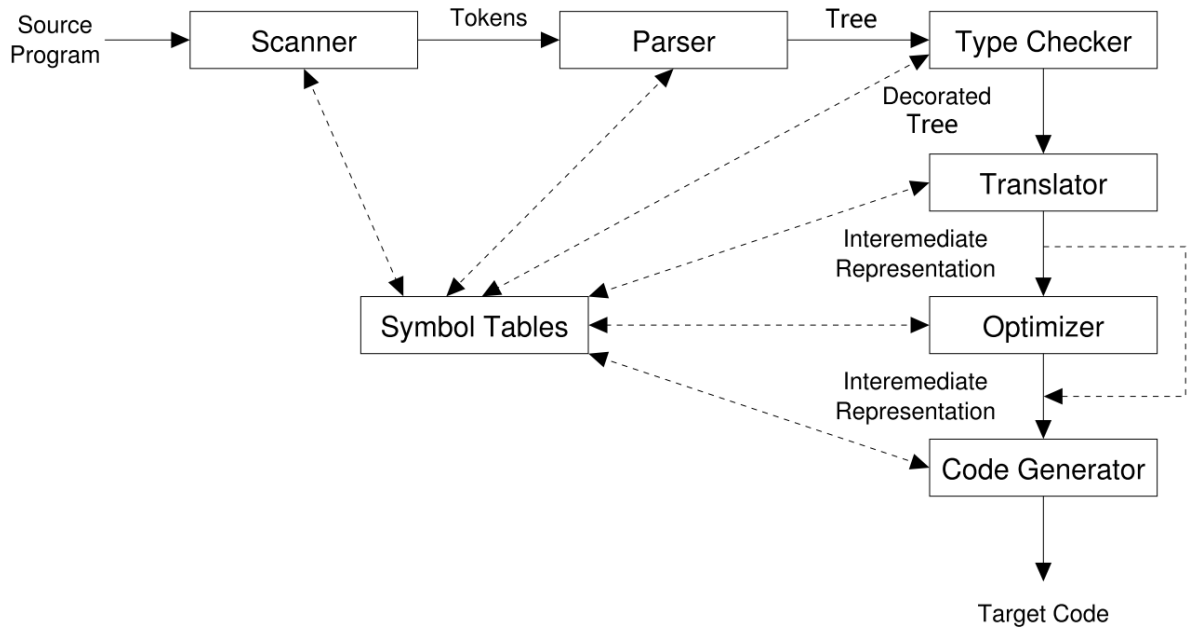


Figure 4.2: A syntax-directed compiler.
Inspired by figure 1.4 in Crafting a Compiler [20, p. 15].

What is interesting about figure 4.2 is the *Symbol Tables* box. As explained in section 3.6 a symbol table is a place where information about identifiers and variables is stored and retrieved when needed. Although it is true that the symbol table was accessed during all compilation phases in some older compilers, this is now being considered bad practice [24]. Instead, it is ideal only to access the symbol table in the *Type Checker* and *Translator* phases [24]. In our case, the symbol table is accessed during a reference check of the functions and parameters, and during type checking. In the caption on figure 4.2, it states that this is the structure of a syntax-directed compiler, meaning that the compilation follows the syntactical structure of the grammar. A syntax-directed compiler is the standard in modern compilers.

4.1 Syntax analysis

The purpose of the syntax analysis is to create a **lexer** (scanner) and a **parser** and thereby produce a **traversable tree structure** [24]. The lexer is responsible for recognizing the different tokens in the language and outputting a token stream. This is done by a series of regular expressions. Therefore, all lexers perform basically the same task, which means that if we decide to make a lexer by hand, it will result in reimplementation of components which all lexers share [20]. This is why a lexer generator can be beneficial.
The parser's task is to take the lexer's token stream as input, check if the syntax is correct and generate a traversable tree structure [20]. The tree strucuture can be either an abstract syntax tree (AST) or a concrete syntax tree (CST) (see figure 4.4b for difference). Parser generators also automate the task of generating code that can parse the tokens for a given context-free

grammar (CFG). Some parser generators even offer an all-in-one solution, meaning that they can both produce the lexer, parser and create the traversable tree automatically. The following section entails the decisions we have made in regards to syntax analysis for this project.

4.1.1 Lexing and parsing strategy

In the process of deciding which strategy to use, we tried out a couple of different approaches, to make sure we would choose the most optimal one for our language. We tried using JFlex/Java CUP, SableCC, ANTLR and making it by hand. We tested the different approaches with a minimal LL(1) version of our language as seen in appendix C. We experienced both the strengths and the weaknesses of each approach.

Making the parser by hand was a simple task, mainly because our language's grammar was LL(1). All we did was implement the pseudocode for the `match()` procedure and `TokenStream` class as described in *Crafting a Compiler* [20, pp. 40 + 149]. Even though this was a very simple process it did take quite a long time to write down all the code. The parser's complexity will rise if our language is not LL(1), because the implementation details will become more complex. Implementing the parser by hand will end up taking a lot of time and the process is more prone to errors compared to letting a tool do the work [20, p. 143].

Working with SableCC was a tiresome process. SableCC is an all-in-one tool, which allows us to create the lexer, parser and abstract syntax tree all at once [26]. However, we disliked the syntax, as we found it harder to read compared to Java CUP and ANTLR.

Making the lexer and parser with JFlex and Java CUP was a fairly easy task, as we could almost just copy/paste our context-free grammar into the .cup-file. JFlex and Java CUP worked together very nicely, and we experienced that the tools were able to provide us with very descriptive error messages, and that they were very intuitive to use. We found Java CUP provided us with a huge set of tools and thereby a lot of flexibility [27]. A major upside of Java CUP is the capability to create an abstract syntax tree from a little bit of action code, which eliminates the need for a lot of manual work. With this in mind Java CUP presents itself as a strong fit for the project, but was not chosen because ANTLR showed to be even more useful.

Working with ANTLR was an enjoyable experience. ANTLR is another tool that can generate a lexer, parser and a traversable syntrax tree [28]. ANTLR can run from command line or from the plugins for various integrated development environments (IDEs). A big benefit of using ANTLR is the live tree available, which is available if an IDE is used, that makes debugging fast, easy and comprehensible. An example of the live tree feature is shown in figure 4.3. ANTLR uses the CFG and token definitions to create the lexer, parser and visitor classes one can use to implement, symbol table, type checker etc.

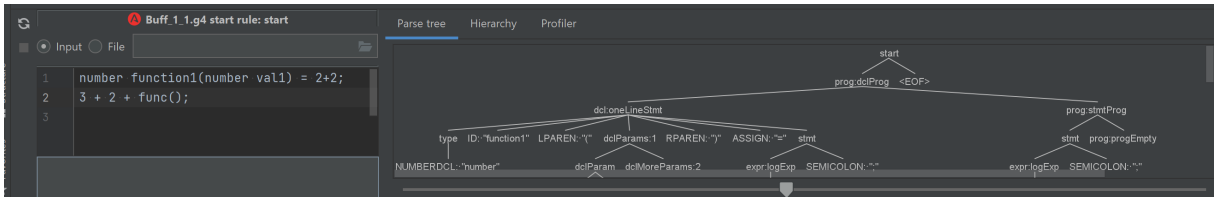


Figure 4.3: Live debugging of g4 file from ANTLR preview.

Another great advantage of ANTLR is that it can take all grammars - except ones that contain indirect left recursion (see definition box 4.1.1). This is the result of ANTLR's Honey Badger release [21, p. xii]. The acceptance of all grammars makes it easier to write readable grammars leading to less complexity which results in fewer mistakes [29].

Definition box 4.1.1: Indirect left recursion

Indirect left recursion means that a the production of a rule does not go directly to itself as the first element, but instead it does indirectly as seen in grammar 4.1.

$\langle Rule\ A \rangle \rightarrow \langle Rule\ B \rangle$ ...
$\langle Rule\ B \rangle \rightarrow \langle Rule\ A \rangle$ ...

Grammar 4.1: Left recursion.

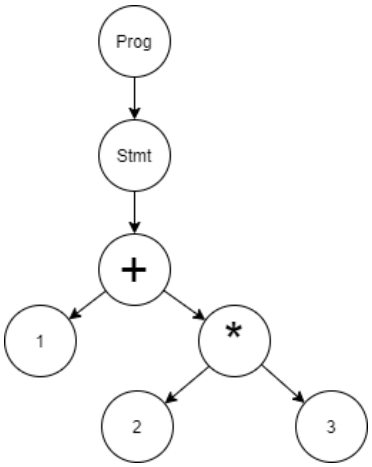
Additionally, ANTLR provides an alternative way to express precedence compared to other tools. Normally precedence is specified by how deep the rule lies within the concrete syntax tree (CST) which is defined by the CFG. This can create a lot of nested rules especially if there exists a number of precedence layers. Take C, which has 15 layers of precedence, as an example. This means that one would need to go through 14 productions, each representing another level of precedence, in order to reach a function call in the CST. ANTLR addresses this by allowing precedence specification through placement in the production list. Take for example the `math`, `term` and `followterm` from grammar 1. This is readable, but presents some complexity regarding readability. Scaling a simple situation where two precedence levels are present to 15 would lead to a lot of readablity issues. In ANTLR this can be written as seen in grammar 4.2 where `TIMES` has the highest precedence followed by `DIVIDE` etc. Beyond making the grammar more readable and concise, it also has the advantage of making the CST more concise. Actually the CST gets so concise that it raises the question of whether to use the CST instead of an abstract syntax tree (AST) as our tree structure. To exemplify this, figure 4.4 presents an AST and a CST on the expression $1 + 2 * 3$. Furthermore, it is not necessary to step from `expr` $\rightarrow$ `val` $\rightarrow$ `1` in figure 4.4b to get the value `1` as ANTLR implements a method called

`Node.getText()` which returns the textual value of the children, hence the only real gain from the AST would be skipping the steps from `codeStmt` to `expr:binaryOp` and the possibility of choosing yourself what information is required throughout the phases. Nevertheless with the capapility of using the already implemented visitors (see subsection 4.1.2 for explanation of the visitor pattern) and information contained in the CST, which is easy accessible, it becomes desirable to use the CST rather than having to implement a new visitor manually based on an manually implemented AST.

