

---

---

# **PEAK+** **(Programming Education And Knowledge+)**

- P4 -

---

---



# **PEAK+**

Project Report  
SW4-01

Aalborg University  
Department of Computer Science

Copyright © Aalborg University 2023

Overleaf, Discord, Jira, Outlook.com, GitHub, Microsoft Teams, ChatGPT, Tablesgenerator.

**Title:**

PEAK+ - Programming Education and Knowledge+

**Theme:**

Design, definition and implementation of programming languages

**Project Period:**

Spring Semester 2023

**Project Group:**

SW4-01

**Participant(s):**

Charlotte Sundahl Elleby  
Christian Povlsen  
Nichlas Seerup Hjorth  
Rasmus Bartholomay Vikøren Borup  
Thomas Bjeldbak Madsen  
Thomas Illum Andersen

**Supervisor(s):**

Lone Leth Thomsen

**Copies:** 1

**Page Numbers:** 137

**Date of Completion:**

May 24, 2023

**Abstract:**

The learning curve of a new programming language for beginners who have experience with block-based languages is equally difficult for beginners, with the same amount of experience, but with text-based languages. This provides a need for a language that introduces beginners to the textual industry-standard programming languages. This report details the development and creation of PEAK+ , a programming language that aims to fulfil this aforementioned need. The necessary syntax for PEAK+ was developed through an analysis of existing programming languages, including Scratch, Quorum, C, and Python. The languages were analyzed with Sebesta's language criteria. Through this analysis, the syntax and semantics of PEAK+ were designed. A functional compiler was built, by going through 3 essential phases: syntax analysis, contextual analysis, and code generation. ANTLR was used as a parser generator tool, and a visitor pattern, developed in C# was used to traverse the compiler. The compiler was tested through unit, integration, and acceptance testing. The language was concluded to have theoretically solved the need for a language between block-based and text-based programming languages, however, to fully conclude a successful solution, proper user testing must be completed.



# Preface

This report was written by CS-23-SW4-01, a group of students in the fourth semester of studying Software at Aalborg University.

## Code snippets

The report contains code snippets. In some snippets, parts of the code that are less relevant to the points being made have been omitted. This is indicated with [. . .] as seen in listing 1.

```
1 //First part of code snippet
2 .
3 .
4 .
5 //Continuation of the code snippet
```

**Listing 1:** Example of the structure of a code snippet. Lines 2-4 indicate omitted code

## Target audience

The target audience for this project is computer science students at the beginning of their fourth semester. The target group for the product is high school students who are learning programming for the first time.

## Application of theory

The report is structured in a way so that the necessary theory for a section or subsection is presented to the reader at the start of the section. Subsequently, the theory will be applied in a way that assumes the reader understands it.

## Acknowledgements

We would like to express our gratitude to our supervisor Lone Leth Thomsen for her guidance and support throughout the project. Furthermore, we would like to thank Morten Konggaard Schou for assisting us with the semantic definitions for PEAK+ .

Aalborg University, May 24, 2023

---

Charlotte Sundahl Elleby  
<celleb21@student.aau.dk>

---

Christian Povlsen  
<cpovls21@student.aau.dk>

---

Nichlas Seerup Hjorth  
<nhjor21@student.aau.dk>

---

Rasmus Bartholomay Vikøren Borup  
<rborup21@student.aau.dk>

---

Thomas Bjeldbak Madsen  
<tbma21@student.aau.dk>

---

Thomas Ilum Andersen  
<tian21@student.aau.dk>

# Contents

<b>Preface</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Problem analysis</b>	<b>3</b>
2.1 Learning programming . . . . .	3
2.2 Transitioning from block-based to text-based programming . . . . .	4
2.3 Analysis of Related Programming Languages . . . . .	5
2.3.1 Analysis of Scratch . . . . .	7
2.3.2 Analysis of relevant text-based programming languages . . . . .	10
2.3.3 Selecting the languages . . . . .	11
2.3.4 Comparing languages . . . . .	19
2.4 Summary . . . . .	22
2.5 Problem statement . . . . .	23
2.6 Configuration Management . . . . .	24
2.6.1 Agile Project Management . . . . .	24
2.6.2 Version Control . . . . .	24
2.6.3 Supervisor meetings . . . . .	25
2.6.4 Additional tools . . . . .	25
<b>3 Language Design of PEAK+</b>	<b>27</b>
3.1 Language Criteria . . . . .	27
3.1.1 Readability . . . . .	27
3.1.2 Writability . . . . .	28
3.1.3 Reliability . . . . .	29
3.2 The Paradigm of PEAK+ . . . . .	29
3.3 Requirements . . . . .	30
3.4 Syntax Design . . . . .	31
3.4.1 Context-Free Grammars . . . . .	31
3.4.2 Backus-Naur Form . . . . .	32
3.4.3 Context-Free Grammar for PEAK+ . . . . .	33
3.4.4 Operator Precedence . . . . .	37
3.4.5 Scope Rules . . . . .	38

3.5	Semantics . . . . .	40
3.5.1	Abstract Syntax . . . . .	40
3.5.2	Type Rules . . . . .	41
3.5.3	Operational Semantics . . . . .	44
3.5.4	Transition Systems for PEAK+ . . . . .	47
3.5.5	Abstract Syntax Tree Design . . . . .	53
<b>4</b>	<b>Implementation</b>	<b>55</b>
4.1	Compiler Phases . . . . .	55
4.1.1	Syntax Analysis . . . . .	55
4.1.2	Contextual Analysis . . . . .	56
4.1.3	Code Generation: . . . . .	56
4.2	PEAK+ compiler . . . . .	56
4.2.1	Deciding Compiler Language . . . . .	57
4.3	Syntax Analysis . . . . .	57
4.3.1	Parsing Theory . . . . .	57
4.3.2	Deciding Parser Tool . . . . .	59
4.3.3	Lexer . . . . .	60
4.3.4	The Visitor Pattern . . . . .	60
4.3.5	Building the Abstract Syntax Tree . . . . .	61
4.4	Contextual Analysis . . . . .	65
4.4.1	Symbol Table . . . . .	65
4.4.2	Type Checker . . . . .	70
4.4.3	Error Handling . . . . .	72
4.5	Code Generation . . . . .	74
4.5.1	Emitting lists and list methods . . . . .	75
4.5.2	Emitting foreach loop . . . . .	77
4.5.3	Emitting functions . . . . .	79
<b>5</b>	<b>Testing</b>	<b>81</b>
5.1	Testing of the PEAK+ compiler . . . . .	81
5.2	Unit Test . . . . .	82
5.3	Integration Test . . . . .	83
5.4	Acceptance Test . . . . .	85
5.4.1	Code Example: Scope test . . . . .	85
5.4.2	Code Example: Update of variables . . . . .	88
5.4.3	User Testing . . . . .	90
<b>6</b>	<b>Discussion</b>	<b>91</b>
6.1	Language Revisions . . . . .	91
6.1.1	Compiler revisions . . . . .	93
6.2	Requirement evaluation . . . . .	94



6.3	Time Management . . . . .	97
6.4	Tests . . . . .	98
6.5	Future Work . . . . .	98
6.5.1	User Test . . . . .	98
6.5.2	Code Functionality . . . . .	99
<b>7</b>	<b>Conclusion</b>	<b>101</b>
	<b>Bibliography</b>	<b>103</b>
<b>A</b>	<b>CFG</b>	<b>107</b>
<b>B</b>	<b>ANTLR CFG + LEXER</b>	<b>109</b>
<b>C</b>	<b>Type System</b>	<b>113</b>
C.1	Expression . . . . .	113
C.2	Statements . . . . .	114
<b>D</b>	<b>Structural Operational Semantics</b>	<b>115</b>
<b>E</b>	<b>Testing</b>	<b>119</b>
E.1	Acceptance Testing . . . . .	119
E.1.1	Code Example: Insertion-Sort . . . . .	119
E.1.2	Code Example: GDB Calculator . . . . .	124
E.1.3	Code Example: test input & repeat . . . . .	128
E.1.4	Code Example: Triangle area calculate. . . . .	130
E.1.5	Mathematical Expressions . . . . .	131
E.1.6	Missing ";" error message . . . . .	132
E.1.7	Wrong type error message . . . . .	133
E.1.8	String Concatenation . . . . .	133
E.1.9	Zero Division . . . . .	134
E.1.10	Symbol Table scope-fix . . . . .	135



# Chapter 1

## Introduction

In today's world, programming has become an important skill [42]. However, for beginner programmers, the process of getting started with programming can be daunting due to the steep learning curve. Existing programming languages can be complex, requiring significant prior knowledge, and making it challenging for beginner programmers to pursue programming as a career or hobby [40].

This problem produces an increasing demand for the creation of new programming languages that are specifically designed for beginners[20]. These languages should be user-friendly, easy to learn, and provide an immersive experience that encourages learners to hone their skills.

As an introduction to programming, block-based languages are very commonly used [45], but as these languages are limited to the block structure, the beginner programmer at some point has to switch to a text-based language. Even though learning block-based languages develops computational thinking [45], transitioning to text-based programming is difficult. It requires persistence to remember the syntax and semantics of the language and programming courses tend to have a high failure rate [12].

The aim of this report is to document the development of a new programming language PEAK+ , targeted towards beginner programmers in high school. This includes examining the rationale behind developing a new programming language for beginners and the essential features of such a language. The specification needed for a beginner-friendly programming language will be discovered by analyzing and comparing four programming languages: Scratch, C, Python, and Quorum.

This report takes the reader through the different phases of creating a compiler for this new language: PEAK+ .



# Chapter 2

## Problem analysis

This chapter will analyze the difficulty of learning programming as a beginner programmer. It will cover some basic analysis of different programming languages with the aim of eventually comparing them, from the perspective of beginner programmers, to finally arrive at a concrete problem statement. The chapter will include facts from studies but also our own experience as beginner programmers.

### 2.1 Learning programming

As programming is an important skill worldwide [42], it is widely incorporated into the study curriculum of high schools and colleges. Learning text-based programming requires the student to learn, understand and remember the syntax and semantics of the language. Furthermore, the students have to evolve their computational thinking in order to solve programming problems. To learn these skills students have to be persistent in the effort of studying programming, and programming courses tend to have a high failure rate [12]. Comparing these facts with our own experiences when being new programmers, it matches quite well. The learning curve required persistent activity and focus on the new content.

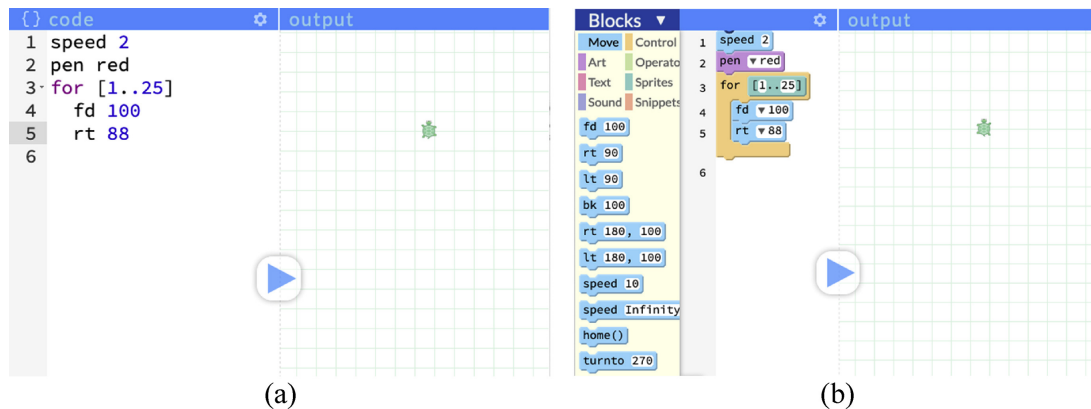
As an introduction to programming, block-based programming is an option. Block-based programming is mainly used to introduce children to programming, but it is also widely used in high school curricula [45].

Instead of writing lines of code and having to remember all the syntax, block-based programming is based on drag-and-drop blocks. Each block represents some function or action that the program should handle. This visual approach provides the user with an intuitive and accessible way to learn to program. The user can easily understand what they are doing and how to continue developing. Furthermore, it is quick to get easy tasks done and the user gets a visual response on what they have created [15].

Block-based programming has a positive impact on the motivation to learn programming and studies show that it evolves the student's computational thinking and learning of algorithmic skills [45]. However, to continue developing as a developer, the student will reach a point where they have to switch to text-based programming.

## 2.2 Transitioning from block-based to text-based programming

The use of block-based programming, in computer science educational purposes, is meant to be a preparation for the student to later transition into text-based programming languages. However, a study from 2019 states that the difficulty in learning a text-based language is not eased by first learning a block-based approach [45]. The study investigated the learning of a text-based language in two different high school classrooms. Both classrooms followed the same curriculum but in an introductory course to programming over five weeks, they used two different versions of the same programming environment: block-based and text-based. In figure 2.1 the two different approach is displayed.



**Figure 2.1:** The two different approaches of the language of pencil.cc with the text-based in the left and block-based in the right [45].

After five weeks both classes switched to learning the same text-based programming language Java. During the process, three tests were conducted, one before and after the introduction course and one after 10 weeks of learning Java. The outcome of the study is displayed in table 2.1.

	Block-based class	Text-based class
Test score before the learning course	54.3%	51.7%
Test score before starting the learning course	66.6%	58.8%
Test score learning Java	64.9%	65.7%

**Table 2.1:** Overview of the average test result for the two different high school classes [45].

Table 2.1 displays the result, it shows that the two classes were very similar in their prior programming knowledge. After the five-week introduction course, the class that followed the block-based version did significantly better. However, even though they gained a better understanding at first and scored higher on the mid-term test, the result of the last test was considered equal between the two classes. Besides the three tests that were conducted, data on their compilations were collected and reviewed. Like the test result, there was not any particular difference to find between the two classes [45].

Based on the result of the study [45] and personal experiences of transitioning from block-based to text-based programming, using a block-based approach as an introduction to programming is simple and more intuitive for a new programmer, compared to the text-based approach. However, when switching from block-based to text-based programming, the same sense of difficulty and the same mistakes appear as for those that were introduced to programming through a text-based approach [45].

In view of the demand for new beginner programming languages, it is interesting to analyze and evaluate the differences between block-based and text-based programming. The analysis should illuminate the positive and negative aspects of text-based and block-based languages with beginner programmers as the target group.

## 2.3 Analysis of Related Programming Languages

To gain an insight into the difference between block-based and text-based languages, these will be evaluated using Sebesta's criterion, which is relevant when examining features in programming languages.

The analysis begins with the block language scratch in section 2.3.1. Furthermore, analysis and evaluation of text-based programming languages, including C, Python, and Quorum, in section 2.3.2. The reasoning behind those specific languages being analyzed will be further elaborated in the following sections.

The languages are examined in terms of how they perform across Sebesta's criteria based on their readability, writability, and reliability. Furthermore, to identify their

advantages and disadvantages for new programmers. Each criterion is further subdivided into several factors that contribute to the overall evaluation of a programming language as displayed in figure 2.2. All the information upon Sebesta's criteria in this report is based on the book "Concepts of Programming languages" [32].

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

**Figure 2.2:** Language Evaluation Criteria and the Characteristics that affect them [32].

### Readability

Readability refers to how easy it is to read and understand code written in a particular programming language. A language that is highly readable will be easy to understand, debug, and maintain. Factors that affect readability include syntax clarity, ease of comprehension, and consistency of style [32].

### Writability

Writability refers to how easy it is to write code in a particular programming language. A language that is highly writable will be efficient for programmers to use, reducing development time and effort. Factors that affect writability include syntax simplicity, ease of expressing algorithms, and flexibility of control structures [32].

### Reliability

Reliability refers to how likely a program written in a particular programming language is to function correctly and consistently under different circumstances. A language that is highly reliable will produce fewer errors and bugs, resulting in more robust and stable programs. Factors that affect reliability include type checking, error handling, and exception handling [32].



### 2.3.1 Analysis of Scratch

Scratch is a Block-based programming language, which is a type of programming language that uses graphical blocks instead of traditional text-based code. These blocks represent programming concepts such as loops, conditional statements, and functions, and are typically arranged on the screen by dragging and dropping them into place. Some other popular block-based programming languages include Blockly and Code.org's App Lab [39]. Scratch is chosen because it is very popular and widely used in education, as it was developed to help children learn and express themselves through writing code. Scratch had 82 million users in December 2021, over 638 million projects created, and is the world's largest coding community for kids [4].

Scratch is a coding language with a simple visual interface [2]. Scratch creates a visual environment for programmers which relies on the drag-and-drop method of programming. The different applicable components are structured in blocks with color coordination similar to LEGO blocks, as seen in figure 2.3.

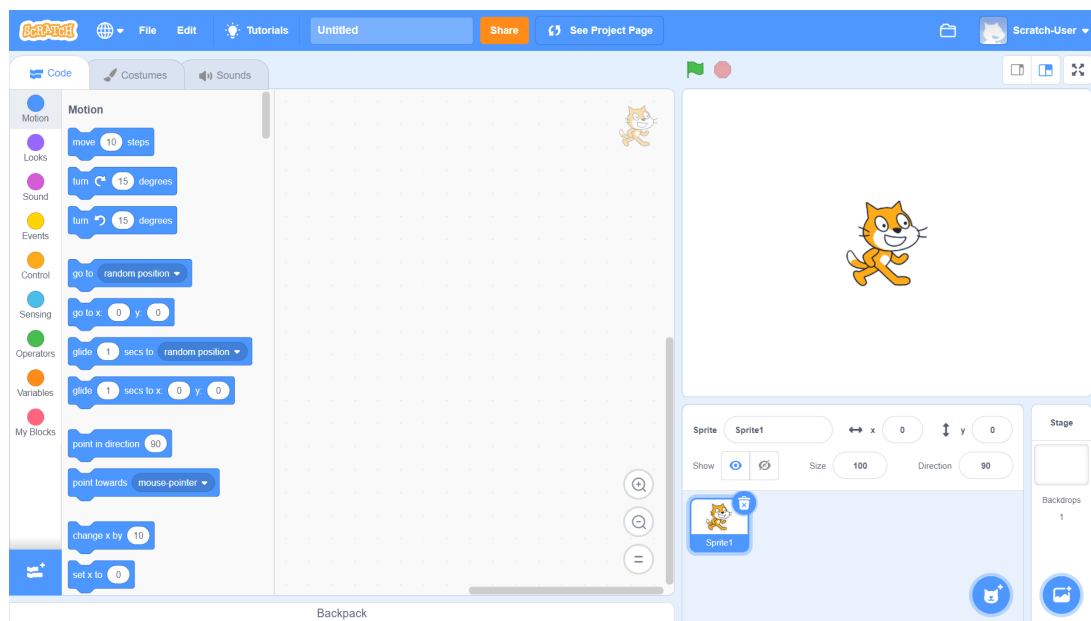


Figure 2.3: Scratch Example [31].

Scratch and other block-based languages, however, also have limitations in regard to their complexity. It cuts down on a lot of essential features that many of the popular text-based programming languages provide. This leads to a limited syntax, which makes it difficult to produce complex applications. The limited portability and scalability of block-based languages underline their purpose and target audience. It is clear from this that block-based languages are not intended to be widely used in the

industry, but rather as a form of introduction to programming.

If we take a look at Scratch in relation to Sebesta's Criteria in table 2.2, the following analysis can be derived.

### **Simplicity & Orthogonality**

A language like Scratch is designed with learning in mind. Simplicity and orthogonality are very relevant when discussing block-based languages. Although Scratch is more focused on new programmers, it also has capabilities that an advanced programmer would use, including mutations of blocks or using component collections.

Simplicity & Orthogonality have a special relation to mutation. A block with mutation(s) can be used in two ways. Beginner programmers can use blocks without mutations easily, while advanced users benefit from the optional parameters of the block.

In regards to abstract programming, component collection can be used when you want to set the width property of every button in a screen, instead of setting the property for the buttons one by one. This leads to a higher abstraction level of planning and development.

### **Data types**

Scratch is very simple, and therefore only provides the basic data types such as string, number, and boolean. This also means that there are no subtypes in Scratch, such as integers or chars. This design consideration certainly simplifies the beginning phases of learning programming, since there are not multiple different ways to store a number. Still, the developer receives a core understanding of what data types are and will benefit from it if they pursue a lower-level language later on.

### **Syntax Design**

The block groups have different colors and it is possible to play with the shapes of the blocks related to their different stacking and embedding types. This means that each block can contain different kinds of function block shapes e.g. a number that is a rectangle with rounded corners fits inside a "change x by \*shape\*" command block. These methods are very helpful for beginners in developing programs since they are given clues as to how to stack the blocks on top of each other. This usually benefits syntax considerations and avoids irritating "syntax error" messages.

### **Support for abstraction**

Scratch has limited support for abstraction in the form of function definition. The programmer is able to create their own blocks and manipulate them to an extent.

However, generalization of data types is not possible and it can be argued that the functions which can be created are restricted in their complexity.

### Expressivity

Expressivity in Scratch is enhanced by the presence of powerful blocks that make it possible to accomplish a lot with a few blocks. This does mean that the programmer does not have full freedom in using these blocks, however, they do accelerate the learning process for beginners as they move towards the intermediate level. Apart from this, Scratch provides limited expressivity, as there in most cases are not multiple ways of accomplishing the same task. This is due to the blocks being specialized for specific purposes, taking away some freedom from the programmer while simplifying their use.

### Type checking

Scratch performs type-checking to ensure that the blocks used in a script are compatible. In Scratch, the blocks are geometrically shaped to indicate their types, like in figure 2.4. For example, a boolean is shaped like a hexagon, and a rounded rectangle is for numbers or strings. By using these shapes, you are able to look at command blocks and the shapes they accept, to determine which kind of types you should make use of. This version of type-checking allows beginners to understand that there exist different types, which are not all interchangeable [22].

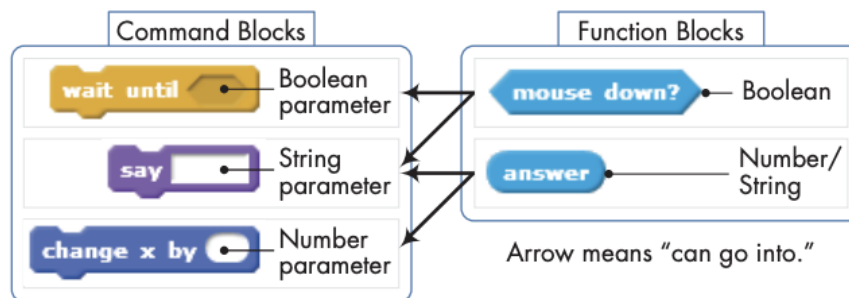


Figure 2.4: Scratch blocks & types [22].

### Exception handling

Exception handling is not usually an accessible feature of Block-based languages. There are no known statements on why exception handling is not an implemented feature within most block-based languages. However, it is fair to say Exception handling can be a complex concept for beginner programmers [33], and could therefore be the reason why it is not implemented. This means Scratch programs simply have to be error-free in order to run without crashing.

**Restricted aliasing**

Since you are not able to have two or more distinct names in a block-based language that can access the same memory cell, as well as the lack of pointer implementation or a form of reference, Scratch does not have aliasing, and strict aliasing is not a relevant problem.

**Summary**

Block-based programming languages rely on graphical blocks instead of traditional text-based code, with Scratch being one of the most popular, with 82 million users and over 638 million projects created. Other popular block-based programming languages include Blockly and Code.org's App Lab. The simplicity and orthogonality of Scratch make it an excellent language for new programmers to learn. However, this simplicity also means Scratch is limited in terms of complexity, making it difficult to produce complex applications. The text also provides an analysis of Scratch in relation to Sebasta's Evaluation Criteria, discussing its simplicity, data types, syntax design, support for abstraction, and expressivity. Overall, Scratch is an excellent language for teaching basic coding concepts to children and beginners, but may not be suitable for developing more advanced applications.

**2.3.2 Analysis of relevant text-based programming languages**

In section 2.3.3, three text-based programming languages are analyzed. In order to select more relevant languages to analyze, considering programming paradigms is appropriate.

**Programming Paradigms**

Programming paradigms help categorize different styles of programming and can be used to describe a programming language. Some of the most commonly used programming paradigms include 'imperative' and 'declarative'. The imperative programming paradigm is a way of programming where the instructions given are used to change between states. Each instruction given, like changing the contents of a variable, represents a state change [50]. Examples of this are C, Java, and C#.

In the declarative programming paradigm, no instructions regarding how to compute anything are given, rather the properties of the needed result are given and the underlying required computations are made based on this result [50]. Examples of this are most database query languages like SQL. Because many paradigms are similar and/or derive from other paradigms, most programming languages are considered to be multi-paradigms.

**Procedural programming:** Within the imperative- and declarative programming paradigm are subcategories. Both 'procedural' and 'object-oriented' are subcategories of imperative programming. The procedural paradigm is about changing states using procedures/subroutines. Procedures are functions with the purpose of changing the value of their arguments and they do not return an output. Most procedural languages are also imperative but the main difference is that the procedural paradigm depends on scopes and therefore most languages containing control structures like *for*, *while* and *if* are procedural [50]. A language without scopes that instead use for example 'goto' would be considered imperative but not procedural. C, Java, and C# are also considered procedural programming languages.

**Object-oriented programming:** Instead of changing states through procedures, the object-oriented paradigm changes states through its objects which themselves can contain procedures and functions. In this way, they are similar to the procedural paradigm, and usually, an object-oriented programming language is also considered procedural as it supports procedures and sequences of instructions [50]. Examples include Java and C#.

**Functional programming:** The most notable subcategory within declarative programming is functional programming. In functional programming, the result wanted by the user is defined using a series of functions [48]. In the functional paradigm, a function cannot contain side effects, meaning that any input is mapped directly to an output. There do not exist popular purely functional programming languages, though some languages like C# include features from the functional paradigm with their use of LINQ and lambda expressions [48].

### 2.3.3 Selecting the languages

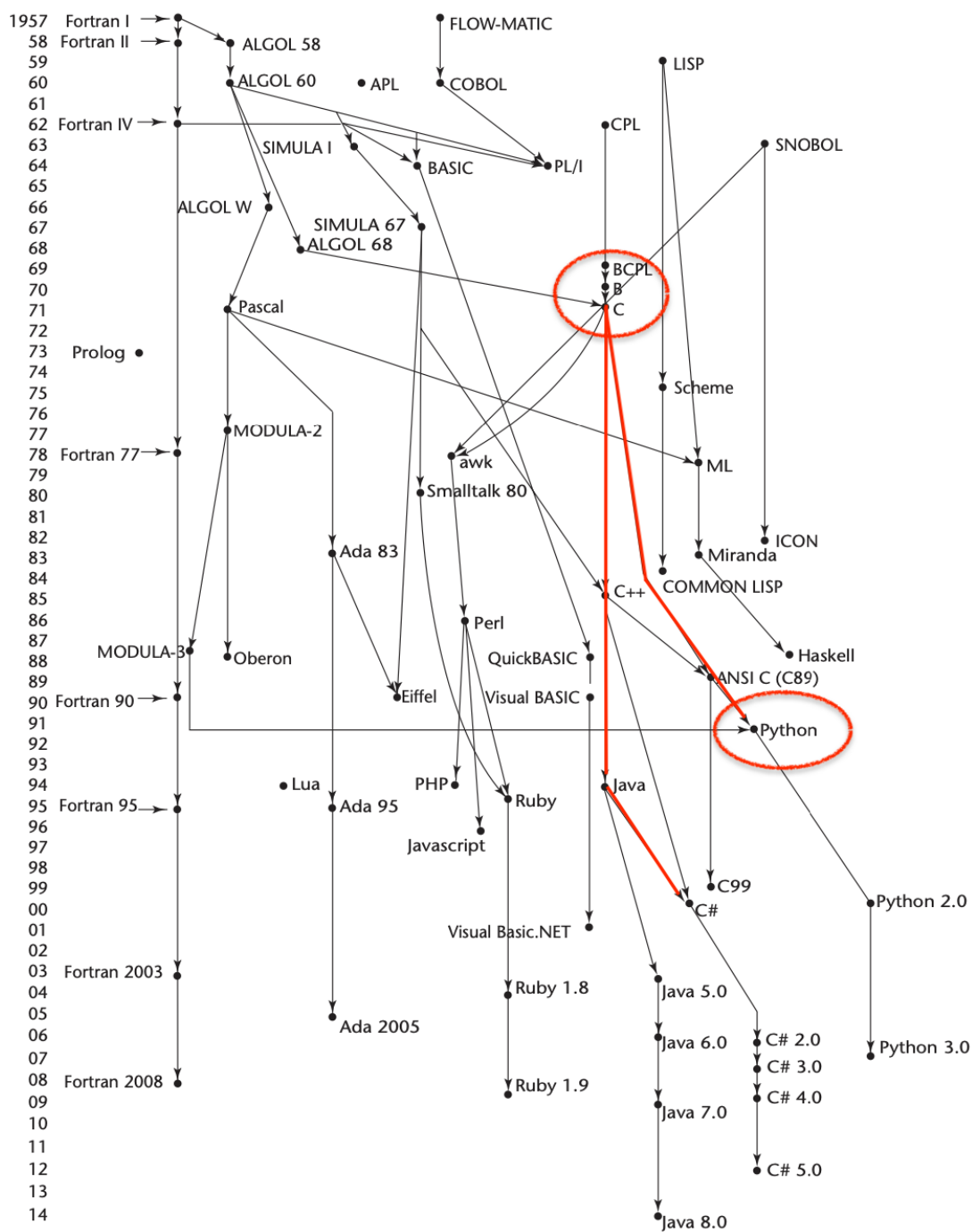
The languages will be from the imperative programming paradigm, including the paradigms: procedural and Object-oriented, as languages in these paradigms, are easy to learn and read, compared to languages from the declarative paradigm [16].

When considering the selection of languages to analyze within the imperative paradigm, Python, Quorum, and C is chosen based on the following arguments:

Python is chosen because it has a simple syntax [28], is widely used in the industry [10], and is a common starting language for new programmers [43].

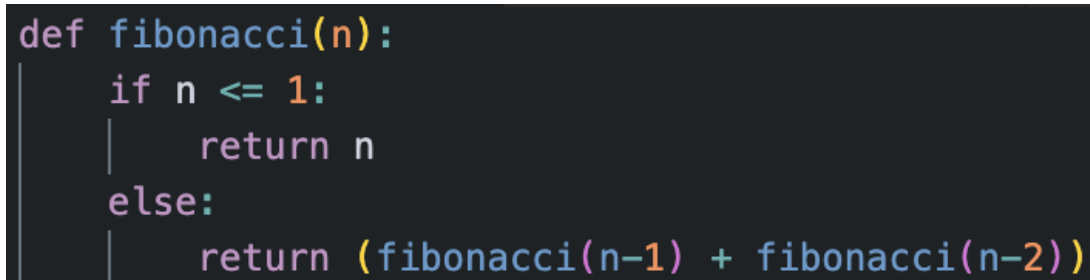
Quorum is interesting to analyze because it is used as an introductory language, e.g. in high schools. It has a simple and readable language design [14]. Furthermore, Quorum is an evidence-based language meaning that it was developed based on the results of scientific research in order to improve its language design [30].

C is selected because it is widely used in the industry [34], and because it is the predecessor for many other widely used languages such as Python, Java, and C#. This can be seen in figure 2.5. C and Python are circled to indicate their presence in the tree, and the red lines indicate the connection between C and the other mentioned languages. However, C is viewed as a less beginner-friendly language, as it has constructs such as pointers, dynamic memory allocation, etc, which can be hard to grasp for new programmers [34].



## Python Language

Python is an object-oriented programming language with a simple syntax. Python supports multiple programming paradigms such as procedural and functional programming [28]. The Python programming language is considered a high-level text-based programming language. With an easy and English-like syntax, Python is considered a beginner-friendly language [35]. An example of a small Python program can be seen in figure 2.6



```
def fibonacci(n):  
    if n <= 1:  
        return n  
    else:  
        return (fibonacci(n-1) + fibonacci(n-2))
```

Figure 2.6: Python Fibonacci Example.

### Simplicity

In Python few lines can amount to larger functionalities. Due to this, some concepts can be easy to learn and read while others can seem cryptic. Python's short syntax makes it easier to write code, improving writability overall.

### Orthogonality

Python has a high degree of orthogonality. This means that the constructs in the language can be combined in multiple ways which initially makes Python easier to read and write [25]. However, having very high orthogonality might be a disadvantage, as it then allows very complex constructs due to combinatorial freedom.

### Data types

Python has multiple built-in data types, such as integers, strings, and floats, but it also supports more complex data types such as lists, etc. With many built-in data types, expressivity is improved. However, there are some downsides to having a lot of built-in data types, one of which is making the language more complex and harder to learn.

### Syntax design

Python's syntax is short and readable [44], which helps improve readability and writability. Python syntax uses indentation instead of curly brackets as scopes, which for some users makes the code more readable, whereas it can cause confusion for others who are used to curly brackets.



**Support for abstraction**

Python includes support for abstraction by letting the programmers define their own functions and classes and such. By having support for abstraction, writability is increased as it allows some functionality to be reused later on.

**Expressivity**

Python has high expressivity. This is due to the fact that Python allows its programmers to express ideas in multiple different ways, each with their own advantages. Expressivity can improve readability and writability. However, high expressivity may also make the language harder to read for some, as too many ways of expressing the same functionality may cause confusion. The programmer has to keep track of a larger collection of constructs in order to fully utilize the functionality of the language.

**Type checking**

Python is a dynamically typed language, which means that the interpreter assigns variables a type at runtime based on the variable's value at the time [24]. By being a dynamically typed language Python becomes more flexible. It can also lead to some type-related errors which may be difficult to catch during development. However, Python is at the same time strongly typed, meaning that variables have a type and that it matters when performing operations on the variable.

**Exception handling**

Python has support for exception handling. This makes it easier to develop code that can handle errors, which can help improve reliability by handling errors, which are expected to occur.

**Restricted aliasing**

Python does not have restricted aliasing. This slightly lowers the reliability of the language compared to languages that support it.

**Quorum Language**

Quorum is a high-level programming language, which is intended mainly for students but is broad enough to also be used commercially [30]. One of the main focuses of the language is for its syntax to be simple and accessible because it is meant to be easy to learn for all types of learners. Furthermore, it aims to improve language design overall [14]. An example of a small Quorum program can be seen in figure 2.7

```
action fibonacci(integer num) returns integer
  if num == 0 or num == 1
    return num
  else
    return fibonacci(num - 1) + fibonacci(num - 2)
  end if
end
```

Figure 2.7: Quorum Fibonacci Example.

### Simplicity

Since quorum is designed to be simple and easy to learn, its syntax is straightforward and easy to read, which enhances its readability. Its simplicity also enhances the language's writability. Finally, Quorum's simple design helps to reduce the potential for errors, which enhances its reliability.

### Orthogonality

Quorum supports a relatively small set of primitive constructs that can be combined in a variety of ways to build control and data structures. This orthogonality helps to make the language easy to learn and use, which enhances readability and writability. Additionally, orthogonality increases the reliability of code written in the language.

### Data types

The data types supported by Quorum include integers, booleans, strings, and arrays. The use of clear and consistent data types enhances Quorum's readability and writability, as it is easy to understand the purpose and behavior of different data types. Furthermore, the use of strong typing and type-checking mechanisms helps to improve the reliability of code written in the language.

### Syntax design

Quorum's syntax is designed to be simple and clear, which enhances its readability. The use of consistent and intuitive syntax design also helps to improve the language's writability, as developers can easily understand how to use different language constructs. However, writability is not as high as it could be, since the syntax in many cases, forces the developer to write the full name of a construct, e.g. "boolean", instead of "bool", as booleans are called in some programming languages. Having to write the full-length construct names instead of an abbreviation, increases readability, but reduces writability, as the programmer would be able to write a program faster using abbreviations for the constructs. Finally, the use of a clear syntax design improves the

reliability of code written in the language.

### **Support for abstraction**

Quorum includes support for abstraction, in the form of object-oriented features such as inheritance and encapsulation. This helps improve writability and reliability, as the developer is able to write reusable code that is easy to modify and maintain.

### **Expressivity**

Quorum low expressivity as it does not have many ways of expressing the same computations. For example, the `count++` notation from C and other languages does not exist in Quorum, where the only way of counting up a variable is `count = count + 1`. An example where Quorum contains constructs that can specify the same computation is with its loops. Quorum has three types of loops: Repeat x times, repeat until, and repeat while. Repeating x times makes it easy to repeat the same action a specific amount of times. However, repeat until and repeat while can also be used for the same purpose, but are more powerful, as they both run based on a boolean condition. By itself, the quorum language does not have very high expressivity, however, this changes with its different libraries, which simplify many different actions. Expressivity enhances the language's readability and writability, and developers can use language constructs that are intuitive and easy to understand.

### **Type checking**

Quorum is statically typed and includes strong type-checking mechanisms, which help to improve the reliability of code written in it. Type checking helps to ensure that code is correct and free from errors related to data types.

### **Exception handling**

Quorum includes support for exception handling by the programmer, which helps to improve the reliability of code written in the language. The error handling in Quorum allows the use of a Try-Catch-Finally statement, however, the syntax is different, as Quorum uses a check-detect-always syntax instead.

### **Restricted aliasing**

Quorum does not include support for restricted aliasing, which slightly lowers the reliability of the language compared to languages that support it.

## **C Language**

C is an imperative text-based programming language that is considered low-level compared to other more popular languages like Python or C#[47]. It is widely used due to its lower level, making it an efficient language for writing code closely linked to machine instructions. One way that C is considered low level is for example its

inclusion of pointers, which can point to an address in memory, similarly to machine instructions in assembly[47]. An example of a small C program can be seen in figure 2.8

```
int fibonacci(int n){  
    if (n <= 1){  
        return n;  
    } else {  
        return (fibonacci(n-1) + fibonacci(n-2));  
    }  
}
```

Figure 2.8: C Fibonacci Example.

### Simplicity

C has a limited amount of data types and keywords making it simple, however, some instructions, which in other languages require one line, may require multiple lines in C. An example of this is C's lack of built-in properties like arrays ".length" property which exists in most languages. An arrays length can be accessed any time through the array in C# and Python, while in c you would have to initialize the array and calculate the length of the array on separate lines. The language also requires some knowledge of low-level programming because of its inclusion of concepts like pointers and dynamic memory allocation.

### Orthogonality

C does not include higher-level components such as lists, but combining the components that are included can create more advanced structures. In C, certain operators cannot be used on some data types which are otherwise possible in some higher-level languages like C#. For example in C strings cannot be concatenated using "+".

### Data types

In C there are a limited amount of data types. Today most languages include data types like "bool" and "string". In C, "true" is represented by any integer value that is not 0, and to make a string, an array of characters is needed.

### Syntax Design

Some of the syntax in C can be considered hard to learn for beginners such as pointers and indirect component selection of struct fields. Many words are reserved and have special meanings in C. The only keyword which violates this is the keyword "static" which can be used to declare global variables within a scope, or used to define that an array parameter needs a specific length.

**Support for Abstraction**

In C, the programmer is able to create abstraction by using functions and structs. These abstractions can be considered low compared to higher-level languages which include classes, inheritance, and dynamic objects.

**Expressivity**

In higher-level languages the programmer is able to have more functionality on fewer lines than in C. Despite this, code can be written concisely in C though it may be hard to read.

**Type Checking**

C contains limited methods of type checking. C allows implicit typing, making detecting errors difficult

**Exception Handling**

C does not have built-in exception handling like other higher-level languages. For example, Quorum and Python have a try-catch structure and the ability to throw an error, like previously mentioned in their analysis. "Try-catch" and throwing an error is not possible in C

**Restricted aliasing**

C has "restricted pointers" which helps express anti-aliasing in C but besides this, C's heavy need for pointers when making functionality, makes aliasing occur often in the language. The restrict keyword on a pointer is a way of telling the compiler that the specific pointer is the only way of accessing the object pointed by it

**2.3.4 Comparing languages**

In this section, a comparison between the languages previously analyzed in section 2.3 is conducted. These languages will be compared based on their overall readability, writability, and reliability in order to understand the difference and deduct how a language aimed at beginners, compares to these languages. Each language is put on a scale ranging from low to high on readability, writability, and reliability. These scales represent how these language rank compared to each other based on the analysis.

## Comparing readability

First up is comparing readability. The languages are ranked on the scale, displayed in figure 2.9.



**Figure 2.9:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing readability.

C has the lowest readability of the languages that have been analyzed in figure 2.9. This is due to the fact that it is a low-level programming language, in which a line of code often closely resembles machine instructions. The main reason for not placing C lower on the readability scale is due to the fact that languages such as Assembly have notably lower readability.

Next on the scale in figure 2.9 is Python which has better readability. The reason is that Python focuses a lot more on writability which will be elaborated on in figure 2.10. In Python, the programmer is able to write readable code for intermediate programmers while also having the ability to write a lot of functionality within a few lines which may be confusing for beginner programmers.

The next language on the scale in figure 2.9 is Quorum. Quorum has been placed high on the readability scale. This is due to the fact that Quorum has a simple syntax, which focuses on readability over writability. Compared to a language like Python where the programmer is able to write flexible code with a focus on writability and less readability, there is less room for complex code in Quorum as it focuses on simplicity.

Scratch has high simplicity which makes it more beginner friendly and readable. Every block is labeled in order to remove ambiguity in the language (see figure 2.3). Coding larger functionalities in Scratch could be considered confusing for beginners, due to the large amount of blocks which can be combined, but despite this, the blocks still help define the functionalities.

### Comparing writability

Next, is comparing the writability. The languages is ranked on the scale, displayed in figure 2.9.



**Figure 2.10:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing writability.

The language with the lowest writability on the scale in figure 2.10 is C. One of the reasons for this is due to C being a low-level language and therefore has low expressivity. C does not contain many ways in which you can express complex concepts in a few lines. Compared to higher-level languages, C also has less support for abstraction like previously mentioned in the analysis. In general, C is very bare-bones and limited compared to other languages.

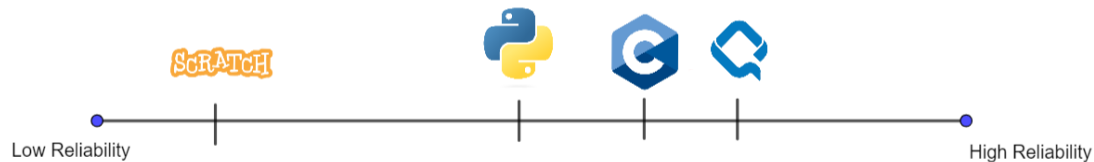
Quorum is placed in the on figure 2.10 regarding writability. Compared to C, it is more expressive and has more support for abstraction, for example in the form of its object-oriented features. Quorum is placed lower compared to Python and Scratch because of its need to be readable. For example, the data type "int" in C, is an abbreviation for "integer", which is the name used in Quorum.