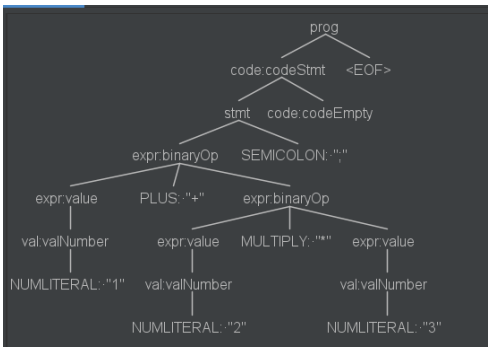
$$
\begin{aligned}
\langle mathExpr \rangle \rightarrow\ & \langle mathExpr \rangle\ \langle TIMES \rangle\ \langle mathExpr \rangle \\
\mid\ & \langle mathExpr \rangle\ \langle DIVIDE \rangle\ \langle mathExpr \rangle \\
\mid\ & \langle mathExpr \rangle\ \langle PLUS \rangle\ \langle mathExpr \rangle \\
\mid\ & \langle mathExpr \rangle\ \langle MINUS \rangle\ \langle mathExpr \rangle \\
\mid\ & \langle val \rangle
\end{aligned}
$$

Grammar 4.2: Precedence specified by position in context-free grammarin Antlr.

Because of the many and powerful features of ANTLR and the fact that it is a multi pass compiler, it is chosen for this project that the lexer, parser, symbol table and code generation will be implemented using this tool.



(a) AST representation of expression $1 + 2 * 3$



(b) Antlr CST representation of expression $1 + 2 * 3$

Figure 4.4: AST and CST representation of the expression $1 + 2 * 3$

4.1.2 The visitor pattern

A feature that plays a huge role in compilers [20] is the visitor pattern. ANTLR automates this pattern, which allows us to focus on the non trivial parts of the compiler such as type checking and code generation, ultimately allowing for a better end product. Therefore, this

section will describe the visitor pattern and why it is desirable.

The visitor pattern is an object oriented pattern that lets you separate functionality of a given object from that object [30]. This is done by creating a class usually called visitor. The visitor class will then implement a number of methods that carry out similar tasks that differ in what class object they act on. However, only implementing a visitor class raises a problem: The visitor will never know what method to call. This problem can be solved via double dispatch, however only few languages support that feature. As Java does not naturally include double dispatch we must solve the problem in another manner. It is possible to simulate the effect by using the visitor pattern. Therefore, each class, that the visitor visits, must implement an `accept()` method that takes a visitor as input and then calls the visitors `visit()` method giving its own object instance as an argument. This leads to the visitor calling the correct method on the object [30]. In the compiler for Buff the classes that implement the `accept()` method are the different nodes of the traversable tree. Additionally, individual visitors responsible for symbol table generation, id-reference checking, type checking and target code generation have been implemented. Because of this pattern, we are able to reuse the traversal method used to process the syntax tree for the four different visitors. Each visitor implements overridden visit methods for every type of node, and each of these methods carries out the accompanying steps for the specific node type.

Definition box 4.1.2: Double dispatch

The visitor pattern can simulate double dispatch in languages which only supports single dispatch. Single dispatch is the ability to select a method based on the dynamic type of the receiver [31]. Double dispatch enables choosing the correct method based on two things: The dynamic type of the receiver and the dynamic type of the method call's arguments [31].

Concrete example of the visitor pattern in the compiler

Following is a concrete example of how the visitor pattern is implemented in the compiler to obtain a double dispatch behavior. The example illustrates code generation for a `Func-call` which is a function call node. The following three code snippets originate from three different files: `JavaScriptCodeGenerationVisitor.java`, `AbstractParseTreeVisitor.class` and `BuffParser.java`. `BuffParser.java` is an ANTLR generated file specific to Buff and `AbstractParseTreeVisitor.class` is a part of the ANTLR library. Finally, `JavaScriptCode-GenerationVisitor.java` has been implemented by us and it extends `BuffBaseVisitor` which extends `AbstractParseTreeVisitor`. After the code snippets a small explanation of the code is included.

```
public class JavaScriptCodeGenerationVisitor extends BuffBaseVisitor<String> {
    ...

    @Override
    public String visitFunccall(FunccallContext ctx) {
        return ctx.ID().getText() + "(" + visit(ctx.exprparams()) + ")";
    }

    ...
```

Code Snippet 6: visitFunccall method in JavaScriptCodeGenerationVisitor

```
public abstract class AbstractParseTreeVisitor<T> implements
↪  ParseTreeVisitor<T> {
    ...

    public T visit(ParseTree tree) {
        return tree.accept(this);
    }

    ...
```

Code Snippet 7: visit method in AbstractParseTreeVisitor

```
public static class FunccallContext extends ParserRuleContext {
        ...

    @Override
    public <T> T accept(ParseTreeVisitor<? extends T> visitor) {
        if ( visitor instanceof BuffVisitor ) return ((BuffVisitor<? extends
        ↪  T>)visitor).visitFunccall(this);
        else return visitor.visitChildren(this);
    }
}
```

Code Snippet 8: accept() method in the FunccallContext class

The visitor pattern is expressed in the following way: `JavaScriptCodeGenerationVisitor` calls its `visit` method, which it inherits from the `AbstractParseTreeVisitor`. Additionally, the function `ctx.exprparams()`, which returns a `ParseTree`, is given as input parameter. `Visit` calls the node's `accept()` method, and provides itself as input. This way, the tree has knowledge of which type of visitor it has been given. Finally, the `accept` method calls the visitor's

appropriate method, which in this case is the `visitFunccall()` method, as the node is of the type `FunccallContext`.

4.2 Contextual analysis

After the tree has been generated by the parser in the syntax analysis phase, we can move on to the contextual analysis phase. In this phase, the generation of the symbol table will take place, together with its usage during *reference checking* and *type checking*.

4.2.1 Symbol table generation

As mentioned in the end of section 3.6, a hash table is used to keep track of the symbols in each scope. Furthermore, an individual symbol table for each scope is used. This way the IDs of parameters and functions can be reused as hash table keys across different scopes.
The symbol table is generated in a single traversal of the CST using a listener. The listener contains references to the current scope, the global scope and the `ParseTreeProperty` containing all scopes.

Scope
The primary functionality of a scope is keeping track of the symbols that are defined within it. This is done in a hash table mapping between a string and a Symbol object. It should also keep track of the scope that it itself is defined in, as Buff allows using functions defined in outer scopes. A scope should therefore recursively check if a symbol is defined in its enclosing scope, if it could not be found in its own hash table. Only when the global scope is reached and the symbol is still not found, should an error be thrown, which indicates that a reference to a non existing symbol has been made. The `Scope` interface can be seen in code snippet 9, and it specifies functions to get and set a scope's enclosing scope, as well as defining and getting symbols.

```
1  public interface Scope {
2
3      Scope getEnclosingScope();
4
5      void setEnclosingScope(Scope scope);
6
7      void defineSymbol(Symbol symbol);
8
9      Symbol getSymbol(String name);
10 }
```

Code Snippet 9: Scope interface

Symbol

A `Symbol` contains information about its name and its type. The type is represented as an integer, as this is the type, that ANTLR uses to represent the tokens in our grammar. The signature of the `Symbol` class can be seen in code snippet 10

```
public class Symbol {
    protected String name;
    protected Integer type;

    public Symbol(String name, Integer type);
    public String getName();
    public Integer getType();

    @Override
    public int hashCode();

    @Override
    public boolean equals(Object obj);
}
```

Code Snippet 10: Symbol class signature

FuncdefSymbol

A `Funcdefsymbol` extends the `Symbol` class, but also contains a list of types that match the types of a function's parameters. This allows for easier typechecking, when determining whether or not correct arguments are given to a function. The signature of the `FuncdefSymbol` class can be seen code snippet 11

```
public class FuncdefSymbol extends Symbol {
    private ArrayList<Integer> parameterTypes;

    public FuncdefSymbol(String name, Integer symbol, ArrayList<Integer>
     ↪  parameterTypes);

    public ArrayList<Integer> getParameterTypes();
}
```

Code Snippet 11: Symbol class signature

SymbolTableGeneratorListener

The `SymbolTableGeneratorListener` is a listener which extends the ANTLR generated `BuffBaseListener`, and is responsible for opening, closing and filling scopes with symbols. A

listener is used rather than a visitor, because traversing the entire tree is required.
A good example for showcasing all three responsibilities of the listener is when a function definition is visitied. The function itself should be saved in the current scope, but its parameters and body should be saved in a new scope of its own. When a function definition is entered, the function name gets defined in the current scope with a call to `currentScope.defineSymbol()`, and a new scope gets created afterwards for the function body with the current scope as a parameter. The new scope is then set to be the `currentScope` for further traversal of that function's branch of the tree. If a symbol already exists with the same id in the scope, an error is thrown as Buff does not support that. The code for this can be seen in code snippet 12.

```
public void enterFuncdef(FuncdefContext ctx) {

    ...

    FuncdefSymbol symbol = new FuncdefSymbol(ctx.ID().getText(),
    ↪  ctx.start.getType(), argumentList);
    try {
        currentScope.defineSymbol(symbol);
    } catch (Exception e){
        errorListener.ThrowError(e.getMessage(), ctx.ID().getSymbol());
    }

    // Making new scope for function body
    Scope newScope = new BaseScope(currentScope);
    attachScope(ctx, newScope);
    currentScope = newScope;
}
```

Code Snippet 12: Function called whenever a function definition node is entered

When the `SymbolTableGeneratorListener` has been through the entirety of the function definition, the current scope should be reset to the previous current scope. This is done by setting the current scope to be the enclosing scope of the current scope. This can be seen in code snippet 13

```
public void exitFuncdef(FuncdefContext ctx) {
    currentScope = currentScope.getEnclosingScope();
}
```

Code Snippet 13: Function called whenever a function definition node is exited

Figure 4.5 illustrates the resulting scopes from traversing a simple function definition, which takes two numbers and returns a number. As seen in the code from code snippet 12 the

function itself is defined in the outer scope, while its parameters are defined in a new inner scope.