In Scratch, each "block of code" has to be dragged in. This makes it simple to create code since all available code blocks can be seen in a sidebar. Scratch is placed lower on figure 2.10 than Python because despite how easy this is, Scratch has limited expressivity compared to Python where it may be faster to write code than looking for and dragging in multiple blocks.

Especially because of Python's high level of expressivity, a lot of functionality can be written in a few lines. Python is therefore placed highest on the scale in figure 2.10. Python also has a lot of support for abstraction and orthogonality compared to C, Quorum, and Scratch.

### Comparing reliability

The last criterion that is compared is reliability. The languages are ranked and displayed in figure 2.11.



**Figure 2.11:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing reliability.

Scratch does not have exception handling and is dynamically typed. If the programmer decides to set the value of a variable equal to two strings added together, no error is displayed, and instead, the value becomes 0. This makes Scratch less reliable and Scratch is therefore placed lowest on the scale in figure 2.11, as the programmer is not warned about unwanted behavior. Scratch does have restricted aliasing making it slightly more reliable.

Python is dynamically and strongly typed and is able to for example interpret strings to numbers and vice versa, however, the programmer can get unwanted behavior when letting Python do automatic type conversion. Python still allows for explicit type conversion though. This gives Python lower reliability, ranked lower than C and Quorum on the scale in figure 2.11, but higher reliability than Scratch.

In C the programmer does not have access to exception handling, but C is statically typed making it more reliable than Python and Scratch in figure 2.11. C has limited support for restricted aliasing, but with its focus on using pointers, it does have aliasing.

Quorum provides exception handling for the programmer and is statically typed. This makes Quorum more reliable than the other languages analyzed.

## 2.4 Summary

To summarize, The languages above are compared based on their ability to be understood by beginners, written in a straightforward way, and produce the desired output without errors. The analysis shows that C has the lowest readability and writability, while Python has high expressivity, which makes it easier to write code. Quorum focuses on readability over writability, making it more beginner-friendly. Scratch is also



beginner-friendly but has limited expressivity. In terms of reliability, C is statically typed, which makes it more reliable than Python and Scratch. Quorum provides exception handling and is also statically typed, making it more reliable than C. C lacks exception handling, making it less reliable than Quorum.

## 2.5 Problem statement

From the analysis, a vacancy between the existing languages can be seen. As a beginner programmer, there are several ways to learn to program. Learning to program in a block-based language is widely used and tends to be a positive way to evolve computational thinking and be introduced to programming. But despite this, beginner programmers find it difficult to transition from a block-based language to some of the industry standard languages. The introduction to a text-based programming language is challenging and requires persistence.

Based on the analysis and comparison of the four different languages there are different pros and cons seen from a beginner's perspective. By the conclusion of the analysis, the goal of our language is a beginner-friendly text-based language that is developed to introduce beginner programmers to the basic and important concepts in programming is needed.

From this, the following problem statement can be derived.

*How can a text-based programming language be developed for beginner programmers, using readable concepts from block-based languages, while focusing on facilitating the transition to programming languages used in the industry?*

A list of requirements has been made in order to successfully be able to say, that the problem statement has been solved. The list has been made in chronological order, and the details, importance, and use will be presented throughout the report.

- Define language criteria.
- Create a Context-Free Grammar (CFG).
- Define semantics.
- Define the scope rules.
- Design Abstract Syntax Tree (AST).
- Create a lexer and parser.
- Build symbol table and handle type checking.
- Successful code generation

- Testing of PEAK+ , including unit testing, integration testing, and acceptance testing.

## 2.6 Configuration Management

This section will cover how this project is managed, including the use of agile techniques, how the use of version control is conducted, and the tools used to manage the project. This section can be skipped, if there is no interest in the configuration management of this project.

### 2.6.1 Agile Project Management

Jira is a proprietary issue-tracking product that enables you to work more Scrum-like, compared to traditional project management approaches, like going over tasks in the waterfall method. It provides the opportunity to have Kanban boards and a road map, as well as a backlog. The backlog gives us the opportunity to write down tasks, also called issues in Jira. From the backlog, you are able to create sprints. The concept of sprints is well-known in the world of agile project management and consists of a dedicated period of time in which a set of tasks will need to be completed on a project [3]. These issues are then able to be assigned to specific sprints and which are then displayed on the Kanban board. So we have the ability to only display certain issues that are needed for these specific sprints. An example of how we used this was in the last month of the project when we wrote all of the issues that have to be handled before we submit the project. We then created sprints for every week until the submission date to make sure that we had sharp deadlines. In 3rd semester we worked on a Trello board and created cards to move around from "To Do" to "Review 1" and so on. Even though it did the job, it did not quite manage to solve our needs. It was a bit confusing having everything displayed at once on the to-do list. In this semester's project, we decided to use Jira instead [5].

### 2.6.2 Version Control

Just like the agile project management in section 2.6.1, we have a lot of experience with version control through GitHub & Overleaf. Version control, also known as source control, is the practice of tracking and managing changes to software code [6]. This worked well throughout the project and gave us the opportunity to have the project code neatly organized, as opposed to having different code projects locally on each of our computers. Version control through GitHub enables us to get a better understanding of the code others wrote. GitHub gives us the opportunity to put rules on branches. We put a rule on the main branch, that said you cannot merge from a branch into the main branch without having two additional group members

to approve of the code. Since the project would end up with multiple group members working on it at the same time, it could quickly become entangled and complicated. Therefore we used GitHub, and create branches for each increment in the backlog, thus keeping the main branch protected. Furthermore, if anything goes wrong on the main branch, it is easily undone by rolling back to an older commit. All code that is displayed in this report, is placed inside a GitHub repository [13].

Another kind of version control we use is Overleaf. Overleaf is a collaborative cloud-based  $\text{\LaTeX}$  editor used for writing, editing, and publishing scientific documents [49]. We used  $\text{\LaTeX}$  & Overleaf throughout their studies and it was therefore natural for us to write our report in Overleaf. Overleaf essentially does version control by itself. Every 5 minutes it makes a git commit with every file, and we are able to review changes since the beginning of the project.

### 2.6.3 Supervisor meetings

The supervisor meetings are weekly meetings that are either physical in the group room or online. Throughout the week between the meetings, we write everything down that we have questions for, and what we would like feedback on in the report by giving a reading guide, then 48 hours before the meeting an agenda, and these questions and the reading guide, and the report are sent to the supervisor.

At the supervisor meetings, one group member takes notes and another goes over the agenda to always have a structure. These roles switch every supervisor meeting so everyone gets to talk.

### 2.6.4 Additional tools

Some additional tools that we use are Discord, Outlook.com, and ChatGPT.

- All internal communication goes through a Discord server. This provides us with the ability to divide different sections into smaller chat channels to create a better overview.
- All external communication is handled through outlook.com, which includes supervisor emails, and study secretary.
- We used AI assistance in the development of the compiler, at no point is ChatGPT a co-writer in the report. We agreed on how the use of ChatGPT should be, and that it should only be to get some advice on specific questions and guidance.



# Chapter 3

## Language Design of PEAK+

This chapter aims to explore the theoretical foundations and practical considerations involved in designing the PEAK+ programming language. This chapter will provide a comprehensive analysis of the various aspects of language design. It will include the language criteria that guide the design process, the requirements that must be considered, and the syntax design, which includes Context-Free Grammars (CFGs) and Extended Backus-Naur Form (EBNF). Additionally, we will examine the static semantics of programming languages, including Abstract Syntax Trees (ASTs), scope rules, type rules and operational semantics.

### 3.1 Language Criteria

To begin, we will discuss the language criteria for PEAK+ that serve as the guiding principles for designing a programming language. These criteria include readability, writability, and reliability. Each of these criteria will be explored in detail. Benefits and trade-offs associated with each will be discussed in relation to PEAK+.

#### 3.1.1 Readability

Based on the analysis of existing programming languages in section 2.3.4, high readability is important when designing a language aimed towards beginner programmers. This readability can be achieved by constructing a syntax that relates to concepts and terminology that beginner programmers are already familiar with from their education. Based on our own experiences, having a readable syntax that relates to known concepts, makes the language syntax more memorable, making the programming concepts easier to grasp and relate to.

As stated in the analysis in section 2.3.4, both Scratch and Quorum have high simplicity and use concepts and terminology related to the English language. Quorum

focuses on readability over writability whereas Python does not. As PEAK+ should help beginner programmers with transitioning to some common industry standards, PEAK+ has been placed between Python and Quorum, but closer to Quorum on the scale in figure 3.1. This is due to the focus on readability, as people who have not programmed before, should be able to better recognize the constructs of PEAK+ . But since PEAK+ should be related to common industry languages, constructs such as statement terminators, and scope declaration should be equivalent to these languages.



**Figure 3.1:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing readability, including PEAK+ .

### 3.1.2 Writability

The writability of PEAK+ is lower since the language should relate to concepts and terminology from high school education. Therefore the language syntax should resemble the English language and math learned in high school. The purpose of PEAK+ is to enable an easy learning curve of the fundamental concepts and implementations used by many of the common industry programming languages. Furthermore, as PEAK+ is intended to be an introductory language, a focus of the language will not be creating large and optimized applications.

That being said, one goal of PEAK+ is to introduce beginner programmers to the use of methods and procedures, which supports a level of abstraction within the language. Because methods are a core principle within programming, they will be implemented, in a manner that supports readability.

On the scale displayed in figure 3.2, PEAK+ has been placed between C and Quorum, but closer to C. This is due to the fact that the language will have low expressivity, limited support for abstraction (no object-oriented constructs), and a limited amount of data types. Like Quorum, the focus of this language is more on readability in preference to writability. Like Quorum, the goal will be for beginners to be able to quickly recognize and understand the language, and therefore most reserved keywords will not be abbreviations like in many languages.



**Figure 3.2:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing writability, including PEAK+ .

### 3.1.3 Reliability

The importance of reliability is to ensure that the language does what the programmer specifies it to do, within the boundaries of the language. In PEAK+ , there are certain standards that need to be upheld within reliability, in order to cater to a beginner programmer.

The language will use static type checking, which should increase the reliability compared to dynamic type checking since the programmer will have to make sure types are appropriately used. On top of this to further simplify the language and improve reliability, aliasing will be restricted by not allowing pointers in the language. From our personal experience, pointers are a tricky programming construct to understand and use, and it is therefore inappropriate to allow aliasing in a language made for beginner programmers. As PEAK+ will have limited or restricted aliasing and no exception handling, it is placed between C and Quorum on the reliability scale in figure 3.3.



**Figure 3.3:** The programming languages C, Python, Quorum, and Scratch placed on a scale showing reliability, including PEAK+ .

## 3.2 The Paradigm of PEAK+

Because the goal is to make a beginner-friendly programming language, which aims to make it easier for its users to transition to industry standard languages, the chosen paradigm for PEAK+ is an imperative and procedural paradigm. They have been chosen because, in our own experience, it is important to learn basic programming concepts first like defining basic programming instructions and scopes. An object-

oriented approach is deemed too advanced and unnecessary, as we only want to include basic concepts such as functions, control structures, and some basic data types. The same reasons apply to why the functional paradigm has not been chosen. The paradigms mentioned here have been previously explained in section 2.3.2

### 3.3 Requirements

In order to develop a programming language for beginner programmers, requirements must be set. The requirements for PEAK+ are displayed in table 3.1.

Priority	ID	Requirements
Must Have	M1	Declaration and assignment of number, decimal, text, and boolean variables.
	M2	Display output and take keyboard input in the console.
	M3	Basic iterative and selective control structures
	M4	Basic arithmetic & logical operations.
	M5	Syntax which resembles concepts and terminology used in high school education.
	M6	Error messages at compile time.
Should Have	S1	A foreach loop.
	S2	String concatenation.
	S3	Declaration and assignment of lists.
	S4	List helper functions like add and remove from the list.
Could Have	C1	Support for abstraction in the form of methods
Won't Have	W1	Exception Handling

**Table 3.1:** MoSCoW requirement table.

The requirements are designed in the light of the target group, beginner programmers. By looking at several different programming tasks given in different beginner programming courses, more features necessary for a beginner language have been settled, leading to the requirements for PEAK+ . The tasks that are examined, are from the imperative programming course in the first semester of our own education[27], Python practice exercises from python.org[29] and general code challenges from codecademy.com [23]. Besides examining programming tasks, the requirements are also defined with the language criteria in section 3.1 in mind.

The language has to implement fundamentals for a programming language, including declarations and assignments of variables (M1) and the ability to generate an output to the console and read keyboard inputs (M2). Another fundamental requirement is the use of different types of control structures, both selective and iterative.



Examples of beginner tasks include repeating some code  $x$  times or as long as a condition is satisfied. In relation to the decision stated in section 3.1.1 about PEAK+ prioritizing readability, the control structures in PEAK+ have to be concrete and easy for the beginner programmer to understand. In (M3) the definition of basic control structures includes the implementation of if-else condition blocks, a counting loop and a conditional loop.

To ensure the readability of PEAK+ , the syntax has to resemble concepts and terminology used in high school, e.g. the use of fully spelled words and basic math symbols, leading to requirements M4 and M5. In addition to this, requirement S2 about string concatenation is implemented, as context to compound words.

Since PEAK+ is a beginner language, it is important that there are error messages at compile time. This relates to requirement M6.

Different ways of storing related variables are also common in the tasks. For example, saving several variables in a list, deleting an entry in a list, and getting a variable from an entry in the list, leading to requirements S3 and S4.

Scopes and Functions are also relevant topics. This entails functions which have the possibility of receiving an input and returning an output. This relates to requirement C1.

PEAK+ will not have exception handling. This feature is deemed as nice to have, but the aforementioned features have been deemed more important due to having a higher impact on the simplicity of PEAK+ .

## 3.4 Syntax Design

To create the language specification for PEAK+ , the first step is defining the syntax. Syntax describes the grammar and structure of valid programs in a language. The syntax design is followed up by operational semantics and type rules in section 3.5.3. This section will cover theoretical sections about context free grammar and Backus-Naur Form, followed up by the context free grammar for PEAK+ .

### 3.4.1 Context-Free Grammars

A context free grammar (CFG) is a way to formally represent the syntax of a language using symbols and rules. They describe all possible programs that are syntactically correct in the language. A CFG is specified as having a finite set of terminal and

non-terminal symbols as well as a start symbol and a finite set of production rules [36]. An example of a CFG is displayed in listing 3.1.

```

1 | Start -> A B $
2 | A -> A a | epsilon
3 | B -> b

```

**Listing 3.1:** An example of a CFG. The language accepts strings of zero or more A's followed by a 'b'

A terminal symbol is a symbol from the language's alphabet[36]. These are always written in lowercase letters. The non-terminal symbols are placeholders for terminal symbols or patterns of terminal symbols. These are written with the first character in uppercase. Non-terminal symbols are derived into other terminals and/or non-terminals until they become exclusively terminal symbols. This is done using production rules [36]. Reviewing the CFG in listing 3.1, it can consists of several production rules. A production rule describes a possible way, in which a non-terminal can expand. The LHS (left-hand-side) of the arrow in each production rule consists of a non-terminal, that is to be derived. The rules to derive the nonterminal are given by the RHS (right-hand-side). There can be several different ways a non-terminal can be rewritten. The different production rules for a non-terminal are separated by a line '|' and can contain combinations of terminals and non-terminals as well as having the possibility of being epsilon ( $\epsilon$ ), meaning an empty string. Furthermore, the first non-terminal in the LHS of the CFG is the start symbol [36].

The terminals in the example in listing 3.1 are *a*, *b*, and *\$*. The start symbol is 'Start' which can be derived into *A B* followed by the *\$*, end of input stream symbol. The process continues until no non-terminal remains in the derived string. In listing 3.1, *A* is either derived to epsilon at the beginning or recursively derived into *A a* until the nonterminal *A* derives into epsilon. Furthermore, *b* is derived from the nonterminal *B*.

A CFG can be represented using BNF (Backus-Naur Form) or EBNF (Extended Backus-Naur Form). The set of strings that a CFG can define is called its language. A string that belongs to the language of the CFG, can be considered a valid string for the CFG [36]. The notation of a CFG can vary, but in this report, productions for non-terminals are shown using an arrow ' $\rightarrow$ '.

### 3.4.2 Backus-Naur Form

BNF (Backus-Naur Form) and EBNF (Extended Backus-Naur Form) are two ways to represent a CFG[36]. In PEAK+ , BNF has been used to represent the CFG. An advantage of using BNF over EBNF is that BNF is simpler in the way that it only uses a small set of metacharacters[19] and rules which are easy to learn and understand. A disadvantage of using BNF instead of EBNF is that BNF uses a larger amount of productions to describe the same grammar as EBNF. This can make the grammar more difficult to read than EBNF which can especially be a problem when creating

complex grammar.

EBNF extends BNF by using elements similar to regular expressions, such as "[" to indicate an optional element or "{" to indicate repetition zero or more times[36].

Overall, the advantages of using BNF are that it is simpler to learn and understand, as it has fewer metacharacters. Furthermore, most CFG tools do not recognize EBNF. For these reasons, we will write the CFG of PEAK+ using BNF.

### 3.4.3 Context-Free Grammar for PEAK+

The PEAK+ programming language focuses on being simple in its syntax and data structures, while also implementing common principles from the programming industry for an easier transition from a beginner language to a language such as C or C#.

In order to make the language easier to understand for new programmers, only 5 data types will be implemented. Three of these data types are primitive, being *number*, *decimal* & *boolean*. The other two are non-primitive data types being *text* & *list*. The data type *number* is an integer, and *decimal* is a floating point number. To increase readability, the *boolean* is a boolean value, having either the value *true* or *false*. Conditional operators taught in high school mathematics are included, such as ' $<$ ' and ' $<=$ '. The logical *or* and the logical *and* are written in text as these operators are not introduced in the high school curriculum. *text* is a string, which is a collection of characters, and *list* is a collection of any of the previous types.

**Overall program structure:** The overall structure of the PEAK+ programming language can be seen in listing 3.2. A program in PEAK+ consists of a list of commands, which can be further broken down into either a statement or a declaration. A declaration consists of a type, an id, and an optional assignment, whereas a statement can be broken further into multiple different productions. Like in many other languages, a semicolon ';' is used as a statement terminator, making it clear when a statement ends. Using semicolons as a statement terminator instead of, for example, a newline, also allows the programmer to span a long statement across multiple lines, making it more readable. This makes it so anyone reading the code is able to read a long statement without having to scroll left and right in their editor. It also allows for multiple statements on a single line.

1	Program	-> Cmds
2	Cmds	-> Cmd Cmds   EPSILON
3	Cmd	-> Stmt   Dcl
4	Dcl	-> Type Id Ass ;
5	Ass	-> = Expr   EPSILON

```

6 Stmt          -> Id = Expr ; | CtrlStrct | ListStmt ; |
   FuncDef | FuncCall ;

```

**Listing 3.2:** Overall structure of PEAK+ CFG

**Control structures:** The control structures of PEAK+ can be seen in listing 3.3. A control structure in PEAK+ can be either a selective control structure (if statement), or an iterative control structure (loop). This can be seen from the Non-terminal *CtrlStrct* in listing 3.3, which can derive from the productions *IfStmt* or *Loop*.

When deciding on control structures, the standard if-else blocks are used to support readability. However, in regards to loops, it has been decided to generalize all loops by using the reserved word *repeat*, instead of the standard *for* and *while*. This is based on the analysis of the Quorum programming language, which simplifies the standard loop structures.

```

1 CtrlStrct -> IfStmt | Loop
2 IfStmt -> if ( Expr ) Block ElseIfStmt
3 ElseIfStmt -> else if ( Expr ) Block ElseIfStmt | Else |
   EPSILON
4 Else -> else Block
5 Loop -> repeat Loops
6 Loops -> LoopStmt | WhileStmt | ForeachStmt
7 LoopStmt -> ( Expr ) times Block
8 WhileStmt -> while ( Expr ) Block
9 ForeachStmt -> for each ( Type Id in Id ) Block

```

**Listing 3.3:** Control structure of PEAK+ CFG

**If statement:** An if statement in PEAK+ starts with the terminal keyword *if*, followed by an *Expr* nonterminal encapsulated by parentheses, a *Block* and an optional *ElseIfStmt* (See line 2 in listing 3.3). An *ElseIfStmt* production has the same structure as an *IfStmt*, but with the terminals *else if* instead of just *if*. Furthermore, it derives to the nonterminal *Else* and *EPSILON*, making the *ElseIfStmt* optional as it has a production that derives to nothing. An advantage of this structure is that it allows for checking multiple expressions and even nesting if statements, which makes writing complex conditional logic easier. However, this can also turn into a disadvantage, as deeply nested code may confuse the programmer as it will be harder to read and understand.

**Loops:** A loop in PEAK+ will always derive to the *repeat Loops* production as can be seen on line 5 in listing 3.3. This means that any loop will consist of the terminal keyword *repeat*, followed by one of the different loop types defined in listing 3.3. Having the structure consist of the terminal *repeat* followed by a loop type, provides a consistent way to define and identify loops in PEAK+. A disadvantage of this is that it adds an extra word to the language every time a loop is created, which gives the language slightly less writability. However, as previously mentioned in section 3.1.1,

readability is prioritized above writability in PEAK+ .