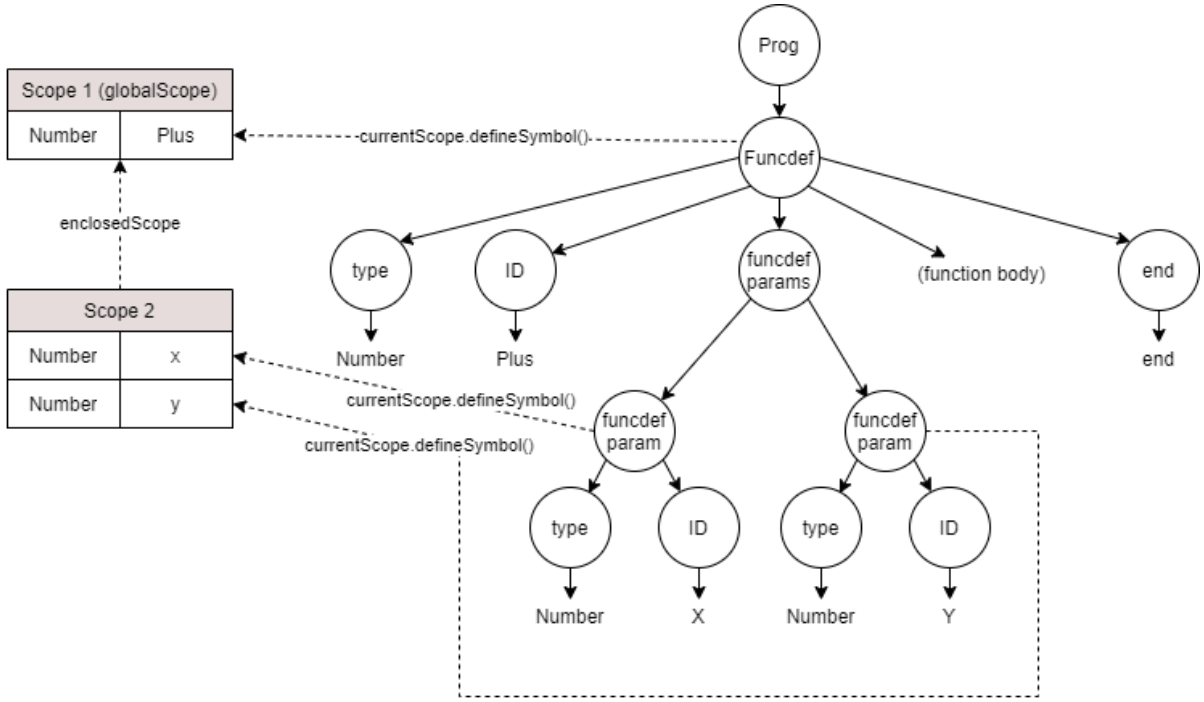


Figure 4.5: Illustration of scopes with symbols defined from a function definition

All that is missing now, is a connection between the tree nodes and the scopes. In future contextual analysis phases, it would make sense to have an easy way to get the scope for a function definition body, as an example. This is where the use of a `ParseTreeProperty` comes in. A `ParseTreeProperty` is simply a map between a node and something else, which ANTLR provides. In this case, it is a map between a node and the scope in which it is defined. In the `SymbolTableGeneratorListener`, this is done with a call to `attachScope` function, and can be seen in code snippet 14

```
1 public ParseTreeProperty<Scope> scopes = new ParseTreeProperty<>();
2
3 ...
4
5 void attachScope(ParserRuleContext ctx, Scope s) { scopes.put(ctx, s); }
```

Code Snippet 14: ParseTreeProperty in SymbolTableGeneratorListener

Figure 4.6 illustrates the link between nodes and scopes through a ParseTreeProperty. This approach allows us to get the scope for a node by using a call to `scopes.get(ctx)`, with `scopes` being the `ParseTreeProperty` and `ctx` being the node.

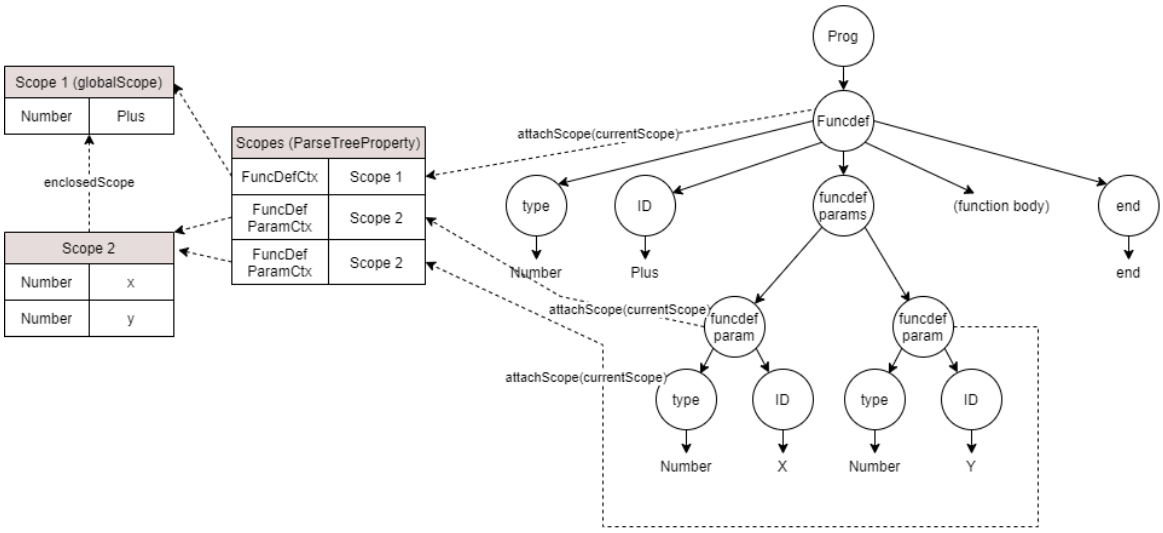


Figure 4.6: Illustration of nodes being mapped to scopes using a ParseTreeProperty

4.2.2 Type checking

Type checking is considered a vital part in the process of identifying programming errors [32]. In Buff we decided to have a static type system, which has resulted in the implementation of the following type checks:

1. The correspondence between the function definition's return type and the return statement(s) return type(s).
2. The expression in the if-statemetent's conditions should be of type boolean.
3. The type of the operators in an expression should have well defined behavior for each operand.
4. Correspondence between the type of the actual parameters in a function call and the type defined in the function definition.

Instead of looking at all of the different scenarios which we have type checked, we will display the most interesting ones.

A simple example of type checking is the second of the four type checking related demands: Checking for boolean expression in the if-statement's condition. The implementation details of this, can be found on code snippet 15. The grammar associated with code snippet 15 is:

$\langle stmts \rangle \rightarrow \langle IF \rangle \langle LPAREN \rangle \langle expr \rangle \langle RPAREN \rangle \langle RETURN \rangle \langle stmt \rangle$

```
public Integer visitStmts(StmtsContext ctx) {
    // Check if statement expression type
    Integer actualType = visit(ctx.expr());
    Integer expectedType = BOOLTYPE;

    if(!actualType.equals(expectedType)){
        throwTypeError(actualType, expectedType, "In an if statement",
        ↪  ctx.expr().start);
    }

    return visit(ctx.stmt());
}
```

Code Snippet 15: Type checking of if-conditional

Note that ANTLR generates all terminals as Java `Integer` types with a variable name written in all caps corresponding to the name of the terminal in the grammar. Therefore, the type checking visitor is based on Java's `Integer` type:

```
public class TypeCheckerVisitor extends BuffBaseVisitor<Integer> {...}
```

On code snippet 15 the `visitStmts()` method visits the expression node, to get the evaluated type of the expression. After that, it simply checks if the type is a boolean. If it is not we throw an error message for the user, if it is we return the type of the statement to be able to check the first of the four type checking related demands.

Type checking of expressions

The backbone of the type checking is the type checking of expressions. To ease the implementation of the type checking of expressions, we have formally defined the type rules of Buff in section 3.4.4. The rules can then be implemented in the type checker on the `exprBinaryOp` productions from the grammar. Having all binary operations confined under one function gives the possibility of switching on a terminal in the production, in our case the operator, to determine the correct action. Instead of implementing numerous functions cluttering the code. This is depicted in code snippet 16.

```
1  @Override
2  public Integer visitExprBinaryOp(ExprBinaryOpContext ctx) {
3      int returnType;
4
5      // Visit the children to thereby get their type.
6      Integer left = visit(ctx.left);
7      Integer right = visit(ctx.right);
8
9      String errorText = "On operation" + ctx.op.getText() + ". Must benumber
       ↪  type";
10
11     switch (ctx.op.getType()) {
12         case PLUS, MINUS, MULTIPLY, DIVIDE, POW -> {
13             if(left != NUMTYPE || right != NUMTYPE)
14                 throwTypeError(left, right, errorText, ctx.op);
15             returnType = NUMTYPE;
16         }
17         case LOGLESS, LOGGREATER, LOGLESSOREQ, LOGGREATEROREQ -> {
18             if (left != NUMTYPE || right != NUMTYPE)
19                 throwTypeError(left, right, errorText, ctx.op);
20             returnType = BOOLTYPE;
21         }
22         case LOGAND, LOGOR -> {
23             if(left != BOOLTYPE || right != BOOLTYPE)
24                 throwTypeError(left, right, errorText, ctx.op);
25             returnType = BOOLTYPE;
26         }
27         case LOGEQ, LOGNOTEQ -> {
28             if(!left.equals(right))
29                 throwTypeError(left, right, errorText, ctx.op);
30             returnType = BOOLTYPE;
31         }
32         default -> throw new IllegalArgumentException("Type not found
           ↪  bytypechecker.");
33     }
34     return returnType;
35 }
```

Code Snippet 16: Type checking of binaryOp

The interesting part is lines 11-32 as it ensures that the code complies with the rules from section 3.4.4. As an example take the rules in table 4.3 which cover the cases where if given two `number` types it should return a `number` type as stated by the ADD-rule. If given two `number` types it should return a `boolean` type as in the GREATER_THAN rules. And finally, if given two `boolean` types it should return a `boolean` type as in the AND rule. The rules are

respectively checked on line 12-15, 17-20 and 22-25. The final rule, `LOGEQ`, is not depicted in table 4.3 as the given type does not matter, as long as the `left` type is the same as the the `right` type.

$$[\text{ADD}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 + e_2 : \text{number}}$$

$$[\text{GREATER-THAN}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 > e_2 : \text{boolean}}$$

$$[\text{AND}_{\text{EXP}}] \quad \frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \ AND \ e_2 : \text{boolean}}$$

Table 4.3: The three interesting cases for the type checker

Type checking return statements

As the return statement of a function will always evaluate to some production of the `expr` rule, it is given from the previous section what underlying logic determines the type. As of this, checking the return types acquired from the statements with the actual return type given by the function definition is straightforward and is depicted in code snippet 17. The check must be conducted for both the `multiLineFunction` and the `singleLineFunction`, but as the logic is the same, only the code for the `singleLineFunction` is shown. The difference between the two lies within a loop that calls `checkReturnTypeCorrespondence` on all return statements in the `multiLineFunction`. The function first visits the `stmt` as this is the return statement in this context and sends it to the `checkReturnTypeCorrespondence` alongside the return type of the function definition given by `ctx.typeAndId()`. The other parameters are used for error messages and will not be discussed.

```
1  @Override
2  public Integer visitOneLineFunction(OneLineFunctionContext ctx) {
3      Integer returnStmtType = visit(ctx.stmt());
4      checkReturnTypeCorrespondence(returnStmtType, ctx.typeAndId(), ctx.stmt(),
       ↪  getReturnType(ctx.typeAndId()));
5      return returnStmtType;
6  }
```

Code Snippet 17: Type checking of `oneLineFunction`

Type checking formal vs actual parameters

Finally, to fully ensure that everything is type correct, the formal and actual parameters must be checked. This is done trough the `visitExprParams` function seen in code snippet 18. One might think that this is a step too deep in the CST as the node in hand is a child of a function-call, however ANTLR provides the `ctx.parent` property which gives the parent of the current node (line 12). Is should be mentioned that this logic could also be implemented in the `visitFuncCall` function with a few moderations. The logic of code snippet 18 is straightforward: First the expressions given by the actual parameters are retrieved (line 4). Hereafter the types of the expressions are acquired (line 8) and compared to the formal parameters (line 17-19) which are retrieved from the symbol table (line 12-14). As the logic concerned with parameter typechecking is all contained within this function, the `visitFuncCall` function need not to concern itself with the types of the parameters and therefore, the function returns the default result.

```
1  @Override
2  public Integer visitExprParams(ExprParamsContext ctx) {
3      //Gets lists of expression nodes in the actual parameters
4      List<ExprContext> params = ctx.getRuleContexts(ExprContext.class);
5
6      // Visits each expression node in the actual params,
7      // and thereby gets their types.
8      ArrayList<Integer> actualTypes = visitAndGetChildrenTypes(i
       ↪  ->visit(ctx.expr(i)), params.size());
9
10     // Retrieves the formal parameter's types
11     // from the function definition found in the symbol table.
12     FuncCallContext funccallContext = (FuncCallContext) ctx.parent;
13     FuncdefSymbol symbol =
       ↪  (FuncdefSymbol)globalScope.getSymbol(funccallContext.ID().getText());
14     List<Integer> formalParamTypes = symbol.getParameterTypes();
15
16     // Check that the types correspond to each other.
17     for (int i = 0; i < actualTypes.size(); i++) {
18         checkFormalVsActualParams(actualTypes.get(i),formalParamTypes.get(i),
           ↪  funccallContext, params, i);
19     }
20
21     return this.defaultResult(); // This is an arbitrary Integer as this value
       ↪  is not used
22 }
```

Code Snippet 18: Type checking of formal and actual parameters

4.3 Code generation

After the Buff source code input has been successfully processed in the contextual analysis phase it is time for generating and emitting the target code. As Buff will be a simple, nonoptimizing compiler, the *Code Generator* phase seen on figure 4.2 can be skipped [20]. Therefore, we will be building a code generating *Translator*, which is responsible of converting our concrete syntax tree directly to JavaScript code. Therefore, the terms "Translator" and "Code Generator" will be used interchangeably from now on.

The implementation of the code generation phase is mostly trivial for a couple of reasons. First of all, because of our choice of JavaScript as the target language for the compiler the syntax, semantics and scope rules are very similar and therefore, the complexity in translation is minimal. As a result, it is possible for the code generator to emit most of the source code directly as it is, and it will behave as valid JavaScript code. An exception to this, is the case where a Buff *PRINTCHAR* is included in the source code. As described in section 3.4.2 the "?" symbol in Buff will result in the name of the function, input parameters and return value of the function call to be written to the JavaScript console. The `PRINTCHAR` has multiple uses. One use is to see what the result of your function is if it is given certain parameters. Another use is to see if a recursive function like factorial evaluates to the desired result for each function call. Thereby, the user will hopefully gain a better understanding of recursion.
To accomplish this, both the input parameters and the result of the evaluated function had to be known. This presented a challenge, as the input parameters are known at the time of the function call, but the result is only known after completion. Therefore, when the code generator visits `ValFunccallPrint()` it generates an anonymous function, that wraps the function call and captures the result. The difference in the code generated between a normal function call and one that should be printed can be seen on code snippet 20 based on the Buff `factorial()` function seen in code snippet 19 as an example. An example of the output that the `factorialPrint` function creates can be seen on figure 4.7.

```
> factorialPrint(5)
  factorialPrint(1) => 1
  factorialPrint(2) => 2
  factorialPrint(3) => 6
  factorialPrint(4) => 24
< 120
```

Figure 4.7: Output in the terminal from the `factorialPrint function given the input 5`

```
// Function factorial without the "?" character
number factorial(number n) =
    if (n IS 0 OR n IS 1)
        return n;
    return n * factorial(n-1);
end

// Function factorial with the "?" character
number factorialPrint(number n) =
    if (n IS 0 OR n IS 1)
        return n;
    return n * factorial(n-1)?;
end
```

Code Snippet 19: Buff code for the `factorial()` and `factorialPrint()?` functions

```
// Function factorial without the "?" character
function factorial(n) {
    if(n == 0 || n == 1)
        return n;
    return n * factorial(n-1);
}

// Function factorial with the "?" character
function factorialPrint(n) {
    if(n == 0 || n == 1)
        return n;
    return n * (()=>{
        let res = factorialPrint(n-1);
        console.log(`factorialPrint(${n-1}) => ${res}`);
        return res;
    })();
}
```

Code Snippet 20: Generated code for `factorial()` and `factorialPrint()?`. The difference lies within the return statement calling function factorial(n-1)

The Java code responsible for this can be seen on code snippet 21. Line 3-4 on code snippet 21 is trivial as all that is done is fetching the parameters and name of a function and setting up the anonymous function. The statement on line 11 in code snippet 21 gets all the parameters and interpolates them in the `result` string.

```java
@Override
public String visitExprFunccallPrint(ExprFunccallPrintContext ctx) {
    String exprParams = visit(ctx.funcCall().exprParams());
    String result = initiatePrintFunction(ctx, exprParams);
    if (!exprParams.isEmpty()) {
        String[] exprParamsArray = exprParams.split(",");
        // Adds result of the parameters to the string
        result += getStringFromTokenList(i ->String.format("${%s},",
        ↪  exprParamsArray[i]), exprParamsArray.length - 1);
        // Adds result of the last parameter to the string
        result += String.format("${%s}", exprParamsArray[exprParamsArray.length
        ↪  - 1]);
    }
    result += terminatePrintFunction();
    return result;
}

private String initiatePrintFunction(ExprFunccallPrintContext ctx,String
↪  exprParams){
    String result = "(()=>{";
    result += String.format("let res = %s(%s);",GetFuncName(ctx.funcCall()),
    ↪  exprParams);
    result += String.format("console.log(`%s(",GetFuncName(ctx.funcCall()));
    return result;
}

private String terminatePrintFunction(){
    return ") => ${res}`); return res;})()";
}
```

Code Snippet 21: Code generation for PRINTCHAR

Another point that makes generating the target code a trivial task is our use of the visitor pattern which integrates perfectly with the file structure generated by ANTLR. The visitor pattern makes implementing this part of the compiler trivial because it works the exact same way as the type checking phase does by having a visitor visit the nodes throughout the CST. The fact that this is trivial should not be considered as a negative thing, instead it shows that the visitor pattern is a strong strategy and solution for a situation like this.

The rest of the process of the code generation can be described shortly in the following way: An instance of the `JavaScriptCodeGenerator` class starts the process by calling its `visit()` method with the root of the CST as input parameter. This kicks off the traversal of the CST, where the visitor visits all nodes. Each node will generate the target code corresponding to that node, and visit its children, to get their target code, if necessary.

It is worth noting that code generation might become less trivial with the introduction of a more complex grammar for Buff, which could become relevant if arrays and strings are included in the language. A change of target language to a language which scope and syntax structure is not as similar to that of JavaScript, would probably also lead to a more challenging code generation phase. With the scope and time restraints of this project in mind, we have chosen not to pursue those possibilities.

5 Testing

There are several different approaches to writing tests for a piece of software, and this chapter elaborates on the approach that we have chosen for testing the Buff compiler.

Before going into too much detail about our testing approach, we will be discussing our choice of testing framework (section 5.1) and our test setup (section 5.2).

5.1 Choice of testing framework

There are multiple testing frameworks for Java one of them being Junit. JUnit is nicely integrated into IntelliJ and works by simply including it through the IntelliJ package manager and will in our case be used for integration testing. Furthermore it supports dividing tests into test cases in .csv files. JUnit is widely used and we found the documentation to be easy to understand. JUnit also allows for parameterized testing, which is something we really wanted in our group, as it enables us to write more DRY tests (definition box 5.1.1). Even though JUnit is a unit-testing framework, it will also be used for the integration testing. This is because the features that JUnit provides are sufficient for the integration testing being done in this project, thus avoiding having to use two different frameworks.

Definition box 5.1.1: DRY [33]

DRY is an abbreviation of the words: "Do not repeat yourself". It refers to the principle in software development of reducing code repetition.

5.2 Test setup

To ensure consistency and readability, (in this regard meaning the ability to understand the purpose of a test via its naming), the tests will use a modification of the the Microsoft .NET naming convention which states that the name of a test method must contain the following [34]:

- The name of the method being tested
- The scenario under which it is being tested

- The expected behavior when the scenario is invoked

Furthermore the naming will contain the three requirements in the aforementioned order. Modifying it to our needs means extracting the component name of the method being tested and post fixing it with "*Tests*". If a function in the type checker is tested, the tests are categorized under "*TypeChecker*". Furthermore the name of the method being tested is replaced with the name of the rule being tested. This allows for multiple different inputs given to the same test function instead of creating arbitrarily many methods that only differ in their input and their name. JUnit uses .csv files to contain the inputs that are used in the tests. In figure 5.1 it is shown how the type checker tests are all confined under the category `TypeChecker` further containing cases that should pass and throw. From the innermost result in figure 5.1 the strength of this setup becomes apparent as it shows exactly under what input the `SimpleBooleanTypeInIfSentence` succeeds or fails.