PEAK+ has three kinds of loops being a counting loop, a while loop, and a for each loop. The counting loop has the following syntax: "repeat (x) times". This is a simple loop, which is intended for beginners to get familiar with loops in their simple form. The loop requires an integer number or variable of the type *number* to determine how many times the loop should run. The while loop functions as in any other programming language, where it runs as long as a condition is fulfilled. The while loop is deemed necessary for the language, as it is able to achieve more than other loop constructs. Finally, the *for each* loop has been selected for the PEAK+ language in order to provide a simple way of looping through each element in a collection. This is simpler than a normal *for* loop as a *for each* loop only requires an initializer expression and the collection to loop through. In comparison, a *for* loop requires both an initializer expression(s), a continue expression, and an update expression(s). A *for*-loop syntax may be difficult for beginner programmers to remember, so, therefore, a *for*-loop control structure will not be included in the language.

**Functions:** The function definition of PEAK+ can be seen in listing 3.3. A function definition in PEAK+ starts with the terminal *function* followed by an *Id*, a list of parameters, the function return type, and finally the function body.

```

1 FuncCall -> call Id ( ArgList ) | call output ( ArgList ) |
    call Type input ( ArgList )
2 FuncDef -> function Id ( ParamList ) FuncReturn Block
3 FuncReturn -> return FuncReturnType
4 FuncReturnType -> Type | nothing
5 ParamList -> Param ParamTail | EPSILON
6 ParamTail -> , Param ParamTail | EPSILON
7 Param -> Type Id
8 ArgList -> Expr ArgTail | EPSILON
9 ArgTail -> , Expr ArgTail | EPSILON

```

Listing 3.4: Function call & declaration of PEAK+ CFG

The *function* terminal is used in order to clearly define that a function is being defined, increasing readability. The return type is placed after the parameter list. This choice is made in order to obtain readability by separating the return type from the start of the declaration. Explicitly having to use the return keyword along with the function keyword in a function definition provides less writability, but more readability. The *FuncReturnType* nonterminal derives to either a *Type*, or *nothing*. The *nothing* return type is equivalent to the void return type in other languages. The *nothing* keyword is used instead, due to the fact that returning *nothing* seems more intuitive than *void* for beginner programmers.

The *FuncCall* nonterminal has one production, starting with the terminal *call*, fol-

lowed by an *Id* and an *ArgList* enclosed by parentheses. The *call* keyword is used to once again increase readability, as it helps clearly define when a function is called.

**Input & output:** The definition of input and output for PEAK+ can be seen in listing 3.5. The *output* terminal is used as a way to write / print something to the console. Whereas the *input* terminal is a way to read input from the user.

```

1 FuncCall -> call Id ( ArgList ) | call output ( ArgList ) |
    call Type input ( ArgList )
2 .
3 .
4 .
5 ArgList -> Expr ArgTail | EPSILON
6 ArgTail -> , Expr ArgTail | EPSILON

```

Listing 3.5: Input & output of PEAK+ CFG

The input and output definition starts with the non-terminal *FuncCall* which can derive to either *call output* or the *call Type input*. As can be seen in listing 3.5, both the terminal *input* and the terminal *output* takes the non-terminal *ArgList* as a parameter. The non-terminal *ArgList* derives into an expression followed by the next non-terminal *ArgTail*. In the CFG, the input may contain multiple expressions. However, in the compiler, the *ArgList* for the input function should only be able to contain a single expression, being a variable of the type defined after the *call* terminal. The reason for using *ArgTail* instead of *Id* in the input even though the input is essentially an id, is to keep consistency among functions, following the general function definition structure.

**Lists:** A list in PEAK+ starts from the non-terminal *Stmt* which then derives to the non-terminal *ListStmt* production as can be seen in listing 3.6.

```

1 Program -> Cmds
2 Cmds -> Cmd Cmds | EPSILON
3 Dcl
4 Dcl -> Type Id Ass ;
5 Ass -> = Expr | EPSILON
6 Stmt -> Id = Expr ; | CtrlStrct | ListStmt ; | FuncDef |
    FuncCall ; | CommentStmt ; | return Type ;
7 .
8 .
9 .
10 ListStmt -> ListOpr | ListOprExpr
11 ListOpr -> Id : Add ( ArgList ) | Id : Replace ( ArgList )
12 ListOprExpr -> Id : IndexOf ( Expr ) | Id : ValueOf ( ArgList )

```

Listing 3.6: Lists in PEAK+ CFG

The non-terminal *ListStmt* can derive either the non-terminal *ListOpr* or the non-terminal *ListOprExpr*. A *ListStmt* will always start with the terminal *Id*, which refers to the *Id* of the list being operated on. If going from the non-terminal *ListOpr* it is possible to derive to the terminal *Add*, to add an element to the list, or derive to the terminal *Replace* in order to replace an element in the list.

The non-terminal *ListStmt* can also derive from the non-terminal *ListOprExpr*. The *ListOprExpr* can derive into the terminal *IndexOf* to get the index of a position in the list, or the terminal *ValueOf* to get the value stored in a specific position in the list.

### 3.4.4 Operator Precedence

Operator precedence defines the order in which operators are evaluated in an expression. The operator precedence for PEAK+ follows the basic mathematics rules, this can be seen in table 3.2. The highest precedence in PEAK+ follows the mathematical PEMDAS operators which are: parenthesis, exponent, multiplication, division, addition & subtraction[21]. The exponent operator is not included in the language, as it is achievable through multiplication, and therefore is not strictly necessary. Following the PEMDAS operators come the relational operators, such as "*greater than*" and Logical operators such as "*and*". These have the lowest precedence in PEAK+ . This is because the logical operators are typically used in conjunction with operators with higher precedence to combine or negate a result.

Precedence	Operator	Description
1	( )	Parenthesis
2	$x * y$	Multiplication
	$x / y$	Division
3	$x + y$	Addition
	$x - y$	Subtraction
4	$x > y$	Greater than
	$x \geq y$	Greater than or equal to
	$x < y$	Less than
	$x \leq y$	Less than or equal to
5	$x \text{ is } y$	Is equal to
	$x \text{ is not } y$	Is not equal to
6	$x \text{ and } y$	Logical and operator
7	$x \text{ or } y$	Logical or operator

**Table 3.2:** Operator Precedence for PEAK+

### 3.4.5 Scope Rules

Scope rules regulate the visibility of identifiers in a programming language. They relate every applied occurrence of an identifier to a binding occurrence. The binding occurrence is defined as the instance where an identifier is declared initially, and the applied occurrence is any point at which the identifier is used as part of an expression [11]. Both of these occurrences can be seen in listing 3.7. Different scope rules give rise to different interpretations of the same piece of code [18].

```
1 | number x = 1; // Binding Occurrence
2 |
3 | number r = x + 1; // Applied Occurrence
```

**Listing 3.7:** Example of a binding occurrence and an applied occurrence of the identifier *x* in PEAK+

There are two kinds of scope rules: Dynamic and static. Dynamic scope rules employ the bindings known when the procedure is called and are based on the program's runtime. Static scope rules employ the bindings known when the procedure is declared and are based on the structure of the program[18]. PEAK+ will use static scope rules. The main reason for this choice is to strengthen the readability of PEAK+ . PEAK+ uses the nested block structure. A nested block structure means, that a program can be subdivided into several disjoint blocks with an infinite amount of blocks inside each block. A block is an area of text in the program that corresponds to some kind of boundary for the visibility of identifiers. The essence is that several local definitions of a single identifier may occur in different blocks (but not in the same block) [11]. The reason for this is to enable flexibility for the beginner programmer. A nested block structure is also a common industry standard, which prepares the programmer for those languages. The scope rules for PEAK+ are:

- A *block* in PEAK+ can be a *function* or control structure, like *if*-statements. PEAK+ also contains special characters like *repeat* and *repeat for each* that iterate over a block a defined amount of times.
- All blocks are defined by using the prefix for one of the above and encapsulating the content between two curly brackets, this can be seen in listing 3.8.
- A local variable declared inside a block can only be applied within that block and any nested blocks within it. Variables declared globally can be used anywhere.
- Variable shadowing is allowed within nested blocks.
- Functions have to be declared in the global scope, and cannot be redeclared in any scope.



```
1 | function Sum(number x, number y) return number
2 | {
3 |     return = x + y;
4 | }
```

**Listing 3.8:** Example of a Block

## 3.5 Semantics

To settle the behavior of the language, the next step in the language specification for PEAK+ is defining the semantics. This section covers abstract syntax, type rules, and operational semantics. The subsections will be structured with theoretical knowledge, followed up by the constructs from PEAK+ .

The source of the knowledge described in this section is from the course Syntax and Semantics and will be cited in the paragraph when appropriate. Definitions that are directly from the course material, will also have standard citations, without quotation marks.

### 3.5.1 Abstract Syntax

Abstract syntax defines the structure of valid programs in the language but in a more abstract and simplified form. It is for example not concerned with parsing and ambiguity problems and it does not have to comply with the standard formats for parsing algorithms[7]. The abstract syntax is defined from a combination of a set of syntactic categories, which are seen in table 3.3. A set of formation rules is defined for these syntactic categories, which can be seen in table 3.4 [7].

$n$	$\in$	<b>Num</b>	Natural Numbers
$d$	$\in$	<b>Dec</b>	Rational Numbers
$txt$	$\in$	<b>Text</b>	Strings
$\mathbb{T}, \mathbb{F}$	$\in$	<b>Bool</b>	Boolean values
$x$	$\in$	<b>Var</b>	Variable names
$L$	$\in$	<b>List</b>	Lists
$f$	$\in$	<b>Func</b>	Function names
$B$	$\in$	<b>BTypes</b>	Base Types
$T$	$\in$	<b>Types</b>	All Types
$e$	$\in$	<b>Exp</b>	Expressions
$S$	$\in$	<b>Stm</b>	Statements
$D_V$	$\in$	<b>DecVar</b>	Variable declarations
$D_F$	$\in$	<b>DecFunc</b>	Function declarations

**Table 3.3:** Syntactic categories of PEAK+

The syntactic categories in table 3.3 represent all elements in PEAK+ . To represent an element of the syntactic categories we use meta-variables, which are the characters in the first column in the table. E.g the syntactic category **Num** in the first row, represents any natural number in PEAK+ and the meta variable  $n$  is used in the abstract

syntax to represent an arbitrary number. Lists are defined using  $L$  and contain any amount of expressions in an ordered format.

$B$	$::=$	number   decimal   text   boolean
$T$	$::=$	$B$   list( $B$ )
$e$	$::=$	$n$   $txt$   $\mathbb{T}$   $\mathbb{F}$   $d$   $x$   $e_1 + e_2$   $e_1 - e_2$   $e_1 * e_2$   $e_1 / e_2$   $e_1$ is $e_2$   $e_1$ is not $e_2$   $e_1 > e_2$   $e_1 \geq e_2$   $e_1 < e_2$   $e_1 \leq e_2$   $e_1$ and $e_2$   $e_1$ or $e_2$   $(e_1)$   $x:\text{valueOf}(n)$   $x:\text{indexOf}(e)$
$S$	$::=$	$S_1 ; S_2$   $D_V$   $D_F$   $x = e$   $x:\text{add}(e_1, \dots, e_k)$   $x:\text{insert}(e, n)$   $\text{if}(e) \{ S \}$   $\text{If}(e) \{ S_1 \} \text{ Else } \{ S_2 \}$   $\text{repeat}(n) \text{ times } \{ S \}$   $\text{repeat while}(e) \{ S \}$   $\text{repeat for each}(B \ x \text{ in } x)$   $\text{call } f(e_1, \dots, e_k)$   $x = \text{call } f(e_1, \dots, e_k)$   $\epsilon$
$D_V$	$::=$	$Tx = e ; S$   $Tx ; S$   $\epsilon$
$D_F$	$::=$	function $f(T_1 \ x_1, \dots, T_k \ x_k)$ return $T \{ S_1 ; \text{return } e \}; S_2$   $\epsilon$

**Table 3.4:** Abstract Syntax formation rules

The formation rules define the structure of the syntactic categories and are written in the same format as production rules in BNF, explained in section 3.4.1 [7]. In the first column in table 3.4, it is denoted that an expression  $e$  in PEAK+ can be one of the related right-hand side formation rules.

The syntactic categories **Num**, **Dec**, **Text**, **Bool**, **Var** and **Func** do not have formation rules as they are trivial. For example, **Text** is a string of characters, as explained in section 3.4.1.

### 3.5.2 Type Rules

To prevent a program to execute with errors, a type system is important. When compiling a program, the type systems' purpose is to catch type errors before the program executes. When a program executes without type errors it is called *well-typed*. To know when a program is well-typed, we define type rules. The type rules of a language define how the different types in the language are allowed to be used correctly.

In the type system, values of variables are kept in an environment. As variables get declared or reassigned, they are updated in the environment [9]. A type environment keeps track of the types of variables. A type environment is a partial function that maps variables to types,  $E : \mathbf{Var} \cup \mathbf{Func} \rightarrow \mathbf{Types}$  [9]. When the state is changed in the environment it can be denoted as  $E[x \mapsto T]$ . The formal definition of an update to the environment is the following function:

$$(E[x \mapsto T])(y) = \begin{cases} E(y) & y \neq x \\ T & y = x \end{cases} \quad (3.1)$$

To define a type system, the first step is to describe a collection of formal statements called *judgments* [9]. The two types of judgments that are used in the type system, are seen in definition 3.2[9].

$$\begin{aligned} E \vdash e : T \\ E \vdash S : ok \end{aligned} \quad (3.2)$$

In definition 3.2 it can be read as, within the environment  $E$  some assertion of a syntactic category is made. In the first judgment, we assert that an expression  $e$  has a certain type, and in the second judgment, we assert that a statement  $S$  is well-formed in accordance with the formation rules. The judgment will be derived from the type rules defined in the type system. Such derivations can be used to describe this system. Definition 3.3 is the template for how the derivations will be presented[9].

$$\frac{E_1 \vdash \alpha_1, \dots, E_n \vdash \alpha_n}{E \vdash \alpha} \quad (3.3)$$

The  $\alpha$  represents a syntactic category implementation and can be any of the two mentioned judgments in definition 3.2. Based on the explained rules, the type system of PEAK+ can be formally described.

The standard derivation for a variable can be performed when a variable reaches its final possible derivation. In definition 3.4 the derivation checks if a variable's type matches what is in the environment.

$$(VAR_{TS}) \frac{E(x) = T}{E \vdash x : T} \quad (3.4)$$

As mentioned in section 3.4.5, PEAK+ allows variable shadowing, meaning a variable can be redeclared in an inner scope. Furthermore, type errors can occur when expressions are made with types that are not compatible. Firstly, the type rules for expressions can be seen in definitions 3.5, 3.6, 3.7, and 3.8.

$$(NUM_{*TS}) \frac{E \vdash e_1 : number \quad E \vdash e_2 : number}{E \vdash e_1 * e_2 : number} \quad \text{where } * \in \{+, *, -\} \quad (3.5)$$

$$(DEC_{*TS}) \frac{E \vdash e_1 : decimal \quad E \vdash e_2 : decimal}{E \vdash e_1 * e_2 : decimal} \quad \text{where } * \in \{+, *, -, /\}, \text{ and } e_2 \neq 0 \quad (3.6)$$

$$(TXT_{TS}) \frac{E \vdash e_1 : \text{text} \quad E \vdash e_2 : \text{text}}{E \vdash e_1 + e_2 : \text{text}} \quad (3.7)$$

$$(BOOL_{TS}) \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 \star e_2 : \text{boolean}} \quad \text{where } \star \in \{>, <, \geq, \leq, \text{is}, \text{is not}\} \quad (3.8)$$

Addition can occur both in an arithmetic sense between **Num** and **Dec**, and in the form of string concatenation. However, string concatenation can only occur between two **Text** variables. **Num** & **Dec** is mainly split up to avoid conflicts when using a floating-point value as a list index. In definition 3.5 the star cannot be a division symbol. If two numbers are divided, the type will convert to a decimal. It is also not allowed to divide by zero. The rest of the type rules for expressions in PEAK+ are found in appendix C

The next syntactic category to define type rules for is variable declarations (meta variable  $D_V$  in table 3.4). These formation rules allow the declaration and assignment of variables with type **T**. The derivations in definitions 3.9 and 3.10, describe the type rules. In  $(DEC_{TS})$  the  $x \notin \text{dom}(E)$  defines that the variable  $x$  is not previously in the domain of the environment  $E$ . This is necessary because a variable declaration creates an instance of a new variable, and there cannot be any existing variables with the same name in the same scope.

$$(DEC_{TS}) \frac{x \notin \text{dom}(E) \quad E \vdash e : T}{E \vdash T \ x = e : ok} \quad (3.9)$$

$$(ASS_{TS}) \frac{E(x) = T \quad E \vdash e : T}{E \vdash x = e : ok} \quad (3.10)$$

The formation rule of  $\epsilon$  is shown in definition 3.11, since it can appear in multiple formation rules.  $\epsilon$  simply derives to a well-formed derivation.

$$(EMPTYSTM_{TS}) \frac{}{E \vdash \epsilon : ok} \quad (3.11)$$

The last syntactic category which will be shown here is the function declaration. In definition 3.12 the function has a return type  $T$ . A function declaration is well-formed when the functions block and return value is well-formed in the environment where the variables  $x_1, \dots, x_k$  have the type  $T_1, \dots, T_k$ . Furthermore, the statement  $S_2$  which is executed after the function call is well-formed in the updated environment.

Definition 3.13 is almost identical to definition 3.12. The change here is, that for a function without a return type, we do not handle the return type in the premises.

$$(FUNCDEC1_{TS}) \frac{E' \vdash S_1 : ok \quad E' \vdash e : T \quad E[f \mapsto (x_1 : T_1, \dots, x_k : T_k \rightarrow T)] \vdash S_2 : ok}{E \vdash \text{function } f(T_1 x_1, \dots, T_k x_k) \text{ return } T \{ S_1; \text{return } e \} ; S_2 : ok} \quad (3.12)$$

$$\text{where } E' = E[x_1 : T_1, \dots, x_k : T_k]$$

$$(FUNCDEC2_{TS}) \frac{E[x_1 : T_1, \dots, x_k : T_k] \vdash S_1 : ok \quad E[f \mapsto (x_1 : T_1, \dots, x_k : T_k)] \vdash S_2 : ok}{E \vdash \text{function } f(T_1 x_1, \dots, T_k x_k) \text{ return nothing } \{ S_1; \text{return} \} ; S_2 : ok} \quad (3.13)$$

There are two ways to handle a function call. As displayed in the formation rules in table 3.4, it can either be *call*  $f()$  or  $x = \text{call } f()$ . In definition 3.14, the left-most derivation checks if there exists a function with the name  $f$  which associates to the formal parameters and return type listed. Thereafter each argument given is crosschecked with the appropriate type to make sure they match. If so the derivation is successful. In definition 3.15 one more premise is added to the derivation, where we check that the variable  $x$  in the environment has type  $T$ . The rest of the type system can be found in Appendix C.

$$(FUNCCALL1_{TS}) \frac{E(f) = (x_1 : T_1, \dots, x_k : T_k \rightarrow T) \quad E \vdash e_i : T_i \quad \text{where } i = 1 \dots k}{E \vdash \text{call } f(e_1, \dots, e_k) : ok} \quad (3.14)$$

$$(FUNCCALL2_{TS}) \frac{E(f) = (x_1 : T_1, \dots, x_k : T_k \rightarrow T) \quad E(x) = T \quad E \vdash e_i : T_i \quad \text{where } i = 1 \dots k}{E \vdash x = \text{call } f(e_1, \dots, e_k) : ok} \quad (3.15)$$

### 3.5.3 Operational Semantics

Before, the structural operational semantics for PEAK+ can be defined, there is some theory that needs to be explained. Firstly, several forms of semantics exist. Semantics that this report will not cover includes denotational semantics, which focuses on the effect of executing programs by assigning a denotation to them, and axiomatic semantics, which focus on the correctness property of programs [7]. The semantics that are important to look at in this context are operational semantics. Operational semantics are descriptions of the meaning of a program, precisely telling you how it should execute. To define the operational semantics, we use Transition Systems.

### Transition Systems

A Transition System is a directed graph and can be defined as a 3-tuple. The formal definition can be seen in definition 3.16 [7].

$$S = (\Gamma, \rightarrow, T) \quad (3.16)$$

The first symbol ( $\Gamma$ ) represents the set of configurations. Each configuration is a vertex of the graph and they represent the current state of the program. An example of this is to think of each vertex containing the known variables which have been declared at that point throughout execution. The arrow symbol ( $\rightarrow$ ) is the transition relation, formally written  $\rightarrow \subseteq (\Gamma \setminus T) \times \Gamma$ . Transitions are the edges of the graph, which act as a bridge between the configurations. From each configuration, you can take the possible transitions to get to another configuration. Each transition is written as a pair  $(a, b)$  where  $a$  can transition to  $b$ . The last symbol ( $T$ ) is the set of terminal configurations. A terminal configuration is a configuration that you cannot transition away from and it is a subset of  $\Gamma$ , written  $T \subseteq \Gamma$ . [7]

We took a structural approach to the operational semantics a.k.a (SOS) because it allows us to define the operational semantics based on a few rules. Having SOS means that the transition system has to comply with the following three rules.

- Determine the format of the transition (all transitions must comply with the chosen format)[7].
- The configurations that are the source of a transition should be terms generated from the abstract syntax in table 3.4 [7].
- Define the transition relation by means of structural derivation rules[7].

The format for each syntactical group will be defined later in subsection 3.5.4 where the transition system for PEAK+ is explained.

### Big-Step and Small-Step Semantics

When deciding to define the operational semantics for a language, two methods can be used. **Big-Step** semantics and **Small-Step** semantics. Both of these semantic types can be expressed as the transition system defined in definition 3.16.

- **Big-Step** semantics (BS) express an entire computation of a given configuration's transition ( $\gamma \in \Gamma$ ) to its terminal configuration ( $t \in T$ ) [7].
- "**Small-Step** semantics (SS) express each transition as a single step of a larger computation. Terminal configurations characterize when the computation stops" [7].

Small-step semantics are often used when more detail about a derivation is required to understand what is happening. Often it is used in concurrency, where multiple executions in a program are happening at the same time[8]. For the sake of simplicity as well as the fact that PEAK+ does not have concurrency capabilities, the semantics will be described using Big-Step semantics.

To show the basic concept of Big-step semantics, an example of the arithmetic summation of two numbers is shown in definition 3.17. Here the format of the transition is  $a \rightarrow_{\text{Exp}} v$  where  $a \in \mathbf{Exp}$  and  $v \in \mathbb{Z}$ . The transition symbol  $\rightarrow_{\text{Exp}}$  can be read as "evaluates to an instance of the expression category" [7].

$$(PLUS_{BS}) \frac{a_1 \rightarrow_{\text{Exp}} v_1 \quad a_2 \rightarrow_{\text{Exp}} v_2}{a_1 + a_2 \rightarrow_{\text{Exp}} v_1 + v_2} \quad (3.17)$$

### Environment-Store Model

Choosing to write the semantics of a programming language in the format shown in definition 3.17 is not extensive enough for PEAK+ . It is unable to handle the scope rules set out for PEAK+ in section 3.4.5 since it assumes a monolithic-block structure. To handle this we use a concept known as the *Environment-Store Model*. This model acts as an addition to the semantics, where it describes how variables are actually bound during execution. Each variable is bound to the location of a storage cell, which holds the value of the variable as its content [8].

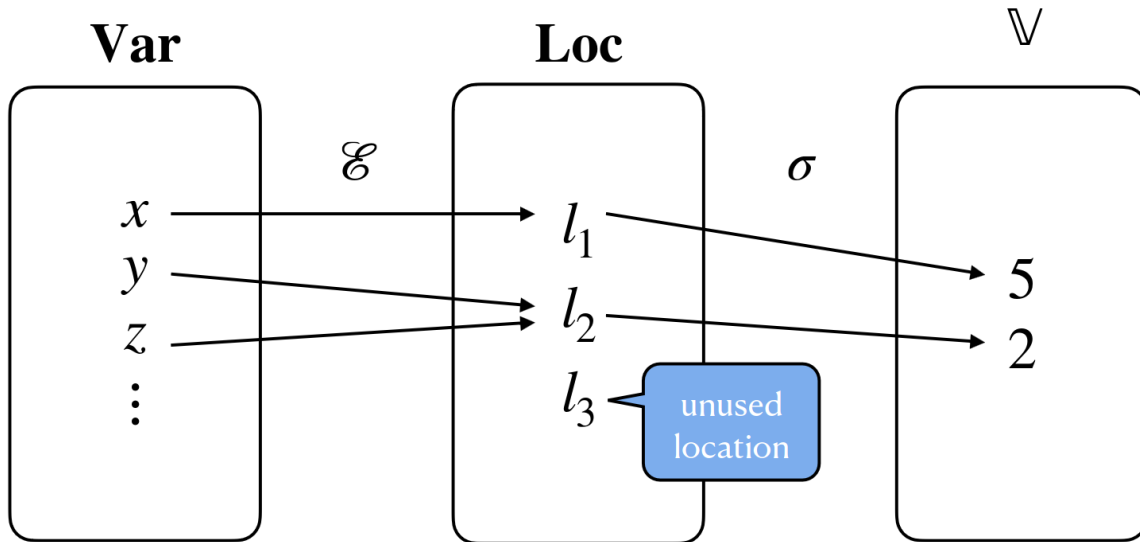


Figure 3.4: Visualization of Environment-Store Model [8]

In order to understand the semantics that is defined using the environment-store model, we need to have three definitions in place.



- **Variable Environment:** A partial function that tells which storage location each variable is bound to [8]. The set of variable environments will be defined as the following:

$$\begin{array}{c|c} \text{Single Instance} & \text{Set of Variable Environments} \\ \hline \text{Env}_v : \mathbf{Var} \rightarrow \mathbf{Loc} & \mathbf{Env}_V = \mathbf{Var} \rightarrow \mathbf{Loc} \end{array}$$

- **Function Environment:** Describes the content of function name bindings. Essentially the name of a function is bound to the statements and parameters inside of its block, as well as the variable environment given at that time, in order to keep the variable environment statically bound. The function environment will be denoted as the following:

$$\begin{array}{c|c} \text{Single Instance} & \text{Set of Function Environments} \\ \hline \text{Env}_f : \mathbf{Func} \rightarrow \mathbf{Stm} \times \mathbf{Var} \times \mathbf{Env}_V & \mathbf{Env}_F = (\mathbf{Func} \rightarrow \mathbf{Stm} \times \mathbf{Var} \times \mathbf{Env}_V)^* \end{array}$$

- **Store:** A partial function which describes the content of a storage cell corresponding to a given location [8]. It is important to note that this is only applicable in relation to variable environments. The store acts as the memory of the model. It will be noted as the following:

$$\begin{array}{c|c} \text{Single Instance} & \text{Set of all stores} \\ \hline \sigma : \mathbf{Loc} \rightarrow \mathbb{V} & \mathbf{Sto} = \mathbf{Loc} \rightarrow \mathbb{V} \end{array}$$

$$\text{Where } \mathbb{V} \in \mathbb{R} \cup \{\mathbb{T}, \mathbb{F}\} \cup \mathbf{Txt}$$

When each of these definitions are used in the following sections where the semantics of PEAK+ are explained, they will be referred to as the symbols written in each of their single instances. For the model to work, the abstract set  $\mathbf{Loc}$  of locations is assumed to exist, a single instance of a location is denoted by  $l$ . The function  $nxt(l) = 1 + l$ , can be called whenever the next available location needs to be retrieved[8].

### 3.5.4 Transition Systems for PEAK+

The necessary theory to understand the big-step semantics has now been described and we will explain the transition system for PEAK+ in this subsection. The semantics will be based on the formation rules in section 3.4. The chosen semantic examples are based on complexity and equivalence.

### Expressions

The format for expressions is the following  $\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v$ . The expression category will be derived first, since it is one of the fundamental levels of derivation, and is used in many of the other categories. The set of configurations for expressions in PEAK+ is that of any expression and variable type.

$$\Gamma_{\mathbf{Exp}} = \mathbf{Exp} \cup \mathbf{Num} \cup \mathbf{Dec} \cup \mathbf{Txt} \cup \mathbf{Bool}$$

The terminal configurations on the other hand are almost the exact set, but where all expressions have been evaluated to one of the base types. The terminal configurations can be formally described as the following definition

$$T_{\mathbf{Exp}} = \mathbf{Num} \cup \mathbf{Dec} \cup \mathbf{Txt} \cup \mathbf{Bool}$$

With this in mind, a couple of examples of how expressions are derived can be seen in the following definitions.

$$(VAR_{BS}) \frac{\sigma(Env_v(x)) = v}{\sigma \circ Env_v \vdash x \rightarrow_{\mathbf{Exp}} v} \text{ if } Env_v[x \rightarrow l] \text{ and } \sigma[l \rightarrow v] \quad (3.18)$$

$$(\star_{BS}) \frac{\sigma \circ Env_v \vdash e_1 \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash e_2 \rightarrow_{\mathbf{Exp}} v_2}{\sigma \circ Env_v \vdash e_1 \star e_2 \rightarrow_{\mathbf{Exp}} v_1 \star v_2} \quad (3.19)$$

Where  $\star \in (*, -, >, \geq, \leq, <, \text{and}, \text{or}, \text{is}, \text{is not})$

$(VAR_{BS})$  is the derivation of an applied variable, which looks up the value of the variable in the store, which is connected to the variable environment through the location. The equation under the line can be informally read as, "In the composite function  $\sigma \circ Env_v$ , the variable  $x$  evaluates to the expression  $v$ .  $(\star_{BS})$  describes the derivation of the transition that most of the arithmetic and boolean expressions evaluate. The addition operator (+) is not included in  $(\star_{BS})$  because it derives differently based on whether the expressions are of type **Txt** or type **Num** / **Dec**. The derivation for addition is showcased in the appendix D

### Declarations

The next formation rules are the declarations. This includes both declaration of a function and the declaration of a statement. It is important to note that declarations can also be considered statements, however because of their fundamental significance, they are displayed separately. The format of a function declaration is the following  $Env_v, Env_f \vdash_l \langle D_F, \sigma \rangle \rightarrow \sigma'$ . How the function environment is updated when a new function is declared will be shown when defining the transition for function declaration.

The set of configurations for a function declaration is the following:

$$\Gamma_{\mathbf{DF}} = \mathbf{DecFunc} \times \mathbf{Sto}$$

The set of terminal configurations for a function declaration which the above set can transition to, is the following:

$$T_{\mathbf{DF}} = \mathbf{Sto}$$

Given the above configurations, the transition relation can be defined through a derivation. The derivation can be seen in definition 3.20.

$$(\mathbf{FUNCDEC}_{BS}) \frac{Env_v, Env_f[f \mapsto (S, x_1, \dots, x_k, Env_v)] \vdash_l \langle S_2, \sigma \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle \text{function } f(T_1x_1, \dots, T_kx_k) \text{ return } T\{S_1; \text{return } e\}; S_2, \sigma \rangle \rightarrow \sigma'} \quad (3.20)$$

Where  $S = S_1; \text{return } e$

The function name  $f$  is appended to the function environment where it maps to the statement ( $S$ ) within the block of the function, along with all the formal parameters  $(x_1, \dots, x_k)$ , and finally the variable environment ( $Env_v$ ) given at the time of declaration. Some of the formation rules have an additional statement at the end of its process, which allows us to show semantically how a given environment is updated when used in the continuation of the program.

The format for variable declarations is  $Env_v, Env_f \vdash_l \langle D_V, \sigma \rangle \rightarrow \sigma'$ . The configurations are the same as the function declarations configurations, but where we use the **DecVar** category instead of the **DecFunc**. The transition relation is shown in definitions 3.21 and 3.22.

$$(\mathbf{VARDEC1}_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v \quad Env_v[x \mapsto l], Env_f \vdash_{nxt(l)} \langle S, \sigma[l \mapsto v] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle T x = e ; S, \sigma \rangle \rightarrow \sigma'} \quad (3.21)$$

$$(\mathbf{VARDEC2}_{BS}) \frac{Env_v[x \mapsto l], Env_f \vdash_{nxt(l)} \langle S, \sigma[l \mapsto d(T)] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle T x ; S, \sigma \rangle \rightarrow \sigma'} \quad (3.22)$$

Where	
$d(\text{number})$	$= 0$
$d(\text{text})$	$= ""$
$d(\text{boolean})$	$= \text{F}$
$d(\text{decimal})$	$= 0.0$
$d(\text{list}(B))$	$= []$

**Table 3.5:** Default values for all variable types

The two variable declarations are fairly similar, ( $\text{VARDEC1}_{BS}$ ) allows the programmer to assign a new variable a value immediately, whereas ( $\text{VARDEC2}_{BS}$ ) is simply a declaration that a variable is now applicable. In both transitions, we update the variable environment with the new variable and the store with the new value. A declaration of an empty variable must be given a default value, which is seen in table 3.5. This is to avoid any errors that might occur when dealing with a variable that does not have a value within its domain. To better show how the updated environment is used, a statement ( $S$ ) is attached to both formation rules as previously done in the ( $\text{FUNCDEC}_{BS}$ ) in definition 3.20.

### Statements

The transition rules for statements are in the format  $\text{Env}_v, \text{Env}_f \vdash \langle S, \sigma \rangle \rightarrow \sigma'$ . Statements are essentially just ways to update the store, this is denoted by the sigma prime notation ( $\sigma'$ ) on the right-hand side of the arrow. Some of the most essential statement derivations are shown here in the report, the rest can be found in appendix D.

Similarly to all the other syntactical categories, the set of configurations needs to be defined for statements. Statements should be transitioned from a pair, given a statement and a store, into just an instance of a store. This is showcased in the following definitions.

$$\Gamma_{\text{stm}} = \text{Stm} \times \text{Sto}$$

This will transition into the terminal configurations given as the following definition.

$$T_{\text{stm}} = \text{Sto}$$

The first derivation shown is the compositional statement. This derivation is essential for allowing the program to be infinitely long, as it implements a recursive element to the statement category. The derivation is shown in definition 3.23.

$$(\text{COMP}_{BS}) \frac{\text{Env}_v, \text{Env}_f \vdash_l \langle S_1, \sigma \rangle \rightarrow \sigma' \quad \text{Env}_v, \text{Env}_f \vdash_l \langle S_2, \sigma' \rangle \rightarrow \sigma''}{\text{Env}_v, \text{Env}_f \vdash_l \langle S_1; S_2, \sigma \rangle \rightarrow \sigma''} \quad (3.23)$$

In  $(COMP_{BS})$  the two statements are separated above the line, so they can each be individually derived. The key observation to make with this derivation is how the store is updated and used. The second statement will use the updated store from the first statement. The final store for the complete derivation is below the line  $\sigma''$ , as it is the updated store after  $S_2$  has been derived.

The next statement described is the assignment statement. The assignment is used whenever updating an existing variable. The point of interest with the  $(ASS_{BS})$  derivation is how the store is updated. The derivation can be seen in definition 3.24.

$$(ASS_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v}{Env_v, Env_f \vdash_l \langle x = e, \sigma \rangle \rightarrow \sigma[Env_v(x) \mapsto v]} \quad (3.24)$$

The expression  $e$  is evaluated given the variable environment and store. In the transition below the line, we note that only the store has updated the value of the location of  $x$ . The variable environment remains the same, this is because the variable  $x$  is assumed to already exist in the environment.

A function call can also be used as an assignment but is one of the more complex derivations of the statement category. It is important to note that a function call originally also could be considered an expression because the function call can return an expression. However, to simplify the operational semantics a function call is only included as a statement. This simplifies the operational semantics since we no longer need to include the **sto** in our *terminal configurations* for the expression category. This change can be made as the result will be equivalent, shown in listing 3.9. The listing shows two syntactically correct function calls, however, the second example cannot occur in the abstract syntax.

```

1 | comment: Function call as statement;
2 | number x;
3 | number y = 0;
4 | function f(number i) return number {
5 |     return i+1;
6 | }
7 | x = call f(y);
8 | if (x > 0) {}
9 |
10 | comment: Function call as expression;
11 | number x;
12 | number y = 0;
13 | function f(number i) return number {
14 |     return i+1;
15 | }
```

```
16 | if (call f(y) > 0) {}
```

**Listing 3.9:** Example of equivalence of function call as statement and expression

$$\begin{array}{c}
 Env_f(f) = (S, x_1, \dots, x_k, Env'_v) \quad \sigma \circ Env_v \vdash e_i \rightarrow_{\mathbf{Exp}} v_i \\
 (FUNCEXP_{BS}) \frac{Env'_v[x_1 \mapsto l_1][x_2 \mapsto l_2] \dots [x_k \mapsto l_k], Env_f \vdash_{nxt(l)} \langle S, \sigma[l_1 \mapsto v_1] \dots [l_k \mapsto v_k] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle x = call f(e_1, \dots, e_k), \sigma \rangle \rightarrow \sigma'}
 \end{array} \quad (3.25)$$

Where  $i = 1 \dots k$   
 if  $l_1 = l$ ,  $l_{i+1} = nxt(l_i)$

The derivation for  $(FUNCEXP_{BS})$  is shown in definition 3.25. Everything above the line should be interpreted as a single long equation. It transitions the call of a function, used as an assignment to a variable. Above the line from left to right, it starts by deriving the function call into its representative statement, denoted by the  $S$  within the angled brackets. The transition also assumes an updated variable environment  $Env'_v$  where all the actual parameters from the call have been updated. The rest of the derivation above the line consists of the function name being associated in the function environment  $Env_f(f)$  with its applied statements and formal parameters. Lastly, the expressions given as actual parameters are evaluated.

Next is the list operations. In definition 3.26 the list addition operation is displayed. The addition operation allows a programmer to append any amount of expressions at the end of the list.

$$(LISTADD_{BS}) \frac{\sigma \circ Env_v \vdash e_i \rightarrow_{\mathbf{Exp}} v_i}{Env_v, Env_f \vdash_l \langle x : add(e_1, \dots, e_k), \sigma \rangle \rightarrow \sigma[Env_v(x) \mapsto L']} \quad (3.26)$$

Where  $i = 1 \dots k$  and  $Env_v[x \mapsto l]$   
 and  $L' = L$  where the values  $v_i$  is appended to the end of the list

In  $(LISTADD_{BS})$  we derive the statement by evaluating every expression into its value, and then the store is updated by updating the values of  $x$ , under the assumption that  $x$  is a list. Similar to the  $(ASS_{BS})$  derivation, the variable environment is not updated, since it is assumed that the list  $x$  already exists in that environment.

The last two statements whose derivation will be shown are the *if-true* statement which can be seen in definition 3.27 and a loop variant displayed in definitions 3.28 and 3.29.

$$(IFTRUE_{BS}) \frac{Env_v, Env_f \vdash_l \langle S, \sigma \rangle \rightarrow \sigma' \quad \sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} \mathbb{T}}{Env_v, Env_f \vdash_l \langle if(e) \{S\}, \sigma \rangle \rightarrow \sigma'} \quad (3.27)$$

The  $(IFTRUE_{BS})$  derives the statement  $S$  in the case that the given expression  $e$  derives to the Boolean value *true*. The expression evaluates to a Boolean in this case since we define the given expression in an *if* statement to be of Boolean type. Since a Boolean also can derive to *false*. It is also necessary to have a semantic derivation for the *if* statement in case the expression evaluates to false, displayed in appendix D.19.

$$(REPEATX-T_{BS}) \frac{Env_v, Env_f \vdash_l \langle S, \sigma \rangle \rightarrow \sigma' \quad \sigma \circ Env_v \vdash n \rightarrow_{\mathbf{Num}} v \quad Env_v, Env_f \vdash_l \langle \text{repeat } (v - 1) \text{ times } \{ S \}, \sigma' \rangle \rightarrow \sigma''}{Env_v, Env_f \vdash_l \langle \text{repeat } (n) \text{ times } \{ S \}, \sigma \rangle \rightarrow \sigma''} \quad (3.28)$$

Where  $n > 0$

$$(REPEATX - F_{BS}) \frac{\sigma \circ Env_v \vdash n \rightarrow_{\mathbf{Num}} 0}{Env_v, Env_f \vdash_l \langle \text{repeat } (n) \text{ times } \{ S \}, \sigma \rangle \rightarrow \sigma'} \quad (3.29)$$

Where  $n = 0$

The purpose of the loop variant derived in the above definitions, is to repeat a statement, a set amount of times. This loop variant can be useful, when wanting to repeat something, regardless of how the environment changes after each iteration. It is necessary to have two derivations of this one loop variant because two different outcomes are possible which change the flow of the program. The first transition  $(REPEATX - T_{BS})$ , derives the loop given  $n > 0$ , indicating that the loop should execute the statement. In order to repeat the loop we derive the exact same statement above line, but where the number now is subtracted by 1. Notice that the updated store  $\sigma'$  derived from the statement, is then used as the input store of the next loop iteration. When the loop counter hits 0, the  $(REPEATX - F_{BS})$  derivation should be activated, which ends the loop, when the number  $n$  is evaluated to 0.

### 3.5.5 Abstract Syntax Tree Design

An Abstract Syntax Tree (AST) is an abstract representation of a parse tree. Parse trees are derived from the CFG, and an abstract syntax tree is instead derived from the abstract syntax described in table 3.4. An AST contains nodes but without the syntactic details and unnecessary in-between steps, and therefore more abstract. The nodes are mostly decided by looking at the CFG and imagining the shortest possible way to represent the same tree. In figure 3.5 all the AST nodes designed for PEAK+ can be seen. The implementation of the AST in the compiler is described in the next chapter.





# Chapter 4

## Implementation

With the language design settled in chapter 3, this chapter documents the implementation of the PEAK+ compiler. It touches on how the compiler is structured, justification of the choices made in the process, and documentation with code snippets of the compiler implementation.

### 4.1 Compiler Phases

The main phases of a compiler are 'Syntax Analysis', 'Contextual Analysis', and 'Code Generation'. In some of these phases, the visitor pattern is mentioned, which is further described in section 4.3.4. In this section, these phases are explained and described below.

#### 4.1.1 Syntax Analysis

**Lexer:** The first part of the syntax analysis is the lexer, also known as the scanner. The lexer processes the input which is a stream of characters in the source language, and divides them into tokens. If a combination of characters does not match any token in the Lexer, an error is reported [37].

**Parser:** During Syntax Analysis the tokens get processed into a Parse Tree, using the CFG designed in section 3.4.3. The tokens are checked to see if they match the defined grammar rules. If no grammar rules match a sequence of tokens, an error is reported [37].