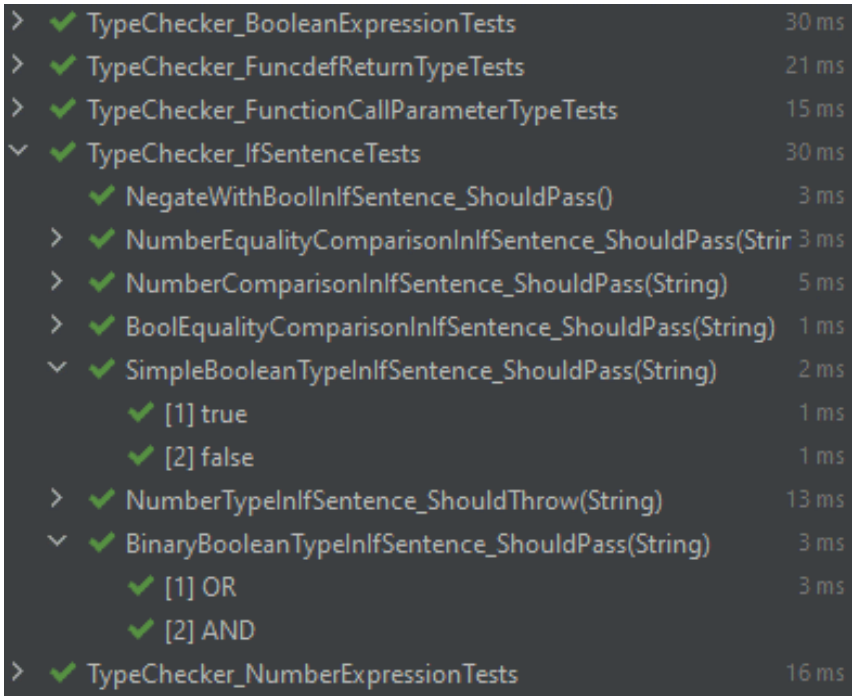


Figure 5.1: Test files structure

5.3 Unit testing

Unfortunately, one of the downsides of using the ANTLR tool is that it is almost impossible to implement unit tests. This is because of the difficulties related to mocking ANTLR tokens and parse trees, and the tight coupling of all components.

What can be unit tested, are the scope- and symbol classes that make up the symbol table, as they are not something that is provided or extended from ANTLR. These tests follow the

previously mentioned setup from section 5.2, and aim to test the code without interference from other parts. Two examples of this can be seen below in code snippet 22 where the `getEnclosingScope()` function is tested to correctly return its enclosing scope, and that `getSymbol("test")` returns the symbol with the name `"test"`.

```
@Test
public void enclosingScopeDefined_shouldBeAbleToRetrieve() {
    // Arrange
    Scope outerScope = new BaseScope();
    Scope innerScope = new BaseScope(outerScope);

    // Act
    var enclosingScope = innerScope.getEnclosingScope();
    // Assert
    Assertions.assertEquals(enclosingScope, outerScope);
}

@Test
public void symbolDefinedInScope_shouldBeAbleToRetrieve() {
    // Arrange
    Symbol symbol = new Symbol("test",1);
    Scope scope = new BaseScope();

    // Act
    scope.defineSymbol(symbol);

    // Assert
    Assertions.assertEquals(scope.getSymbol("test"), symbol);
}
```

Code Snippet 22: The implementation of visitExprBinaryOp

Unfortunately, this is the only part of our software that can be unit tested . However, if we assume that a mockup of an ANTLR concrete syntax tree (CST) is possible, then some unit tests for more parts of the compiler can be implemented. This is demonstrated in code snippet 23. The code tests `visitExprBinaryOp()` which implementation can be seen in code snippet 24. Ideally, we would implement six more tests to represent the last three cases of the switch case seen in `visitBinaryOp()`, two tests for each case: A test for ensuring that the function works without throwing an exception, given a valid expression input, and a test that ensures that an exception is thrown when the type checker encounters an invalid binary expression. We consider code snippet 23 to demonstrate this methodology clearly enough, and we therefore see no additional benefit of presenting the last six tests here.

```java
@ParameterizedTest
@CsvFileSource(resources = testPath + "plusMinusMultiplyDividePowNUMTYPE.csv")
public void typeCheckingExprBinaryOp_ArithmeticOperatorOnNUMTYPE_ShouldPass(String binaryOp) {
    //Arrange
    Integer leftChild = NUMTYPE;
    Integer rightChild = NUMTYPE;
    SymbolTableGeneratorListener symbolTable = new MockSymbolTable();
    TypeCheckerVisitor visitor = new TypeCheckerVisitor(symbolTable.globalScope, symbolTable.scopes);
    ExprBinaryOpContext ctxMockup = generateExprBinaryOpMockup(binaryOp, leftChild, rightChild);

    //Act
    Integer typeFoundByVisitor = visitor.visitExprBinaryOp(ctxMockup);

    //Assert
    Assertions.assertEquals(NUMTYPE, typeFoundByVisitor);
}

@ParameterizedTest
@CsvFileSource(resources = testPath + "plusMinusMultiplyDividePowBOOLTYPE.csv")
public void typeCheckingExprBinaryOp_ArithmeticOperatorOnBOOLTYPE_ShouldThrow(String binaryOp) {
    //Arrange
    Integer leftChild = NUMTYPE;
    Integer rightChild = NUMTYPE;
    SymbolTableGeneratorListener symbolTable = new MockSymbolTable();
    TypeCheckerVisitor visitor = new TypeCheckerVisitor(symbolTable.globalScope, symbolTable.scopes);
    ExprBinaryOpContext ctxMockup = generateExprBinaryOpMockup(binaryOp, leftChild, rightChild);

    //Act & Assert
    Assertions.assertThrows(RuntimeException.class, () -> visitor.visitExprBinaryOp(ctxMockup));
}
```

Code Snippet 23: A demonstration of a unit test for `visitExprBinaryOp()` assuming that a mockup of a parse tree is possible

```java
@Override
public Integer visitExprBinaryOp(ExprBinaryOpContext ctx) {
    int returnType;

    // Visit the children to thereby get their type.
    Integer left = visit(ctx.left);
    Integer right = visit(ctx.right);

    // Now we know that the two operators are of the same type: 'left == right'
    ↪ // true
    String errorText = "On operation" + ctx.op.getText() + ". Must be number
    ↪ type";
    switch (ctx.op.getType()) {
        case PLUS, MINUS, MULTIPLY, DIVIDE, POW -> {
            if(left != NUMTYPE || right != NUMTYPE)
                throwTypeError(left, right, errorText, ctx.op);
            returnType = NUMTYPE;
        }
        case LOGLESS, LOGGREATER, LOGLESSOREQ, LOGGREATEROREQ -> {
            if (left != NUMTYPE || right != NUMTYPE)
                throwTypeError(left, right, errorText, ctx.op);
            returnType = BOOLTYPE;
        }
        case LOGAND, LOGOR -> {
            if(left != BOOLTYPE || right != BOOLTYPE)
                    throwTypeError(left, right, errorText, ctx.op);
            returnType = BOOLTYPE;
        }
        case LOGEQ, LOGNOTEQ -> {
            if(!left.equals(right))
                throwTypeError(left, right, errorText, ctx.op);
            returnType = BOOLTYPE;
        }
        default -> throw new IllegalArgumentException("Type not found by
        ↪ typechecker.");
    }

    return returnType;
}
```

Code Snippet 24: The implementation of visitExprBinaryOp

As mentioned, the test on code snippet 23 assumes that a mockup of a `exprBinaryOp` can be created. In code snippet 23, the existence of the function `generateExprBinaryOpMockup()`, and its ability to create a representation of a valid ANTLR parse tree, is also assumed. This is an example of how we would have liked to unit test the parts of the compiler that we have

implemented ourselves, such as the `TypeCheckerVisitor`'s methods.

5.4 Integration testing

Integration testing revolves around testing if two or more components work together as intended. In our case an example would be testing if a given source program would result in the correct CST after it has been processed by both the lexer and the parser.
To ensure that all components work together a modular build starting from the lexer is used for testing. This process is depicted in figure 5.2

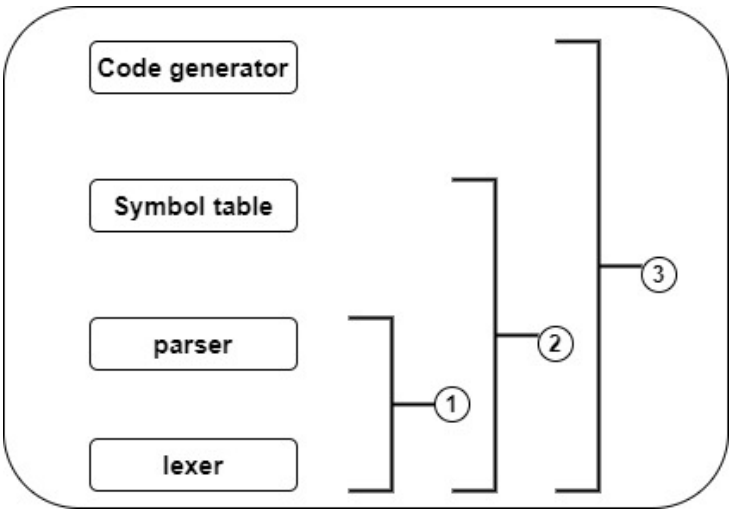


Figure 5.2: Integration test sequence

5.4.1 Lexer, parser and symbol table

Testing the integration between lexer, parser and symbol table was a minor task as it only had to test inputs where symbol table reading and writing is necessary. Therefore only the rules `IfFuncDef`, `OneLineFunction`, `Funccall` and `Funcdefparam` was tested. Code snippet 25 tests whether or not the return type of the function definition "*boolean func() = if (true) return true; return false; end* is boolean. On line 5 a call to the function `walker` is given a string from which it returns a symbol table. Hereafter, ANTLR's lexer is used to ensure the correct type notation is used for boolean. Lastly, the expected result is asserted against the actual result in line 10.

```java
@Test
public void funcDefWithBoolean_GivenBooleanType_ReturnsTrue() {
    // Arrange
    SymbolDefListener symbolTable = walker("boolean func() = if (true) return
    ↪  true; return false; end");
    // Act
    int expected = BuffLexer.BOOLTYPE;
    int actual = symbolTable.globalScope.getSymbol("func").getType().getVal();
    // Assert
    assertEquals(expected, actual);
}
```

Code Snippet 25: Lexer, parser and symboltable integration test

5.4.2 Parameterized tests

Code snippet 26 in combination with code snippet 27 shows an example of how test cases are implemented using CSV files. The test decorator `@ParameterizedTest` is JUnit specific and it specifies that the test should get its test variables from the location specified by the `resources` parameter. Parameterized tests have been implemented in order to adhere to the DRY coding principle (definition box 5.1.1), because they allow us to write the test once, and then automatically run it for each test case found in the comma-separated values (CSV) file, thus avoiding code repetition.

```
Type , Expression
"boolean","2"
"boolean","2.5"
"number","true"
"number","false"
```

Code Snippet 26: Comma-separated values file that includes test cases for testing conflicting data types

```java
@ParameterizedTest
@CsvFileSource(resources = testPath + "conflictingTypes.csv", numLinesToSkip = 1)
public void conflictingReturnTypes_ShouldThrow(String type, String expr) {
    // Arrange
    ParseTreeVisitor<Integer> visitor = generateVisitor(String.format("%s f() = return %s; end", type, expr));
    // Act & Assert
    Assertions.assertThrows(RuntimeException.class, () -> visitor.visit(tree));
}
```

Code Snippet 27: Test for conflicting data types

5.5 Acceptance testing

Acceptance testing is one of the last levels of testing, and is usually done after integration testing. The purpose of acceptance testing is to test and verify that the requirements for a product is met. This is usually done with respect to a user's or a business' requirements for a product, but can in theory be done for any type of requirements. In our case, the user would be a beginner writing programs in Buff, and running the resulting JavaScript code.

Acceptance tests are mostly defined by a "story" of how a user would interact with the product. As the product in this project is a language and compiler, these stories will be very similar, and all revolve around writing and compiling a program. In section 3.4, it is described that Buff should include and allow sequential operations, conditional selection and looping constructs to obtain full power of a computing machine. These three constructs will be the basis of our acceptance test, as a user should be able to write a program in Buff utilizing them in their program. An acceptance test has been written and run for each of these constructs which can be seen below.

In code snippet 28, the function `plus()` is called two sequential times with different arguments.

Sequential operations

```
number plus(number a, number b) = a + b;

plus(1, 2)?;
plus(-5, 6)?;
```

OUTPUT:

```
plus(1,2) => 3
plus(-5,6) => 1
```

Code Snippet 28: Acceptance test for sequential operations

in code snippet 29, the functions `isPositive()` is defined with conditional check of whether or not the given argument a positive or negative number.

Conditional selection

```
boolean isPositive(number x) =
    if(x >= 0)
        return true;
    return false;
    end

boolean isNegative(number x) = NOT isPositive(x);

isPositive(-10)?;
isPositive(10)?;
isNegative(-10)?;
isNegative(10)?;
```

OUTPUT:

```
isPositive(-10) => false
isPositive(10) => true
isNegative(-10) => true
isNegative(10) => false
```

Code Snippet 29: Acceptance test for conditional selection

In code snippet 30, recursion is used to get the the factorial of a number given as the argument.

Looping construct

```
number fact(number x) =
    if(x IS 0 OR x IS 1)
        return x;
    return fact(x - 1)? * x;
    end

fact(10)?;
```

OUTPUT:

```
fact(1) => 1
fact(2) => 2
fact(3) => 6
fact(4) => 24
fact(5) => 120
fact(6) => 720
fact(7) => 5040
fact(8) => 40320
fact(9) => 362880
fact(10) => 3628800
```

Code Snippet 30: Acceptance test for looping construct

6 Discussion

During the process of this project multiple topics of discussion have surfaced. In a broad sense, these topics include the choice of using third party tools, design decisions made for Buff, how Buff compares to other programming languages and what future possibilities exist for Buff. This chapter dives into discussions regarding these topics, and consider both advantages and disadvantages about the decisions that have been made during the project.

6.1 Using a tool for parts of the compiler creation

In this project we have chosen to use the parser generator called ANTLR to assist us in the process of making a compiler for Buff. As always when writing software it can be beneficial to lean on the work of others instead of reinventing the wheel. However, it is important to be cautious when doing so, as external frameworks and tools often come with limits, or force certain restraints upon the project. We think that it is important to consider the potential benefits and drawbacks of using a tool like ANTLR. Therefore, this section reflects on the impact that ANTLR has had on the project.

CST instead of AST

ANTLR is capable of automatically generating a concrete syntax tree (CST) in continuation of the parsing step without requiring a lot of manual work from the user. However, in the Languages and Compilers course we are advised to create an abstract syntax tree (AST). This is a lot more difficult to accomplish when using ANTLR, and it would require writing a lot of trivial and tedious code. In order to make an informed decision about which approach to take, we tried implementing both a CST and an AST. A clear disadvantage of the CST is the great amount of unnecessary information that accompanies each node, which requires deeper traversals in order to reach leaf nodes. An example of this could be in a plus node, which in a CST would contain a left and a right expression node which could each contain a value (figure 6.1). Only the values are necessary to evaluate in the following compilation steps, and their parent expression nodes are thereby unneeded. In such a situation it can become useful to use an AST, as the unnecessary nodes are removed. This can make working with the tree simpler as it avoids the need to traverse through many levels of child nodes.

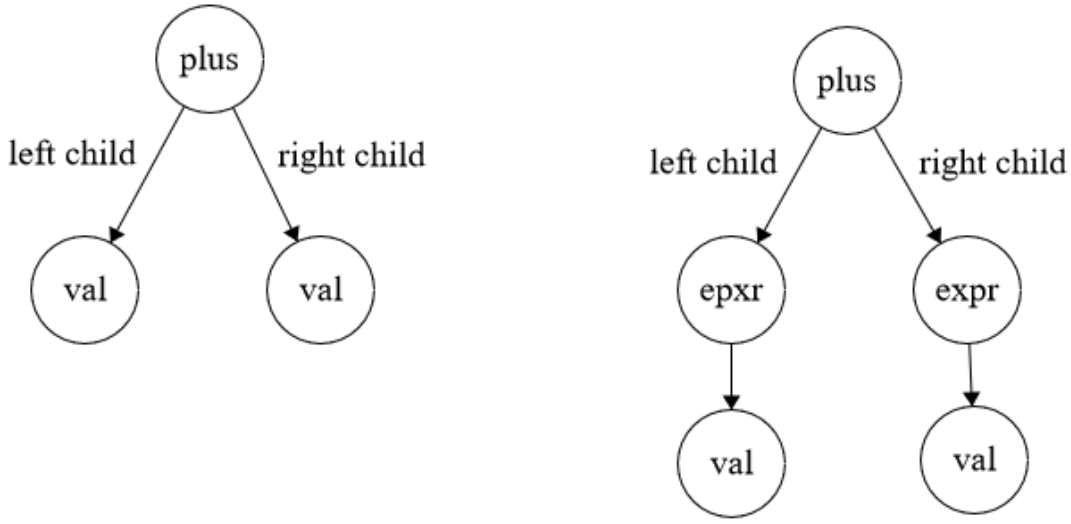


Figure 6.1: Example of plus node as an AST (left) and a CST (right)

Considering the reasoning about CSTs and ASTs and the observations made in subsection 4.1.1, an AST can seem like the obvious best choice. However, it is important to keep in mind that the AST does not construct itself. The only way to obtain an AST is to make an abstraction of the CST at some point, which will require traversing the many excess levels of nodes that the CST contains. However, when the AST has been obtained, all following work involving it will become easier to perform, but must be implemented manually. Therefore, the easier work process accompanies the drawback of tedious code writing. If only considering the complexity of the tree structure and not the implementation of visitors and other functionality the AST becomes less tedious to work with than the CST provided by ANTLR. One must also consider that when the context-free grammargrows in terms of productions, the complexity of the CST rises. For Buff it was possible for us to create a relatively simple CFG with few levels of nested production rules. Because of this, the CST that ANTLR creates is not that much more tedious to work with than the AST representation. Last but not least, ANTLR provides us with an implementation of the visitor pattern. Using that saves us a lot of time compared to implementing the visitor pattern from scratch for the AST.
With all these points taken into consideration, we found that the best solution for this project would be to use the CST provided by ANTLR. The simpler workflow that comes with an AST does not outweigh the tedious work that would be required to transform the CST to an AST in our case. Given another scenario where the differences between a CST and an AST are more prominent, it might very well be worthwhile for the development process to create the AST.

6.2 Language inspiration

As stated in section 3.1 Buff attempts to represent the mathematical language as closely as possible. Furthermore, it was discussed how syntax inspiration could be found in Python as it is known for its readable syntax. This assisted in design decisions later on, but as Buff strives towards a 1:1 correspondence with mathematics we should also have looked at programs like MatLab and Maple whose only purpose is to solve complicated mathematical operations. Therefore, both of these programs could provide inspiration for how to implement the syntax for Buff. It should be noted that both are operated inside an IDE and therefore present further capabilities for syntax by enabling constructs like the $\frac{numerator}{denumerator}$ which would not be possible in a language like Buff.

6.3 Comparison between Haskell, Java, Buff and mathematics

In order to elaborate on the strengths of Buff it is favorable to compare it to other programming languages as well as the mathematics that it tries to resemble as closely as possible. Following are some functions written in Buff, Haskell, Java and mathematics that will serve as a starting point for discussing Buff.
We compare Buff to **Haskell** because Haskell is a prominent functional languages [35]. Additionally, Haskell was included in the initial research, that was conducted during the problem analysis phase, as seen in section 2.6.1. It was found that the language's syntax caused trouble for beginners. **Java** is a language from the object-oriented programming paradigm, and therefore it has quite different features. When designing Buff, we have attempted to steer away from some of the syntactical aspects of Java, as we wanted to make Buff's syntax as recognizable as possible for beginners. Furthermore, Java is a very popular language that it is used by a lot of developers every day. Finally, **mathematical syntax** is also presented, as this is the syntax that Buff aims towards resembling.

6.3.1 Fibonacci function implementations

$$
\begin{aligned}
&Fib_0 = 0 \\
&Fib_1 = 1 \\
&Fib_n = Fib_{n-1} + Fib_{n-2} \qquad \text{for } n \geq 2
\end{aligned}
$$

Table 6.1: Fibonacci recurrence defined with mathematics

When looking at the different implementations of the Fibonacci numbers in table 6.1, code snippet 31, code snippet 32 and code snippet 33 it is apparent that the Java implementation

```haskell
fib x = if x <= 1 then x else fib (x - 1) + fib (x - 2)
```

Code Snippet 31: Fibonacci function implemented in Haskell

```java
public static int fib(int x){
    if (x <= 1){
        return x;
    } else {
        return fib(x - 1) + fib(x - 2);
    }
}
```

Code Snippet 32: Fibonacci function implemented in Java

(code snippet 32) is the most verbose of the three programming languages. Because of the object oriented nature of Java the function begins with an access modifier and must also be wrapped in a class which is left out in this case as it is unnecessary bloat. This is neither necessary nor possible in Buff as classes are not supported, which in turn allows for a simpler syntax.
The syntax rules of Buff omit the `else` keyword, which is exercised in both Haskell and Java. One might argue that the wording of the `else` keyword could aid in understanding Buff code when reading it (see section 6.4). In Buff an implicit effect of the `else` keyword takes place through the `if` and `return` structure. It would not make a difference for the semantic behavior of a Buff program if the `else` keyword was included in the grammar, however we have decided against this with the aim of higher mathematical similarity.