**AST:** Using the Parse Tree, an AST is generated to reduce the size of the tree by simplifying details. The AST nodes are designed in section 3.5.1. The compiler goes through each grammar rule in the Parse-Tree using the visitor pattern and converts the Parse-Tree into AST nodes [37].

### 4.1.2 Contextual Analysis

**Symbol Table:** To create a Symbol Table, the AST is iterated through a visitor pattern. In the Symbol Table, each scope containing its declared variables is stored. The Symbol Table also checks for scope issues and improper usage of declared variables and reports any error if so [37].

**Type Checker:** The Type Checker uses the AST and the Symbol Table to check for any type-related errors. This can be done using the visitor pattern [37].

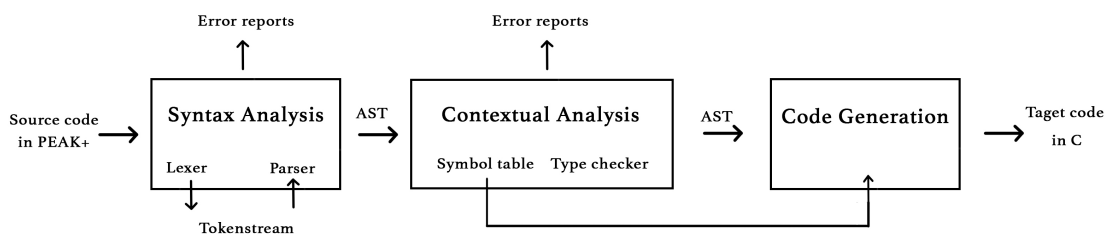
**Decorated AST:** When type-checking each node in the AST, additional information like type information and symbol table entries, can be added, generating a decorated AST. However, it is not necessary to create a decorated tree. It depends on the requirement for the compiler [37].

### 4.1.3 Code Generation:

Finally the compiler goes through Code Generation. Code Generation once again uses the visitor pattern. In this phase, instructions based on the decorated AST nodes are generated. These instructions get translated to the target language of the compiler [37].

## 4.2 PEAK+ compiler

To create the PEAK+ compiler, these three compiler phases are implemented. The rest of this chapter will describe the implementation of syntax analysis, contextual analysis, and code generation.



**Figure 4.1:** Overview of the PEAK+ compiler

As displayed in figure 4.1 which shows an overview of the PEAK+ compiler, the input to the lexer is the source code written in PEAK+ . The lexer generates the token

stream that the parser uses to generate the AST. Then the scope rules are checked in the symbol table, followed by type checking. The PEAK+ compiler will not generate a decorated AST in the contextual analysis phase. Since PEAK+ is a simple language, there is no need for decorating the AST before translating it into the target code. The target code for PEAK+ is the C programming language. One of the reasons C is chosen is because of its flexibility, which makes it easier to adjust to the semantics of PEAK+ , more details on the reasons behind C as the target code is explained in section 4.5.

### 4.2.1 Deciding Compiler Language

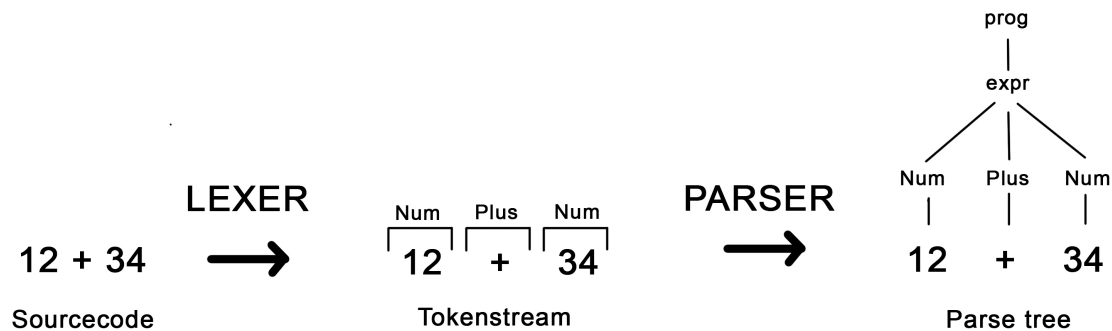
In order to best implement a visitor pattern later on in development, an object-oriented language is needed. The visitor pattern is explained further in section 4.3.4. For this reason, the PEAK+ compiler is going to be developed in C#. This is also because C# is a language that the group is acquainted with as well as how the project organization works in C#.

## 4.3 Syntax Analysis

Syntax Analysis is the first phase of the compiler which ends up generating an Abstract Syntax Tree (AST). This section will cover the theory of the parser algorithm followed by the implementation in the PEAK+ compiler including the Parser Generator, and how the AST has been built.

### 4.3.1 Parsing Theory

When a compiler translates source code, the parser defines the structure of the code by producing a parse tree. The basis for creating the parse tree is the token stream created by the lexer. The Lexer converts the source code characters into specified tokens. Lexical grammar uses regular expressions to define the grammar rules of a token. As displayed in figure 4.2, the lexer, for example, translates the source code "12 + 23" into the *Num*, *Plus*, and *Num* tokens. The parser uses this token stream to produce the parse tree by using the correlated CFG production rules. Implicit in the displayed example, a lexer rule will exist, specifying that a sequence of digits corresponds to the *Num* token, the plus symbol corresponds to the *Plus* token, and that the correlated CFG production rules say that a program consists of an *expr* and that a *expr* can be a *Num Plus Num* [38].



**Figure 4.2:** An example of the process of translating source code into a parse tree

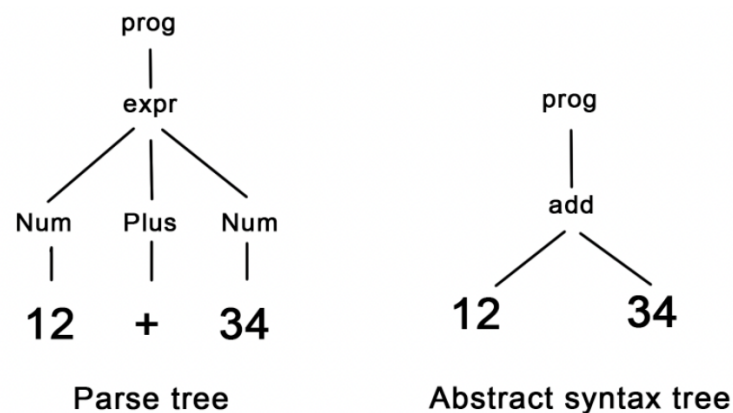
By generating the parse tree, the parser checks the syntactic correctness of the source code. If the source code consists of syntax errors, error reports will be thrown. An example of syntactically correct but semantically incorrect code is displayed in listing 4.1. The parser will not produce any errors, since the structure of the code is correct. But if the code is executed, it will fail since the variable `x` is not declared [38].

```

0 | int y = 10;
1 | int sum = x + y;
  
```

**Listing 4.1:** example of code with correct syntax but wrong semantics.

A parse tree is also called a concrete syntax tree (CST) and is traversed, from the root node of the tree, down to its subtrees representing smaller pieces of the code. The same principle applies to the AST, except that the CST is represented using the concrete syntax while the AST is more simplified and has fewer details that are not relevant for generating code [38]. The difference between the two trees can be seen in figure 4.3.



**Figure 4.3:** Examples showing the difference between a CST and an AST

There are two approaches to parsing: top-down and bottom-up. A top-down parser traverses the tree from the root node until each leaf is visited. These are also known as LL(K) parsers (Left-to-right read of input, Leftmost derivation). The k indicates the amount of lookahead the algorithm has, meaning how many tokens the parser has to include in its decisions of what grammar rule to follow. LL parsers are easy to implement but less efficient because they cannot handle left recursion grammar rules. However, there will always be an equivalent non-recursive grammar to a left recursive grammar, but the rewritten grammar rules can be more complex to understand, or need more lookaheads than the recursive grammar [38].

A bottom-up parser starts from the lowest part of the tree, ending at the root. The LR (Left-to-right read of the input, Rightmost derivation) parser is a Bottom-up parser and is harder to implement, but can handle most grammars like left recursion. Both LL and LR parsers use parse tables that store information about the next action to take. For LR parsers, these tables can become large and complex. LALR (Look-Ahead, Left-to-right) is a variant of the LR parser and attempts to address the LR parsers issue by having large table sizes be more compact at the cost of potential conflicts with look-ahead [38].

### 4.3.2 Deciding Parser Tool

Generating the parser can be done by hand or by using a parser generator tool. Using a tool can be faster and more accurate, especially with larger languages. A tool helps to reduce errors in the parser but often has some requirements for the grammar, like having to be LL(1). However, by generating the parser by hand, the developer has more control over how to handle the special requirements. Though, if the language's grammar is large and complex, it is a difficult and time-consuming task to write the parser by hand, leading to errors in the parser. Even though PEAK+ is not that complex, our learning curve in this project means that changes probably would occur along the way. Because of the time consumption of having to rewrite the parser by hand several times, combined with our missing experience in writing a parser, we decided that a tool would be the appropriate choice. There are many different parser generators, and we researched JavaCup, SableCC, and ANTLR.

JavaCUP is an LALR bottom-up parser and generates a parser written in Java. Both SableCC and ANTLR are LL(k) and can generate a parser in multiple different languages. We prefer writing code in C# and therefore the choice is between SableCC and ANTLR. After experimenting with getting both tools to work, ANTLR is the first to successfully work and to show a visual parse tree. In view of getting the parsing process started, ANTLR is therefore chosen as the parser generator. ANTLR can handle left recursion, and is an LL(\*) parser meaning it can handle any amount of lookahead tokens [38].

### 4.3.3 Lexer

ANTLR provides the ability to perform lexical analysis if it receives a lexer. Because of this, we are able to take our CFG and transform all terminal rules into keywords stored in the lexer. Some of the same terminal rules appear multiple times in the CFG, and therefore it is now easier to change all appearances of a terminal rule at once. Examples of keywords can be seen in listing 4.2 for various list operations. In the lexer, definitions of regular expressions can be made. ANTLR uses different characters in its regular expressions compared to what normally is used. ANTLR for example uses `~` instead of `^` for excluding characters. Regular expressions are e.g. used in PEAK+ for string values. In PEAK+, a string value starts with `"` followed by any character which is not a `"` and ends with `"`. In ANTLR this regular expression can be seen in listing 4.2 as `'STR'` as well as some examples of other lexer rules. The full lexer can be seen in appendix B.2.

```

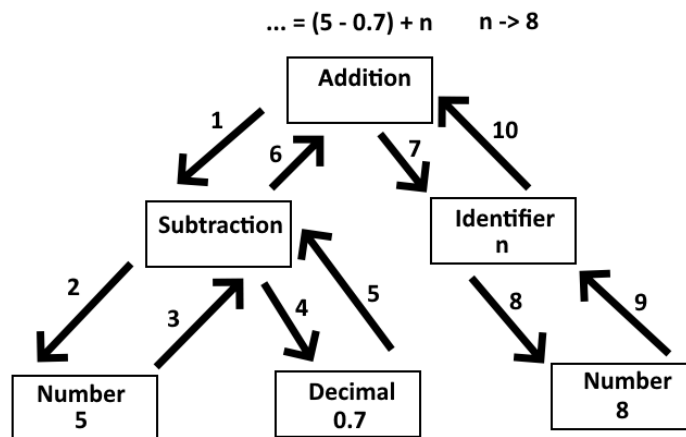
1 LISTADD: 'Add';
2 LISTIDXOF: 'IndexOf';
3 LISTREPLACE: 'Replace';
4 LISTVALOF: 'ValueOf';
5
6 COMM: 'comment: '~(';'')*';
7 STR: '"' (~'"')* '"';
8 DEC: ('+' | '-' )? [0-9]+'.' [0-9]+;
9 INT: ('+' | '-' )? [0-9]+ ;
10 ID: [a-zA-Z_][a-zA-Z_0-9]* ;
11 WS: [ \t\n\r\f]+ -> skip ;

```

Listing 4.2: Part of PEAK+ 's Lexer - CobraCompiler/ExprLexer.txt

### 4.3.4 The Visitor Pattern

In order to traverse a tree like the CST, a visitor pattern is required. A visitor pattern utilizes a depth-first search method to traverse the tree. It solves the problem of having to modify each class in a larger class structure whenever a change needs to be implemented. Instead, all visitor methods are contained within one class, and therefore only the functions have to be modified. This pattern is used in multiple phases of the PEAK+ compiler, like when building the AST, creating the symbol table, type-checking, and generating the target code. Based on the context, a visit function in the pattern can return something different. For example, when building the AST, each visit function can return an AST node in order to build upon the AST [46]. An example of how the visitor pattern visits nodes using the depth-first search algorithm can be seen in figure 4.4.



**Figure 4.4:** Example of the visitor pattern. The numbers above the arrows show which order the nodes are visited

### 4.3.5 Building the Abstract Syntax Tree

Using ANTLR generates multiple classes including *ExprParser*, *ExprLexer*, and *ExprParserBaseVisitor*. The order in which ANTLR generates the parser can be seen in listing 4.3. First ANTLR uses an *AntlrInputStream* class in order to turn the input into a stream of characters that ANTLR can use, seen on line 28. This stream of characters is used to instantiate a new *ExprLexer* class which is assigned to the lexer variable on line 29. The lexer is then given to a *CommonTokenStream* class which converts the lexer into tokens for the parser. This can be seen on line 30. The returned token stream is given to *ExprParser* which finally generates the parser on line 31. From the parser, the start-node can be retrieved, which in our CFG is 'program'. This root is assigned to a variable called 'cst' (concrete syntax tree) on line 35. The 'cst' is required in order to build the AST.

```

28 | var inputStream = new AntlrInputStream(new StringReader(
    |     exprText));
29 | var lexer = new ExprLexer(inputStream);
30 | var tokenStream = new CommonTokenStream(lexer);
31 | var parser = new ExprParser(tokenStream);
32 | .
33 | .
34 | .
35 | var cst = parser.program();

```

**Listing 4.3:** SymbolTable Class - CobraCompiler/Program.cs

Before building the AST, classes for each of the possible nodes must be created. 'ASTNodes' is a class that is created to contain these as well as the datatypes in PEAK+ stored as an enumeration. These nodes are designed in section 3.5.5. Each node inherits from an abstract class 'ASTNode' and through this, all nodes contain a property

'Line' which represents the line in the code where the node originates from. This property is used to inform the user where an error stems from if one occurs.

An example of the 'AssignNode' can be seen in listing 4.4. In PEAK+ , an assignment is written as *identifier = expression*. 'AssignNode' inherits 'CommandNode' since an assignment is a command. It contains an *IdentifierNode* on line 54 and an *ExpressionNode* on line 55.

```

52 | internal class AssignNode : CommandNode
53 | {
54 |     public IdentifierNode Identifier { get; set; }
55 |     public ExpressionNode Expression { get; set; }
56 | }

```

**Listing 4.4:** SymbolTable Class - CobraCompiler/ASTNodes.cs

ANTLR generates the 'ExprParserBaseVisitor' class which is used by a class 'BuildASTVisitor'. It is used for building the AST by having 'BuildASTVisitor' inherit from it. 'ExprParserBaseVisitor' provides a visitor pattern for going through the symbols in the parse tree. It is a generic class and the generic type given is the return type of all visitor implementations of the class. This is in our case the class 'ASTNode'. The class contains unique visit methods for each non-terminal in PEAK+ 's CFG, as well as a general 'Visit()' method. In each of these methods, you are able to access the non-terminal 'context' which is used to get its children. The context can also be used to retrieve the line in which a rule occurred that can be stored in an AST node. The children can be used on the 'Visit()' method to visit the non-terminals children.

Using this, we can visit all children in the parse tree and return the correct ASTNodes. Each unique visit method has the following structure:

- All children are initialized at the top using the symbol's context
- If a node is to be returned, this also gets initialized here as 'outputNode'
- One or more if-statements check for necessary syntax rules and visit the children
- At the bottom, the *outputNode*'s properties are assigned, and it is returned

In order to test if 'BuildASTVisitor' returns the correct output, a 'prettyPrint()' function is run every time a new node is added to the AST. In order to correctly print the nodes at their respective levels of the AST, an indentation function 'incrIndent()' can be called. This function increments indentation for the next time a node is pretty printed. A function 'decrIndent()' decreases indentation again. An example of how it looks in the terminal, as a result of the implemented indentation can be seen in figure 4.5. Because of time pressure, the pretty printer is not being maintained in the second iteration as it is mostly used to check if the output of the *BuildASTVisitor* matches the



source code. In the second iteration, we use debugging features to ensure that the resulting AST match.

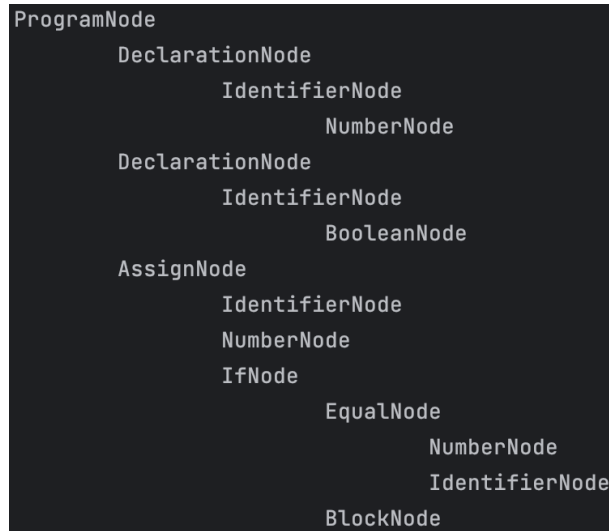


Figure 4.5: Pretty print example.

An example of one of the visit functions can be seen in listing 4.5. This example is of a declaration rule in PEAK+ 's CFG which has the following structure: *dcl* → *type ID ass SEMI*;. In the function, all children are initialized at the top on lines 139-142 as well as the node to return on line 144, which is a declaration node. On lines 149-152 an if-statement checks that the syntax is correct. Since a declaration-node contains an identifier-node we initialize this on line 154. An identifier node has a type node that is visited on line 159 and returned. This type node is assigned to the identifier node on line 160. On line 162, the declaration is checked for any assignments, since it is possible in PEAK+ to have a declaration without any assignments. This assignment is assigned as an 'expression' to the 'outputNode' on line 167. The identifier node is also assigned on line 145 and the output is returned on line 170.

```

132 public override ASTNode VisitDcl([NotNull] ExprParser.
    DclContext context)
133 {
134     //outputNode = DeclarationNode
135     //visit(type) -> gets TypeNode
136     //visit(ass) -> gets ExpressionNode
137     //Type and Expression is assigned to an identifierNode
138     //The identifierNode is assigned to the OutputNode
139     var type = context.type();
140     var ID = context.ID();
141     var ass = context.ass();
142     var SEMI = context.SEMI();

```

```
143
144     var outputNode = new DeclarationNode() { Line = ass.start.
145         Line };
146     prettyPrint("DeclarationNode", context);
147     incrIndent();
148
149     if ((type != null) &&
150         (ID != null) &&
151         (ass != null) &&
152         (SEMI != null))
153     {
154         var identifier = new IdentifierNode() { Line = ID.
155             Symbol.Line };
156         prettyPrint("IdentifierNode", context);
157         incrIndent();
158
159         identifier.Name = ID.ToString();
160         var typeNode = (TypeNode)Visit(type);
161         identifier.TypeNode = typeNode;
162
163         if (ass.ChildCount > 0)
164         {
165             outputNode.Expression = (ExpressionNode)Visit(ass);
166         }
167
168         outputNode.Identifier = identifier;
169     }
170     decrIndent();
171     return outputNode;
172 }
```

**Listing 4.5:** BuildASTVisitor Class - CobraCompiler/BuildASTVisitor.cs

## 4.4 Contextual Analysis

Contextual Analysis is the phase of the compiler that usually "decorates" the AST to make a Decorated AST. It uses the Symbol Table and Type Checking to add any additional information which can for example be type information when performing type conversion. After the decorated AST is made the compiler enters the Code Generation phase. In PEAK+ , we did not add type information to the AST until the second iteration, as it is necessary for the code generation to recognize for example if an addition-node is concatenating strings or adding two numbers.

### 4.4.1 Symbol Table

The symbol table created for PEAK+ is developed in C#. It is created using a 'Scope' class which represents the scopes of the program. This class can be seen in listing 4.6 and contains a stack of dictionaries, used for storing symbols and a reference to the scope 'Parent' (meaning its outer scope). It also stores a reference to the Block Node tied to the scope. In order to get a symbol from the dictionary when knowing its ID, the dictionary's key is the symbol's ID. This helps to get the symbol in constant time, which is more efficient for the compiler than iterating through all symbols until it finds a match. The Symbol class in listing 4.7 defines the properties of a symbol, which includes its name, type, and reference to the declaration node. The Type property is of type *TypeEnum*, which is an enumerator that defines the possible types of symbols: number, text, boolean, decimal, list\_number, list\_text, and list\_boolean.