Next we take a look at a simple function that takes two parameters as input and returns their sum.

```
number fib(number x) =
    if (x <= 1)
        return x;
    return fib(x - 1) + fib(x - 2);
end
```

Code Snippet 33: Fibonacci function implemented in Buff

6.3.2 `add()` function implementation

$$add(x, y) = x + y$$

Table 6.2: `add()` defined with mathematics

```
add x y = x + y
```

Code Snippet 34: Function that returns the sum of two numbers implemented in Haskell

```
public static int add(int x, int y) {
    return x + y;
}
```

Code Snippet 35: Function that returns the sum of two numbers implemented in Java

```
number add(number x, number y) = x + y;
```

Code Snippet 36: Function that returns the sum of two numbers implemented in Buff

The `add()` functions shown in table 6.2, code snippet 34, code snippet 35 and code snippet 36 help to illustrate a syntax detail of Buff that results in a high level of mathematical similarity. Function declarations in Buff can be implemented as seen in the Fibonacci functions above with the use of the `return` and `end` keywords. However, it is also possible to declare functions on one line, and omitting the aforementioned keywords. This syntax form looks very much like a function from mathematics, with the only exceptions being the keyword `number` and the final semicolon. It can be argued that this syntax will allow a beginner to understand the meaning of the code almost immediately when coming from a high school math background. As seen in code snippet 34, Haskell also allows for a one-line style of syntax, and Buff has also partially found inspiration in Haskell. However, Haskell also allows the programmer to omit the parentheses around the formal parameters. As mentioned in section 2.6.1 Haskell has proven to be difficult for beginners to understand because of its high level of flexibility which includes the option to separate parameters with white-space characters. Since this approach in Haskell syntax does not follow the syntax of mathematics we do not consider it ideal for beginners, and instead we believe that the Buff syntax rules are preferable. On the other hand we have the Java example in code snippet 35, where it is not possible to omit the curly brackets that wrap the scope of the function, the access modifier and `return` keyword.

This rigidness of the language's syntax is good for the recognizability of the language. Remember that the lack of recognizability as a result of too much flexibility was one of Haskell's points of weakness (section 2.6.1). We consider Buff's level of flexibility to be low enough to ensure recognizability, yet it is still high enough to allow for short one-line functions similar to mathematical syntax, as well as block-style functions that can provide higher readability when function bodies become too long to fit on a single line.

6.4 The *if-then else* construct

In section 3.4.2 it was explained why Buff would only contain the *if-then* construct and not include any *else* construct as it was not needed, because no side effects are permitted. However, later discussions on the topic involved readability as it is one of the main criteria for Buff. Including the *else* or even the *else if* construct could lead to greater readability by supporting the predicting factor *ability to read* as the function in code snippet 37 could be read out in plain English without deeper knowledge of language constructs.

```
number fib(number n) =
    if (n IS 0)
        return 0;
    else if (n IS 1)
        return 1;
    else
        return fib(n-2) + fib(n-1);
end
```

Code Snippet 37: Buff `fibonacci()` function including the *else-if else* construct

Code snippet 37 shows another issue regarding consistency in the language design that follows from not using the *else* construct. This is because Buff enforces the use of a *return* statement inside an *if* construct, because it is the only meaningful statement to have in that context. The *else* construct is left out even though it might contribute to readability, but instead simplicity was favored. A perk of only including the *if* conditional construct and not the *else* is that it leads to less bloat in the code which can become quite cumbersome in some scenarios. This would result in a decrease in readability instead of an increase. Therefore, it depends on the complexity of the code, whether or not the *else* construct is favorable.

6.5 Testing methodology

Ideally we would like to achieve 100% coverage of all parts of the Buff compiler. Including unit tests of all functions and integration tests for all integrating components. However, in

reality this is rarely possible, and this project is no exception.

6.5.1 Consequences of choosing ANTLR

Using a third party tool like ANTLR has had a big influence on how we have been able to test the Buff compiler. This is because ANTLR generates a lot of the code for the compiler's backbone. ANTLR produces the lexer and parser based on the grammar that we provide. Because the lexer and parser are created by ANTLR we have chosen not to write individual tests for those two parts. This decision has been made because ANTLR is widely used, and therefore we trust that it has been tested thoroughly. Additionally, testing the components of the compiler that take action after the lexer would require mock-ups of the tokens and parse trees that are generated, which is not done as the parse tree structures are very complex ANTLR classes. Instantiating them depends on a large of amount of other ANTLR-generated files and logic. This makes it a very time demanding task to mock the parse tree.

6.5.2 Testing the context-free grammar

In order to make sure that the context-free grammar (CFG) works as intended, it would have been beneficial if we could have tested it directly. Our search for a tool or method that would allow us to implement such tests unfortunately came away empty handed. We found that the next best solution was to test the grammar rules via the parse tree generated by the parser. These tests have been conducted by writing several test cases for different possible Buff programs using the syntax as it was intended and designed, in addition to test cases that consisted of invalid Buff syntax. These tests helped us develop Buff as we have been able to check if the language could be used as intended, and if it would throw compile errors when invalid syntax was written. We are aware that it is a sub-optimal solution to test the grammar through a dependency to the ANTLR-generated parts of the compiler, yet it was better than nothing.

6.5.3 Testing the code generation phase

Last but not least, it would have been desirable to ensure that the code generation produces the right output code. Different approaches have been considered. Since the compiler compiles Buff to JavaScript by emitting the JavaScript code as a string, we initially created tests that compared the emitted string to our predetermined expectation of the generated code. Because these tests worked by making a direct comparison on strings it did not account for the semantics and behavior of the code in any way. This means that an introduction of an insignificant white space, that would have no impact on the actual run time behavior of the compiled program, would result in a failed test, regardless of the fact that the emitted program would work as intended. Additionally, testing in this way leads to a high cost in terms of maintainability, as the slightest change to the code generation would require corresponding

test cases to be modified to reflect the change, even though the change might only be of a syntactical nature that has no semantic effect on the program as in the case of optimization. Because of these considerations regarding direct comparisons of strings, as a way to test the code generation, we have decided against testing with such a method.

As an alternative, we considered that testing the code generation should rather test if the behavior of the emitted code was equal to the behavior of the source code. This way the test would disregard any syntactical difference that does not have an impact on the program's behavior. Testing with this method would require multiple phases. We imagine that the desired test method could be achieved by writing tests that assert if different Buff programs result in the expected output after they have been compiled and the target code has been executed. Because of the mathematical domain specific nature of Buff the behavior of all programs is to return a mathematical expression. Because of this, the assertion step of the test will be quite simple. However, the act step of the tests would be a bit more complicated. It would involve writing to and reading from files, because we would like to make the tests as automatic as possible. In order to capture the output of the target code program, we would have to write the emitted JavaScript code to a JavaScript file, execute the file with Node, save the output of program to another file and finally read that file from within the test to compare the content with the test's expected result. Because this process involves many steps and would require a lot of time to implement, we have decided not to prioritize it with the time limit of the project taken into consideration. An alternative solution that would help in the implementation of such tests was found later. The solution involved using a `ScriptEngineManager` which allows the developer to run code, such as JavaScript, in Java hence the behavior could more easily be captured and tested. This solution was not implemented as time was sparse at time of discovery.

6.6 Future works

As shown in the results of the acceptance tests, presented in section 5.5, it is already possible to use Buff to write programs that include **sequential operations**, **conditional selections** and **looping constructs**. However, Buff can still be improved upon, and in this section we will present a couple of suggestions for future development on Buff.

6.6.1 Implementing mathematical sets

Sets play a big part in many forms of mathematics such as linear algebra and discrete mathematics where one cannot perform many computations without the ability to group data. Therefore, it would make sense that a language based on mathematics should have some data structure supporting mathematical sets. Having sets in Buff would be preferable, because performing operations such as summing up a set of numbers is going to be cumbersome in

Buff with no data structure supporting sets. In this case sets would be a desirable feature, but because of the time and research needed to be able to implement sets, we decided that it would not be worth it compared to the perks of using our time on implementing other important features and optimizing existing features.

6.6.2 Scopes and nested function definitions

In section 4.2.1 it was described how a variable would first be searched for in the current scope then enclosing scope, then that scope's enclosing scope and so on. This is needed in case of nested function definitions. However, it was found that implementing nested function definitions was a topic of greater complexity and would not be possible within the scope of the project without neglecting other important features. Therefore, the enclosing scope of a given function will always be the global scope hence the program will only ever contain two scopes. We still feel that the way we implemented the symbol table was good, because it makes our solution more scalable.

6.6.3 Printing in Buff

From the theoretical aspect of functional programming a determining characteristic is that it is pure, meaning it has no side effect. When looking at the practical aspect, some impurity is needed as described in section 2.4. With this in mind, the language should always strive towards purity and thereby, only include the absolute necessary impure features and limit it as much as possible. In regards of Buff, the only impure aspect is printing by suffixing a function call with "?" character. Furthermore, printing cannot be customized nor used in any computation and will therefore never have any effect on the running program. Because of this strict implementation it would be desirable to copy this approach, such that not only the function calls could be printed, but also any expression such that a Buff program could be compiled and produce the output seen in code snippet 38. This functionality was not included due to time constraints, but would be desirable to implement in future works.

```
// Code
(2 + 5) * 7 ?

// Result
(2 + 5) * 7 -> 49
```

Code Snippet 38: Future printing capabilities in Buff

7 Conclusion

The project in hand set out to solve the problem statement presented in section 2.8 which is as follows:

How can a functional language which resembles mathematical syntax and an appertaining compiler be implemented that has the potential to serve as the first programming language for beginners?

To solve this, a problem analysis was conducted that went through initial research on what difficulties a beginner had when learning programming. It was found that the syntax differing from their usual understanding and confusing concepts such as assignments were a determining factor in the difficulty of learning a programming language. Both problems could be minimized by using a functional language as it does not concern itself with difficult concepts such as assignments and other side effects, but rather the pure aspect of programming. Furthermore, other research found that some determining predictors also played a role in learning programming. To support these predictors, the functional languages were also found to be the best solution. Therefore, a language design for a functional language was formed and argued for throughout chapter 3. Finally, the implementation showed some difficulties in implementing a functional language such as the concept of high order functions and nested function definitions, resulting in some common aspects of a functional languages not being part of Buff. However, this is seen as a minor drawback as many other aspects such as immutable data, a minimal amount of side effects and recursion are a part of Buff. Therefore, the omission of high order functions and nested function definitions is not seen as a determining factor in whether or not the problem statement is fulfilled.