The scope rules which the symbol table manages are described in section 3.4.5.

```
28 public class Scope
29 {
30     public Scope()
31     {
32         Symbols = new Dictionary<string, Symbol>();
33     }
34     public Dictionary<string, Symbol> Symbols { get; set; }
35     public Scope Parent { get; set; }
36     public BlockNode Block { get; set; }
37 }
```

Listing 4.6: Scope Class - CobraCompiler/SymbolTable.cs

```
21 public class Symbol
22 {
23     public TypeEnum Type { get; set; }
24     public string Name { get; set; }
25     public ASTNode Reference { get; set; }
```

26 | }

**Listing 4.7:** Symbol Class - CobraCompiler/SymbolTable.cs

The SymbolTable class in listing 4.8 builds and stores the symbol table. Besides its 'Visit()' functions, the class provides six methods *BuildSymbolTable()*, *ProcessNode()*, *NewScope()*, *ExitScope()*, *Insert()*, and *Lookup()*. The *BuildSymbolTable()* method takes the root of the AST as its input and processes each node in the tree using the visitor pattern.

The *NewScope()* method is used when visiting a new block when building the symbol table. It takes a *BlockNode* as a parameter, creates a new scope, and pushes it onto a stack. This stack keeps track of the current scope by storing it at the top. If possible, the parent of the scope is also set. The scope is also stored in a separate dictionary where each scope can be accessed using its *BlockNode*. Finally, the blockNode which belongs to the now current scope, is stored.

The *ExitScope()* method pops the top element of the scope stack to represent exiting a scope. This method is called when exiting a scope during the symbol table creation.

The *Insert* method takes a string *name* and a *TypeEnum* as its inputs and adds a new symbol to the current scope at the top of the stack. *Insert* is called when visiting a declaration node, as the newly declared variable needs to be stored in the symbol table.

The *Lookup* method takes a string *name* and a *BlockNode*. It starts at the scope belonging to the *BlockNode* and searches outwards through each scope's dictionary for the requested symbol and returns it if found. If the symbol is not found, *null* is returned and an error is reported. This function is called when meeting an Identifier node in the AST, which represents already declared variables.

```

39 public class SymbolTable : ASTVisitor<ASTNode?>
40 {
41     public Dictionary<BlockNode, Scope> _scopes;
42     public Stack<Scope> _stackScopes;
43     private ErrorHandler symbolErrorHandler;
44     public BlockNode _currentBlock;
45     .
46     .
47     .
48     public SymbolTable(ErrorHandler errorHandler)
49     {
50         symbolErrorHandler = errorHandler;

```

```

51     _scopes = new Dictionary<BlockNode, Scope>();
52     _stackScopes = new Stack<Scope>();
53 }
54 public SymbolTable BuildSymbolTable(ASTNode astRoot)
55 {
56     Visit((ProgramNode)astRoot);
57     return this;
58 }

```

Listing 4.8: SymbolTable Class - CobraCompiler/SymbolTable.cs

```

232 public override ASTNode? Visit(DeclarationNode node)
233 {
234     if (_reservedKeywords.Contains(node.Identifier.Name))
235         node.Identifier.Name = $"{node.Identifier.Name}_";
236
237     var sym = Lookup(node.Identifier.Name, _currentBlock);
238
239     string underscores = "";
240     while (sym != null)
241     {
242         underscores += "_";
243         sym = Lookup($"{node.Identifier.Name}{underscores}",
244                     _currentBlock);
245     }
246
247     node.Identifier.Name += underscores;
248     Insert(node.Identifier.Name, node.Identifier.TypeNode.Type,
249           node);
250     Visit(node.Expression);
251     return null;
252 }

```

Listing 4.9: Visit(DeclarationNode node) - CobraCompiler/SymbolTable.cs

*NewScope()* in listing 4.10 creates a new scope in the symbol table. It takes a block node on line 75, which is associated with the scope, and assigns it to a new scope on line 78. If any scopes already exist, the scopes parent is assigned to the current scope at the top of the stack on lines 80-81. The scope is pushed onto a stack of scopes and added to a dictionary of scopes on lines 83-84. The current block is then updated to refer to the new scope block on line 85. The method uses *\_stackScopes.Peek()* on the dictionary on lines 81 and 85 to retrieve the top element which is the current scope. This function is called when entering a new block of code, such as a function or loop. Variables declared within this block will only be visible within this scope and any nested scopes.

```
75 private void NewScope(BlockNode blockNode)
76 {
77     var scope = new Scope();
78     scope.Block = blockNode;
79
80     if (_stackScopes.Count > 0)
81         scope.Parent = _stackScopes.Peek();
82
83     _scopes.Add(blockNode, scope);
84     _stackScopes.Push(scope);
85     _currentBlock = _stackScopes.Peek().Block;
86 }
```

**Listing 4.10:** NewScope() - CobraCompiler/SymbolTable.cs

The method *ExitScope()* in listing 4.11 exits the current scope in the symbol table. It pops the top dictionary from the scope stack using the *Pop()* method, effectively removing it from the symbol table on line 90. It also updates the current block to be that of the new current scope after popping the stack on lines 92-93. It does this if the block is not of type program-node as you can not go further out than the program-node. This method is called when exiting a block of code.

```
89 private void ExitScope()
90 {
91     _stackScopes.Pop();
92     if (_currentBlock is not ProgramNode)
93         _currentBlock = _stackScopes.Peek().Block;
94 }
```

**Listing 4.11:** ExitScope() - CobraCompiler/SymbolTable.cs

The method *Insert()* in listing 4.12 inserts a new symbol into the symbol table. It takes three arguments: the name of the symbol, its type represented by a *TypeEnum*, and the node of the declaration on line 98. Because of the scope rules defined in chapter 3, the same name for a variable can not be defined within the same scope, only in separate scopes. Therefore *Insert()* checks if the current scope contains the name of the variable being inserted on line 100. If it already exists, an error will be reported on line 102. If this is not the case, it adds a new entry to the dictionary using the name parameter as the key and a new *Symbol* object as the value with all the arguments given on line 106.

```
98 public void Insert(string name, TypeEnum type, ASTNode node)
99 {
100     if (_stackScopes.Peek().Symbols.ContainsKey(name))
101     {
102         SymbolError(node, $"The variable '{name}' is defined
            twice within the same scope.");
    }
```

```
103         return;
104     }
105
106     _stackScopes.Peek().Symbols.Add(name, new Symbol
107     {
108         Name = name, Type = type, Reference = node
109     });
110 }
```

**Listing 4.12:** Insert() - CobraCompiler/SymbolTable.cs

*Lookup()* in listing 4.13 searches for- and returns a symbol in the symbol table. It takes two arguments being the name of the symbol to look up and a block on line 114. The method first gets the scope associated with the block given on line 116. It enters a while loop checking that the scope has a value on line 118. It then uses a for each to loop through the scopes symbols on line 120. It checks if the current symbol's name matches the name given on line 122. If this is the case, it returns the current symbol on line 124. If the symbol is not found in the scope symbols the scope gets updated to be its own parent on line 128. If the symbol is not found, the method returns *null* on line 130.

This method is called when needing to find a symbol associated with a variable name in order to for example look up the type of the variable that has been previously declared. The function can also be used to detect errors, such as when attempting to use an undeclared variable, as giving the name of an undeclared variable, would result in the function returning null.

```
114 public Symbol? Lookup(string name, BlockNode blockNode)
115 {
116     var scope = _scopes[blockNode];
117
118     while (scope != null)
119     {
120         foreach (var symbol in scope.Symbols.Values)
121         {
122             if (symbol.Name == name)
123             {
124                 return symbol;
125             }
126         }
127
128         scope = scope.Parent;
129     }
130     return null;
}
```

131 | }  

---

**Listing 4.13:** Lookup - CobraCompiler/SymbolTable.cs

This section highlights the importance of the *NewScope()*, *ExitScope()*, *Insert()*, and *Lookup()* methods for creating, updating, and searching for symbols in the symbol table, which is essential for performing tasks such as type checking and code generation. Overall, this implementation of a symbol table using a stack of dictionaries, as discussed in section 3.4.5 about scope rules, serves as a useful tool for managing the scope and the lifetime of variables and functions in a PEAK+ program.

**Fixing scope rules**

During the second iteration of PEAK+ , a problem regarding function declaration was realized. This problem was caused by the difference in starting points in PEAK+ and C as well as the inability to declare functions inside functions in C. This problem is described more in-depth in section 4.5.3. In PEAK+ , you are able to declare a variable with the same name twice in different scopes and also declare a function wherever as long as it is in the outer-most scope. When generating code for C, functions need to be declared outside the *main()* function but still have the scope rules of PEAK+ .

Therefore a clearer distinction between variables of the same name is needed. This distinction is made by placing '\_' behind the names of variables already declared and accessible from the scope. This change occurs when visiting a declaration node and the code for it can be seen in appendix E.1.10 in listing E.20.

This fix had to match what happens when visiting an identifier. When visiting an identifier, the name of the variable gets looked up until it finds one that is not already declared, adding a '\_' behind it each time. If it finds one that is not declared, it uses the symbol of the one just before the symbol not declared. This visit function for an identifier-node can be seen in appendix E.1.10 in listing E.21.

Because the functions in C are declared outside of *main()*, the variables not declared in the function need to be passed as arguments instead. Therefore another function *AddIdToFunctionBlock()* in the symbol table is added which is used when visiting the function declaration. This function can be see in appendix E.1.10 in listing E.19.

**4.4.2 Type Checker**

Both the Type-Checker and the Emitter class have to traverse the AST. Therefore the AST-Visitor class is created. The AST-Visitor creates *visit()* functions for each AST node that can appear in an AST.



In order to check for type errors at compile-time, a *Type-Checker* class is created, in which the nodes in the AST are visited. The *Type-Checker* class, therefore, implements the *AST-Visitor* class. In the *visit()* function for a node, the children of the node are visited. Any errors caused by invalid type operations are checked and the type of the node is returned.

An example of this is any of the infix-expressions such as *addition node* which can be seen in listing 4.14. In this node, the left and right sides are visited in order to get their types on lines 278-279. Then any errors regarding invalid typing are collected. If the left or right side type is of type *boolean*, an error is reported on lines 281-285. If the left or right side is of type *list*, an error is reported on lines 286-290. If only the left or only right side is of type *text*, an error is reported on lines 291-296. Finally, the type of the addition node is assigned. If both sides are *number*, the type becomes *number* on lines 298-299. If both sides are *text*, the type becomes *text* on lines 300-301. If one of the sides is *decimal*, the type becomes *decimal* on lines 302-303. The type of node is returned on line 307.

```

273 public override TypeEnum? Visit(AdditionNode node)
274 {
275     //Visits left and right side and gets their type
276     //Check if the types match, and that they are of valid
        typing for Addition
277
278     TypeEnum? leftType = Visit(node.Left);
279     TypeEnum? rightType = Visit(node.Right);
280
281     if (leftType == TypeEnum.boolean || rightType == TypeEnum.
        boolean)
282     {
283         TypeError(node, $"Addition of type 'boolean' is not
            allowed.");
284         return null;
285     }
286     else if (isList(leftType) || isList(rightType))
287     {
288         TypeError(node, $"Addition of type 'list' is not
            allowed.");
289         return null;
290     }
291     else if ((leftType == TypeEnum.text && rightType !=
        TypeEnum.text) ||
292             (leftType != TypeEnum.text && rightType ==
        TypeEnum.text))
293     {

```

```
294         TypeError(node, $"type 'text' can only be added with
295             another value of type 'text.'");
296     }
297
298     if (leftType == TypeEnum.number && rightType == TypeEnum.
299         number)
300         node.Type = TypeEnum.number;
301     else if (leftType == TypeEnum.text && rightType == TypeEnum
302         .text)
303         node.Type = TypeEnum.text;
304     else if (leftType == TypeEnum._decimal || rightType ==
305         TypeEnum._decimal)
306         node.Type = TypeEnum._decimal;
307     else
308         throw new Exception();
309
310     return node.Type;
311 }
```

**Listing 4.14:** Visit(AdditionNode node) - CobraCompiler/TypeChecker.cs

### 4.4.3 Error Handling

As mentioned before, the compiler is written in C#, which means that whenever an error occurs, a C# exception is called. A C# exception is arguably not very readable for beginners as these exceptions often use complex programmer terms. It is important that the compiler gives readable and understandable error messages to the user. In that way, we can accommodate beginner programmers who are not already familiar with complex error messages.

The focus is to create comprehensive error messages that all get displayed at once, with the exception of syntax errors as they create errors to the symbol table and type checking that is unnecessary to display to the user. Therefore the syntax errors get displayed, and the program stops. The reason behind adding the errors to a list and writing them to the terminal is that they should be grouped together. This means that all of the errors get added to the list one by one if there are multiple, and displayed together so that the beginner programmer can see all of the errors in one go, instead of having to fix one error, and then compile again to see the next error.

Error handling is the first phase the parser goes through. This makes sense because this is where the first errors that can occur get handled, since this is where all of the syntactic errors can happen, and the syntax has to be perfect before the AST and

Symbol Table can continue with the code.

A class called "ErrorHandler" contains the code to take care of errors in the PEAK+ compiler.

```

8 public class ErrorHandler : BaseErrorListener,
   IAntlrErrorListener<IToken>
9 {
10     private readonly List<string> _syntaxSyntaxErrorMessages =
        new();
11     public IReadOnlyList<string> SyntaxErrorMessages =>
        _syntaxSyntaxErrorMessages;
12
13     private readonly List<string> _symbolErrorMessages = new();
14     public List<string> SymbolErrorMessages =>
        _symbolErrorMessages;
15
16     private readonly List<string> _typeErrorMessages = new();
17     public List<string> TypeErrorMessages => _typeErrorMessages
        ;
18
19     public override void SyntaxError(IRecognizer recognizer,
        IToken offendingSymbol, int line, int charPositionInLine
        , string message, RecognitionException e)
20     {
21         var error =
22             $"Error line {line}: {message} Caused by {
                offendingSymbol.Text}.";
23         _syntaxSyntaxErrorMessages.Add(error);
24     }
25 }

```

**Listing 4.15:** ErrorHandler - CobraCompiler/ErrorHandler.cs

First, the class ErrorHandler declares 3 lists. each list contains its own type of error message. There is one for Syntax errors, Symbol errors, and one for Type errors. The SyntaxError function is generated from ANTLR. The method gets called when there is a syntax error, and it adds a new *error* to the syntax error list. By default, ANTLR implements an unattainable error listener in the BaseErrorListener. An error listener is used to look for and retrieve errors. In ANTLR, these errors get printed in the terminal if found and then the rest of the compiler is run. This creates an issue where the compiler continues even though an error occurred, creating issues in later phases. The fix is for the compiler to create an instance of the *ErrorHandler* class and replace the default one existing in the parser on lines 33-35 in listing 4.16. This way, after the parser has collected its errors, the error listener can now show the errors to the user and terminate the compiler.

```
33 var errorHandler = new ErrorHandler();
34 parser.RemoveErrorListeners(); // Remove the default
    ConsoleErrorListener
35 parser.AddErrorListener(errorHandler); // Set the ErrorHandler
    as the error listener
```

**Listing 4.16:** Editing the ErrorHandler on the parser - CobraCompiler/Program.cs

The code in listing 4.16 is contained in the program.cs class and is executed before the parse tree is generated. On line 38 in listing 4.17 the compiler parses the source code when calling *parser.program()*. Every time there is a syntax error, the error listener adds an error message to the *SyntaxErrorMessages* list. On line 39 the compiler checks whether the list with syntax errors contains any error messages. If it does, a for each loop iterates through all of the errors and prints them to the console on lines 42-45. Once the compiler has written all of the errors in the terminal, a method called *Environment.Exit(1)* shuts down the program. This is done because having errors in the syntax will also cause issues in future phases. To not confuse the beginner programmer, all syntax errors should be corrected before recompiling. This way of adding error listeners to each phase and collecting errors is repeated for every phase. This is because any errors in a phase will cause issues in all future phases.

```
38 var cst = parser.program();
39 if (errorHandler.SyntaxErrorMessages.Count > 0)
40 {
41     Console.WriteLine("Syntax errors:");
42     foreach (var errorMessage in errorHandler.
        SyntaxErrorMessages)
43     {
44         Console.WriteLine(errorMessage);
45     }
46     Environment.Exit(1);
47 }
```

**Listing 4.17:** Handling syntax errors - CobraCompiler/Program.cs

## 4.5 Code Generation

When type checking is complete, the next phase is code generation.

Before proceeding with code generation, a target language must be chosen. In order to decide which language to compile to, we have looked at languages that we already have experience with from previous semesters. An argument for choosing C# is that C# is a language the group knows very well and it will be easier for us to navigate. An argument for choosing C is that C is more flexible and it will therefore be easier to adjust the compiled code to PEAK+ 's semantics. In the first iteration, C# is chosen

as the target language in order to be able to finish it faster and get a working emitter. However, for the second iteration, C is chosen as the target language. Changing the code in the emitter from C# to C can be done quickly for most constructs, as the two languages are similar in syntax in many ways. However creating data structures such as lists etc. will take longer since a list is not a construct that exists in C. Here, a linked list will need to be implemented and methods defined for operations on the linked list. However, the advantages are greater in the form of more control and flexibility in the end.

The code generation phase is done using an emitter, which produces the target language code by visiting the AST. When visiting the AST, each visited node is converted to the equivalent code in C. This process uses the visitor pattern like in the Type-checker and Symbol-table phases. In the code generation phase, each node returns a `StringBuilder`, containing code in the target language. When compiling the generated target code you get an executable program that can be run in order to get results.

Below a couple of examples of code-generation of some different components in PEAK+ can be seen, as well as how these components translate to C code:

#### 4.5.1 Emitting lists and list methods

In order to emit lists and list methods, a new data structure has to be implemented as lists do not exist in C. In order to be able to instantiate lists and apply list operations whenever necessary, the list helper methods along with the list data structure are defined in the *ProgramNode* visitor. These methods are declared before the *main()* function in C. This way, when visiting nodes further down the AST, these functions can be called when necessary. This also helps make the generated C code more readable when checking if the generated output matches the input. The list construct in C has been designed as a singly linked list, meaning a struct containing a pointer to the value of its current element and a pointer to the next element (See lines 61-64 in listing 4.18. Note that with this implementation, a specific index cannot be accessed in constant time (e.g. the *ValueOf()* function seen on lines 68-85 in listing 4.18), but that the list has to be iterated through from the starting element in order to obtain the value of the specific index. However, since the language is not intended for creating large and optimized applications, the linked list data structure is deemed sufficient for *List* implementations of PEAK+ .

```
60 //Struct List:
61 stringBuilder.AppendLine("struct node\n{");
62 stringBuilder.AppendLine(" void *value;");
63 stringBuilder.AppendLine(" struct node *next;");
64 stringBuilder.AppendLine("};");
65 .
```

```

66 .
67 .
68 //List:ValueOf() function:
69 stringBuilder.AppendLine("void *ValueOfList(struct node *list,
    int index)");
70 stringBuilder.AppendLine("{");
71 stringBuilder.AppendLine(" struct node *curr_node = list;");
72 stringBuilder.AppendLine(" int i;");
73 stringBuilder.AppendLine(" for (i = 0; i < index; i++) {");
74 stringBuilder.AppendLine(" if (curr_node == NULL) {");
75 stringBuilder.AppendLine(" fprintf(stderr, \"Error: Invalid
    index\\n\");");
76 stringBuilder.AppendLine(" exit(EXIT_FAILURE);");
77 stringBuilder.AppendLine(" }");
78 stringBuilder.AppendLine(" curr_node = curr_node->next;");
79 stringBuilder.AppendLine(" }");
80 stringBuilder.AppendLine(" if (curr_node == NULL) {");
81 stringBuilder.AppendLine(" fprintf(stderr, \"Error: Invalid
    index\\n\");");
82 stringBuilder.AppendLine(" exit(EXIT_FAILURE);");
83 stringBuilder.AppendLine(" }");
84 stringBuilder.AppendLine(" return curr_node->value;");
85 stringBuilder.AppendLine("}");

```

**Listing 4.18:** Emitting the code for the list implementation and ValueOf function - CobraCompiler/Emitter.cs

There are multiple different list visitors, each emitting the code for a list method. The implementation of the *ListValueOfNode* visitor can be seen in listing 4.19. This method first determines the type of the list and appends a cast of the correct type to the *StringBuilder* in order to ensure that no errors occur when receiving the return value from the *ValueOfList()* function. This can be seen in lines 713-725. Next the calling *ValueOfList()* function shown in listing 4.18 on lines 68-85, is appended to the *StringBuilder* in line 727 in listing 4.19. Finally, the argument expression of the *ValueOf()* (The expression representing the index being accessed in the list) is appended to the *StringBuilder* on lines 731-736. This needs to be handled differently based on if the expression is an identifier or a value.

```

707 public override StringBuilder Visit(ListValueOfNode node)
708 {
709     var stringBuilder = new StringBuilder();
710     Symbol list = _symbolTable.Lookup(node.Identifier.Name,
        _currentBlock);
711
712     //Return type:
713     switch (getListType(list.Type))

```

```

714     {
715         case TypeEnum.number:
716         case TypeEnum.boolean:
717             stringBuilder.Append($"*(int *)");
718             break;
719         case TypeEnum._decimal:
720             stringBuilder.Append($"*(float *)");
721             break;
722         case TypeEnum.text:
723             stringBuilder.Append($"*(char **)");
724             break;
725     }
726
727     stringBuilder.Append($"ValueOfList({list.Name}, ");
728     List<string> arguments = new List<string>();
729
730     var expr1 = node.Arguments.Expressions[0];
731     if (expr1 is IdentifierNode)
732         arguments.Add($"&{Visit(expr1)}");
733     else
734         arguments.Add($" {Visit(expr1)}");
735
736     stringBuilder.AppendLine($"{{string.Join(", ", arguments)}}");
737
738     return stringBuilder;
739 }