Section 2.5 outlined three principles an introductory course in programming should include. As Buff aims to be a language that can be used in an introductory course it should process the capabilities to fulfill all three principles.
The first two ***Convey the elementary techniques of programming*** and ***Introduce the essential concepts of computing*** are both fulfilled as of subsection 3.4.2, where it was explained how Buff provides full power of a computing machine as it allows for: **Sequential operations** via use of *semicolon*, **Conditional selection** via use of the *if* construct and lastly **Looping** via use of *recursion*. The last one ***Foster the development of analytical thinking and problem solving skills*** correlates with the predictor *ability of problem solving and logical thinking* which both are somewhat abstract criteria to fulfill in a language, but Buff attempts to fulfill this by ensuring

that complex algorithms (nurturing the ability to solve complex issues) can be written in an intuitive mathematical syntax, such that no extra knowledge is required before one can approach a problem. From this, it is found that one can implement complex algorithms by use of basic principles in combination with more complex concepts such as recursion.

As of this in continuation of chapter 6 and the preceding sections, Buff is said to fulfill the problem statement from section 2.8.

Bibliography

[1] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, Oct. 2017. DOI: 10.1145/3077618. [Online]. Available: https://doi-org.zorac.aub.aau.dk/10.1145/3077618.

[2] I. Boljat and N. Bubica, "Teaching of novice programmers: Strategies, programming languages and predictors," Tech. Rep., 2014.

[3] D. Turner, "Total functional programming," *Journal of Universal Computer Science*, vol. 10, no. 7, pp. 751–768, 2004.

[4] V. Verma, *Does java support goto?* Available at https://www.geeksforgeeks.org/functional-programming-paradigm/ (01/03/2021).

[5] GeeksforGeeks, *Does java support goto?* Available at https://www.geeksforgeeks.org/g-fact-64/ (01/03/2021).

[6] Oracle, *Java language keywords*, Available at https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html (01/03/2021).

[7] K. Hinsen, "The promises of functional programming," Tech. Rep., 2009.

[8] Haskell, *Haskell - an advanced, purely functional programming language*, Available at https://www.haskell.org/ (01/03/2021).

[9] M. Chakravarty and G. Keller, "The risks and benefits of teaching purely functional programming in first year," English, *Journal of Functional Programming*, vol. 14, no. 1, pp. 113–123, Jan. 2004, Copyright - 2004 Cambridge University Press; Last updated - 2010-06-08. [Online]. Available: https://www-proquest-com.zorac.aub.aau.dk/scholarly-journals/risks-benefits-teaching-purely-functional/docview/213462701/se-2?accountid=8144.

[10] W. Kurt, *Crafting a Compiler*. Manning Publications Co., 2018, ISBN: 9781617293764.

[11] Haskell.org, *Haskell in education*, Available at https://wiki.haskell.org/Haskell_in_education (08/03/2021).

[12] V. TIRRONEN, S. UUSI-MÄKELÄ, and V. ISOMÖTTÖNEN, "Understanding beginners' mistakes with haskell," English, *Journal of Functional Programming*, vol. 25, p. 30, 2015, Copyright - Copyright © Cambridge University Press 2015; Last updated - 2017-04-10. [Online]. Available: https://www-proquest-com.zorac.aub.aau.dk/scholarly-journals/understanding-beginners-mistakes-with-haskell/docview/1885730854/se-2?accountid=8144.

[13] Edsger W.Dijkstra, *To the members of the budget council*, Available at https://www.cs.utexas.edu/users/EWD/transcriptions/OtherDocs/Haskell.html (20/05/2021).

[14] DPD, *Functional as a first language*, Available at https://softwareengineering.stackexchange.com/questions/73505/functional-as-a-first-language (20/05/2021).

[15] R. W. Sebesta, *Concepts of programming languages*, Global edition. Pearson education limited, 2016, ISBN: 978-1-292-10055-5.

[16] University of Oxford, *Functional programming: 2020-2021*, Available at http://www.cs.ox.ac.uk/teaching/courses/2020-2021/FOCS/ (18/05/2021).

[17] Tomas Petricek, *What to teach as the first programming language and why*, Available at http://tomasp.net/blog/2019/first-language/ (20/05/2021).

[18] Mario T. Lanza, *Why aren't functional programming languages used more as a first programming language in computer science courses?* Available at https://www.quora.com/Why-arent-functional-programming-languages-used-more-as-a-first-programming-language-in-Computer-Science-courses (20/05/2021).

[19] A. Krauss, *Programming concepts: Static vs. dynamic type checking*, Available at https://thecodeboss.dev/2015/11/programming-concepts-static-vs-dynamic-type-checking/ (05/03/2021).

[20] C. N. Fischer, R. K. Cytron, and J. Richard J. LeBlanc, *Crafting a Compiler*. Pearson Education, Inc., 2009, ISBN: 978-0-13-606705-4.

[21] T. Parr, *The definitive Antlr4 Reference*. The Pragmatic Programmers, 2012, ISBN: 9781934356999.

[22] S. circles, *How to use operator precedence in algebra*, Available at https://www.intmath.com/blog/mathematics/how-to-use-operator-precedence-in-algebra-12416 (26/03/2021).

[23] D. M. Ritchie, "The development of the c language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.

[24] B. Thomsen, Lecture in Languages and Compilers, Mar. 2021.

[25] H. Hüttel, *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010.

[26] SableCC, *Sablecc documentation*, Available at https://sablecc.org/documentation (07/03/2021).

[27] F. Flannery, C. S. Ananian, D. Wang, and M. Petter, *Cup user's manual*, Available at http://www2.cs.tum.edu/projects/cup/docs.php (07/03/2021).

[28] Antlr, *Antlr*, Available at https://www.antlr.org/ (2014).

[29] E. T. from the arXivarchive page, *Data mining reveals the crucial factors that determine when people make blunders*, Available at https://www.technologyreview.com/2016/06/24/108265/data-mining-reveals-the-crucial-factors-that-determine-when-people-make-blunders/ (25/03/2021).

[30] T. sprint, *Design patterns - visitor pattern*, Available at https://www.tutorialspoint.com/design_pattern/visitor_pattern.htm (26/03/2021).

[31] L. Bettini, S. Capecchi, and B. Venneri, "Double dispatch in c++," *Software: Practice and Experience*, vol. 36, no. 6, pp. 581–613, 2006.

[32] L. Prechelt and W. F. Tichy, "A controlled experiment to assess the benefits of procedure argument type checking," *IEEE Transactions on Software Engineering*, vol. 24, no. 4, pp. 302–312, 1998. DOI: 10.1109/32.677186.

[33] Wikipedia, *Don't repeat yourself*, Available at https://en.wikipedia.org/wiki/Don%27t_repeat_yourself (13/05/2021).

[34] M. Corp., *Unit testing best practices with .net core and .net standard*, Available at https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices (09/04/2021).

[35] Guru99, *What is functional programming? tutorial with example*, Available at https://www.guru99.com/functional-programming-tutorial.html#3 (19/05/2021).

Appendices

A Source code

All source code related to the project can be found in the following git repository:

`https://github.com/humleflue/P4`

B Documentation

All documentation related to the project can be found on the following URL:

`https://skaalum.tech/buff/documentation/`

C Initial version of the CFG

The initial version of our context-free grammar (CFG) can be found in grammar 1, and the definition of the terminals can be found in table 3.2. For the sake of simplicity in the initial version of the CFG, we have decided on the following:

- all declarations have to appear before the first statement.
- to be able to implement the parser by hand the the language had to be converted into LL(1), with the possible consequence of lowering the readability of the CFG.
- only the mathematical operators plus and minus are going to be implemented.

$$
\begin{aligned}
\langle prog \rangle &\rightarrow \langle dcls \rangle\ \langle stmts \rangle\ \$ \\
\langle dcls \rangle &\rightarrow \langle dcl \rangle\ \langle SEMICOlON \rangle\ \langle dcls \rangle \\
&\mid\ \lambda \\
\langle dcl \rangle &\rightarrow \langle type \rangle\ \langle ID \rangle\ \langle LPAREN \rangle\ \langle dclparams \rangle\ \langle RPAREN \rangle\ \langle ASSIGN \rangle\ \langle stmt \rangle \\
\langle type \rangle &\rightarrow \langle NUMBERDCL \rangle \\
&\mid\ \langle TEXTDCL \rangle \\
\langle dclparams \rangle &\rightarrow \langle dclparam \rangle\ \langle dclmoreparams \rangle \\
&\mid\ \lambda \\
\langle dclmoreparams \rangle &\rightarrow \langle COMMA \rangle\ \langle dclparam \rangle\ \langle dclmoreparams \rangle \\
&\mid\ \lambda \\
\langle dclparam \rangle &\rightarrow \langle type \rangle\ \langle ID \rangle \\
\langle stmts \rangle &\rightarrow \langle stmt \rangle\ \langle SEMICOLON \rangle\ \langle stmts \rangle \\
&\mid\ \lambda \\
\langle stmt \rangle &\rightarrow \langle math \rangle \\
&\mid\ \langle TEXTVAL \rangle \\
\langle math \rangle &\rightarrow \langle term \rangle\ \langle followterm \rangle \\
\langle followterm \rangle &\rightarrow \langle simpleops \rangle\ \langle math \rangle \\
&\mid\ \lambda \\
\langle term \rangle &\rightarrow \langle funccall \rangle \\
&\mid\ \langle Val \rangle \\
\langle simpleops \rangle &\rightarrow \langle PLUS \rangle \\
&\mid\ \langle MINUS \rangle \\
\langle val \rangle &\rightarrow \langle LPAREN \rangle\ \langle math \rangle\ \langle RPAREN \rangle \\
&\mid\ \langle NUMBERVAL \rangle \\
\langle funccall \rangle &\rightarrow \langle ID \rangle\ \langle LPAREN \rangle\ \langle stmtparams \rangle\ \langle RPAREN \rangle
\end{aligned}
$$

$$
\begin{array}{lll}
\langle stmtparams \rangle & \rightarrow & \langle stmt \rangle \ \langle stmtmoreparams \rangle \\
& | & \lambda \\
\langle stmtmoreparams \rangle & \rightarrow & \langle COMMA \rangle \ \langle stmt \rangle \ \langle stmtmoreparams \rangle \\
& | & \lambda
\end{array}
$$

Grammar 1: Our initial context-free grammar, which is LL(1).