```

**Listing 4.19:** Emitting the code for the list ValueOf node - CobraCompiler/Emitter.cs

## 4.5.2 Emitting foreach loop

The "repeat-for-each-loop" in PEAK+ has no equivalent data structure in C. Therefore one has been designed in the visitor method for the for-each node. First, a new scope is opened using curly brackets, as a for-each loop has its own scope. When the new scope has been opened, the *ForeachBlockNode* is visited. This visitor contains the rest of the implementation of the for-each loop. First, the list and the local variable in the *repeat for-each* loop are initialized with the values from the symbol table (See lines 1105-1106 in listing 4.20). Then a list with the name 'list' is initialized in the *StringBuilder* on line 1109 with a reference to the head of the original list. This temporary variable is used on the next line as the condition for a while loop on line 1110. This while loop runs until the end of the list is reached. Then a local variable is declared with the correct type and is assigned to the current value of the list on lines 1112-1129. The

block is then appended to the *StringBuilder* on line 1131-1150 where after the list is set to point at its next element on line 1152. After this, the while loop is closed.

```

1101 public override StringBuilder Visit(ForeachBlockNode node)
1102 {
1103     var stringBuilder = new StringBuilder();
1104     _currentBlock = node;
1105     var list = _symbolTable.Lookup(node.ListName, _currentBlock
1106         );
1107     var localVariable = _symbolTable.Lookup(node.LocalVariable.
1108         Identifier.Name, _currentBlock);
1109     _currentBlock = node;
1110     stringBuilder.AppendLine($"struct node *list = {list.Name};
1111         ");
1112     stringBuilder.AppendLine("while (list != NULL) {");
1113     stringBuilder.Append($"{{getTypeInC(localVariable.Type)} {
1114         localVariable.Name}");
1115     //Assign the current value in the list to the LocalVariable
1116     switch (localVariable.Type)
1117     {
1118         case TypeEnum.number:
1119             stringBuilder.AppendLine($" = *(int *)list->value;
1120                 ");
1121             break;
1122         case TypeEnum._decimal:
1123             stringBuilder.AppendLine($" = *(float *)list->value
1124                 ");
1125             break;
1126         case TypeEnum.text:
1127             stringBuilder.AppendLine($" = (char *)list->value;
1128                 ");
1129             break;
1130         default:
1131             stringBuilder.AppendLine($" = list->value;");
1132             break;
1133     }
1134     //Go through the commands in the block
1135     foreach (var command in node.Commands)
1136     {
1137         switch (command)
1138         {

```



```

1136         case DeclarationNode declarationNode:
1137             stringBuilder.Append(Visit(declarationNode));
1138             stringBuilder.AppendLine(";");
1139             break;
1140         case AssignNode assignNode:
1141             stringBuilder.Append(Visit(assignNode));
1142             break;
1143         case StatementNode statementNode:
1144             stringBuilder.Append(Visit(statementNode));
1145             break;
1146         default:
1147             throw new Exception($"Command was not valid");
1148     }
1149     _currentBlock = node;
1150 }
1151
1152 stringBuilder.AppendLine($"list = list->next;");
1153
1154 stringBuilder.AppendLine("}"); //Close While Loop
1155 _currentBlock = node;
1156 return stringBuilder;
1157 }

```

**Listing 4.20:** Emitting the code for the repeat for each loop block - CobraCompiler/Emitter.cs

### 4.5.3 Emitting functions

When attempting to make equivalent functions from our language to C, an issue arose. In PEAK+, you are able to create functions anywhere in the code, as long as the declaration is in the outer-most scope. This way, any variables declared before the function declaration are available inside the scope of the function. C's *main()* function corresponds to PEAK+ 's outer-most scope but in C, you cannot declare a function within another function.

Because of this, the AST had to be sorted before being passed to the code generation. This can be seen in listing 4.21 where the 'Commands' of the program node (the root node) are ordered by nodes that are not a function declaration on lines 74-75. The AST is sorted such that all function declarations appear first in the program. This way, the emitter can create all function declarations before entering the *main()* function.

```

74 ((ProgramNode)ast).Commands =
75     ((ProgramNode)ast).Commands.OrderBy(x => x is not
        FunctionDeclarationNode).ToList();

```

**Listing 4.21:** Sorting AST so function declarations appear at the top - CobraCompiler/Program.cs

Now we have to figure out how the declared functions get access to the variables which are declared before the function declaration in PEAK+ . This is done when visiting a function declaration in the symbol table by storing the types and names of all variables which are used, but not declared, in the function block. This is better described in section 4.4.1. An example of this written in source-code and target-code can be seen on listing 4.22 and 4.23.

```
0 | number example_var = 10;
1 | function example_func() return nothing
2 | {
3 |     example_var = 20;
4 | }
5 | call output(example_var);
```

**Listing 4.22:** Example of a function being declared after a variable has been declared. The function then uses the variable without passing it as an argument

```
1 | void example_func(int example_var)
2 | {
3 |     example_var = 20;
4 | }
5 | void main()
6 | {
7 |     int example_var = 10;
8 |     printf("%d\n", example_var);
9 | }
```

**Listing 4.23:** Example of the corresponding code from listing 4.22 written in C. The function gets declared before entering main and an additional argument is added for the variable used outside of the function

# Chapter 5

## Testing

In order to validate that the compiler is correct, unit, integration, and acceptance testing is conducted and explained in this chapter. Furthermore, a user testing plan is described.

Selected tests are displayed in this chapter, and the rest are displayed in appendix E.

### 5.1 Testing of the PEAK+ compiler

The testing is conducted by using the Arrange, Act & Assert (AAA) testing pattern. This is a pattern that helps make tests clear and understandable because of the separation between the setup, execution, and verification parts of the test. The Arrange section initializes objects and variables passed to the method being tested. The Act section invokes the said method with the arranged parameters. Lastly, the Assert section is the one that clarifies if the test is successful or not, by setting one or more conditions that, if true, means that the test is successful [41]. As a metric to get an understanding of which parts of the program are tested and which are not, a code coverage overview is made. As shown in figure 5.1, not all of the code is tested. This is due to time restraint or the code not being important to the compiler. This is for example the *prettyPrint()* function in the *BuildASTVistor* class, as it only acts as a visualization under development. The setup of testing is done by creating a whole new separate project inside the same solution in visual studio. That way we can access the necessary methods that have to be tested.

All tests are accessible in the project repository in GitHub [13].





Symbol ▾	Coverage (%)	Uncovered/Total Stmts.
▾  Total	<div><div style="width: 38%;"></div></div> 38%	2006/3253
▾ <input type="checkbox"/> CobraCompiler	<div><div style="width: 38%;"></div></div> 38%	2006/3253
>  CobraCompiler	<div><div style="width: 36%;"></div></div> 36%	1358/2114
>  BuildASTVisitor	<div><div style="width: 42%;"></div></div> 42%	584/1007
>  ASTNodes	<div><div style="width: 52%;"></div></div> 52%	64/132

Figure 5.1: Code coverage

## 5.2 Unit Test

The first level of testing conducted is unit testing. Unit testing is the smallest testable part of an application and the purpose of unit testing is to separate modules from their dependencies and test individual methods. The framework we use for testing is NUnit [26], which is a unit-testing framework for all .Net languages that support the AAA testing pattern. As mentioned in section 5.1, only some code is unit tested. This is due to most of the PEAK+ compiler consisting of visitor patterns which are easier to test using integration testing because of the number of functions integrated with each other. This is shown in section 5.3.

An example of a unit test can be seen in listing 5.1. The function being tested is *NewScope()* by asserting if a new scope had been added to the stack of scopes in the symbol table. The Arrange block creates new instances of *SymbolTable()* and *BlockNode()* on lines 57-58. In the Act stage, *NewScope()* in the symbol table is called and the block node is passed as an argument to the method. This happens on line 61. This should open a new scope containing the given block node. To test that *NewScope()* did push this new scope to the stack, we test that our *symbolTable.\_stackScopes* contains one scope on line 64. This will return true if *NewScope()* pushes a new scope onto the stack. We also test that the stack contains a scope with a block that is equal to *blockNode* on line 65. When running these tests, a green mark is displayed if all of the Assert methods return true. If this is the case, it can be concluded that these unit tests are successful, and in this case, the *NewScope()* method in listing 5.1 works as expected.

```

53 [Test]
54 public void NewScope_PushesScopeToStackScopes()
55 {
56     // Arrange
57     var symbolTable = new SymbolTable(new ErrorHandler());
58     var blockNode = new ASTNodes.BlockNode();
59
60     // Act

```

```

61     symbolTable.NewScope(blockNode);
62
63     // Assert
64     That(symbolTable._stackScopes, Has.Count.EqualTo(1));
65     That(symbolTable._stackScopes.Peek().Block, Is.EqualTo(
        blockNode));
66 }

```

**Listing 5.1:** Unit test of the `NewScope()` method that creates a new scope and tests if it pushes this scope to the stack - `PEAKCompilerTesting / SymbolTableUnitTest.cs`

## 5.3 Integration Test

As unit testing is for verifying the individual performance of modules and methods, integration testing has the purpose of verifying that they still exhibit the expected behavior as a group. This means that individual modules are combined and tested as a group to evaluate the compliance of the compiler. This section will test how the lexer, parser, AST builder, code generator, and so on work in conjunction.

The code in listing 5.2 is the integration test for `lookup()` in the symbol table. The `SymbolTableIntegrationTest` class contains all integration tests associated with the symbol table. The integration test in listing 5.2 starts by creating a new instance of the `SymbolTable` on line 9. Then a `BlockNode` named `blockNode1` gets instantiated on line 10. Then `NewScope()` in the symbol table is called with `blockNode1` as its argument on line 11. The `Insert()` method is called twice to insert "x" and "y" in the current scope, which is the one belonging to `blockNode1`. This happens on line 12-13. This concludes the Arrange stage. In the Act stage, we call `Lookup()` in the symbol table three times on line 16-18. This looks for "x", "y", and "z" in the current scope. These methods return objects of type `Symbol` which contains a name, reference, and type. These symbols are checked in the assert section to see if the names of the symbols match what is expected. "z" is not inserted in the symbol table and should therefore return `null`. These asserts happen on line 21-23.

```

3 public class SymbolTableIntegrationTest
4 {
5     [Test]
6     public void Lookup_ReturnsCorrectSymbol()
7     {
8         // Arrange
9         var symbolTable = new SymbolTable(new ErrorHandler());
10        var blockNode1 = new ASTNodes.BlockNode();
11        symbolTable.NewScope(blockNode1);

```

```

12     symbolTable.Insert("x", ASTNodes.TypeEnum.text,
13         blockNode1);
14     symbolTable.Insert("y", ASTNodes.TypeEnum.text,
15         blockNode1);
16
17     // Act
18     var result1 = symbolTable.Lookup("x", blockNode1);
19     var result2 = symbolTable.Lookup("y", blockNode1);
20     var result3 = symbolTable.Lookup("z", blockNode1);
21
22     // Assert
23     Assert.That(result1.Name, Is.EqualTo("x"));
24     Assert.That(result2.Name, Is.EqualTo("y"));
25     Assert.That(result3, Is.Null);
26 }
27 }

```

**Listing 5.2:** Integration test of the lookup method from the symbol table - PEAKCompilerTesting / SymbolTableIntegrationTest.cs

The second integration test shown in listing 5.3 is a visit function for visiting the program node in the code generation phase. Code generation is done by the *Emitter* class, which goes through the program, using the visitor pattern, and converts the program into the equivalent version in C. The C version is kept in a string builder which can then be tested. A string of PEAK+ code that is used for testing the emitter can be seen on line 85. The string declares a number variable *x* to the value 10. All the phases that lead up to the code generation, are then run in the rest of the arrange section. In the Act section, the function which visits the root node in the emitter is run on line 89. The generated C code is returned to a variable *sb* of type string builder. The expected string of C code is built on line 91-95, and are then tested for equivalence in the Assert section on line 98. The goal of this test is to make sure, the emitter returns the correct C program as a string.

```

82 public void EmitterVarDec()
83 {
84     // Arrange
85     var exprText = "number x = 10;";
86     ...
87
88     // Act
89     StringBuilder sb = new Emitter(st).Visit((ASTNodes.
90         ProgramNode)ast);
91
92     String testString = "#include <stdio.h>\n#include <stdlib.h>\n#include <string.h>\nchar* concat(const char *str1,

```

```

92         const char *str2) "
93         +
94         ...
95         +
96         "return curr_node->value;\n}\nvoid main(){\nint x = 10;\n}\n
97         n";
98
99         // Assert
100        Assert.That(sb.ToString(), Is.EqualTo(testString));
101    }

```

**Listing 5.3:** Integration test of the Emitter class used in the code generation phase - PEAKCompilerTesting / EmitterTest.cs

## 5.4 Acceptance Test

This section will cover the acceptance tests made to determine if the PEAK+ compiler generates the correct target code. This is done by running a number of PEAK+ code examples through our compiler and checking whether the resulting programs are as expected. These tests are based on the beginner exercises explained in section 3.3. If the code compiles correctly to C without errors, the C code is then compiled and run in order to see if the terminal prints the expected results. Furthermore, the generated C program is inspected to ensure it is correctly translated.

Each test contains the code written in PEAK+ , then the code compiled to C, and finally the result of compiling and executing the C code.

### 5.4.1 Code Example: Scope test

The code example in listing 5.4 is used to show that the scope rules defined in chapter 3 work as intended in PEAK+ . In the example, a number *y* is declared in the outer scope on line 1 with the value 10. Then *print()* is declared which prints *y* using *output()* on line 2-5. *output()* is then called with *y* in the outer scope on line 6. In the if statement with condition *true*, *print()* is called followed by *y* being outputted on lines 7-10. Then *y* is redeclared with value 12 on line 11 and then *print()* is called, followed by *y* being outputted on line 12-13. The *print()* functions declaration can only access the global *y* with value 10 and should therefore output 10, while the *output()* on line 13 should print 12. After the if-statement the *print()* and *output()* functions should output 10 on line 15-16. The result can be seen in figure 5.2.

**Source Code in PEAK+ :**

```

1  number y = 10;
2  function print() return nothing
3  {
4      call output(y);
5  }
6  call output(y);
7  if(true)
8  {
9      call print();
10     call output(y);
11     number y = 12;
12     call print();
13     call output(y);
14 }
15 call print();
16 call output(y);

```

**Listing 5.4:** Acceptance test scope code examples**Target code in C:**

In listing 5.5, the generated C code from the scope test program is displayed. From lines 1-4, the void function *print()* is defined. It takes a parameter *y*, which is then outputted in the function.

When first creating this test, an issue arose which was caused by translating the scope rules defined in PEAK+ 's semantics to code written in C. This issue was caused by the differences in starting points for PEAK+ and C. In both C and PEAK+ , functions can only be declared in the global scope. However, unlike in PEAK+ where the starting point is placed at the first line of code, C uses the function *main()* as its starting point. This results in a problem when declaring functions in PEAK+ , which is addressed in section 4.5.3.

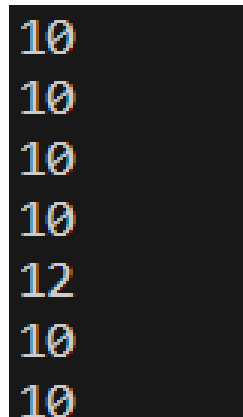
The original version is identical to the output shown in listing 5.5 except in the original, where nothing is used to distinguish between the first and second declarations of the variable *y*. The *print\_()* function on line 13 referenced the *y* variable initialized on line 12, but in reality, it should have referenced the *y* on line 6.

This issue required a fix to help distinguish between declared variables of the same name. The fix is described in section 4.4.1. The code snippet shown in listing 5.5 displays the resulting C code after the fix, and correlates with the program in PEAK+ . The result when executing the code can be seen in figure 5.2.



```
1 void print_(int y)
2 {
3     printf("%d\n", y);
4 }
5 void main() {
6     int y = 10;
7     printf("%d\n", y);
8     if (1)
9     {
10         print_(y);
11         printf("%d\n", y);
12         int y_ = 12;
13         print_(y);
14         printf("%d\n", y_);
15     }
16     print_(y);
17     printf("%d\n", y);
18 }
```

Listing 5.5: Acceptance test scope code examples in C

**Result:**

```
10
10
10
10
12
10
10
```

Figure 5.2: The result for the scope code example

### 5.4.2 Code Example: Update of variables

The code example in listing 5.6 is used to show that declaration and re-assignment of an existing variable works as intended. First, multiple different variables of different types are declared on lines 1-3, and  $n$  is then outputted on line 5. Afterward, the user is prompted for a new input for  $n$  on line 7. This new value of  $n$  is then outputted on line 9. A couple of examples of comments can be seen on lines 10-11, which are ignored when compiling to C. The new  $n$  should overwrite the original value of  $n$ .

#### Source Code in PEAK+ :

```

1  number n = 123;
2  text t = "test";
3  decimal d = 0.20123123947342375;
4  call output("Previous value is ");
5  call output (n);
6  call output(". Please input a new number");
7  n = call number input();
8  call output("New value is ");
9  call output(n);
10 comment: This is a comment;
11 comment: This is also a comment;

```

Listing 5.6: Acceptance test update of variables with input code examples

The C code in listing 5.7 corresponds to what is expected. First, the three different variables are declared on lines 3-5, whereafter three lines are printed to the console on lines 6-8. Next, *input()* is called. The input function is a custom helper function that is added to the emitter. It takes in a format and the variable size, scans a user input, and returns it. The *input()* function returns a void pointer, which needs to be cast to the correct pointer type and dereferenced to get the value. This can be seen on line 9. Note that the decimal value in listing 5.6 is rounded up when compiling to C. This is because float only being able to contain 7 decimals [1]. This was first discovered when running the code example in listing 5.7. This is touched upon in the discussion section 6.1.1. The result when executing the code is displayed in figure 5.3.

#### Target code in C:

```

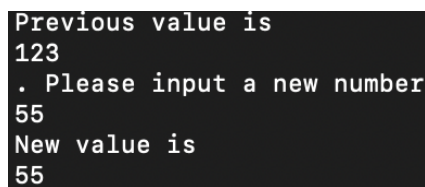
1  void main()
2  {
3  int n = 123;
4  char * t = "test";
5  float d = 0.20123124;
6  printf("%s\n", "Previous value is ");

```

```
7 | printf("%d\n", n);
8 | printf("%s\n", ". Please input a new number");
9 | n = *(int *)input("%d", sizeof(int));
10 | printf("%s\n", "New value is ");
11 | printf("%d\n", n);
12 | }
```

**Listing 5.7:** Acceptance test update of variables with input code examples in C

### Result:



```
Previous value is
123
. Please input a new number
55
New value is
55
```

**Figure 5.3:** The result for the variable update code example

Along with these examples, some other acceptance tests are also made. These can be seen in section E.1 in the appendix. A short description of these examples can be seen below:

- **Insertion-Sort:** This code example initializes a list of numbers and prints them. Afterward, this list is sorted using the 'SortNumberList()' which uses the insertion-sort algorithm. The new sorted list is then also printed.
- **Time-Conversion:** This code takes a number of seconds as input. By using division and a function 'mod()', which calculates modulo, it calculates how many weeks, days, hours, and minutes these seconds are equivalent to. These values are then printed.
- **GCD calculator:** This code calculates the 'greatest-common-divisor' of two numbers. this code includes 'GetMax()', 'GetMin()' and 'mod()'. In case the given input is wrong, the user is asked once again for an input. After printing GCD, the code gives the user the opportunity to write two new numbers.
- **Input Repeat:** This example declares a function called 'StartFunction()'. This function has a while-loop which runs while a boolean value is true. In the loop, the user is prompted and asked to write "start" to begin. The user is then asked to write how many times a message should be repeated. A message saying "This is a repeated message" is then repeated to the number requested by the user. Afterward, the user is prompted to write 'start' again.

- **Triangle Area:** This example declares a function 'triArea()' which calculates the area of a triangle. This function is then called and printed four times, all with different inputs.
- **Mathematical Expressions:** This example tests that all the mathematical expressions included in PEAK+ works as intended. This is done by declaring variables and using the infix mathematical expressions on the variables.
- **Missing semi error:** This code example is used to test if the correct error is being output when the code is missing a ";".
- **Wrong type:** This code example tests the output message for when an incorrect type has been used.
- **String concatenation:** This code example is used to test for string concatenation. this is done by creating a text variable and using the infix operator "+" between two strings, and afterward outputting the variable.
- **Zero division:** This code example is used to make sure that division by zero is not possible in PEAK+ and that it gives the correct output in the terminal.

### 5.4.3 User Testing

In order to test and get feedback on whether or not PEAK+ has succeeded to become a beginner-friendly programming language, a user testing plan is produced. First up, the tester has to be exposed to a number of small programs written in two different languages, one should be PEAK+ and the other should be in a non-beginner-friendly language as mentioned in section 2.3.3. The purpose of this is to test the readability of PEAK+ , by having the tester look through these programs and write down what the tester thinks the programs do.

Second, the user test will cover the usage of PEAK+ . The tester has to be introduced to PEAK+ functionality, and the basics of it, to be able to write a basic beginner programming exercise, like those explained in section 3.3, that the requirement for PEAK+ is build upon. In this test, we will observe if the tester has problems writing code in PEAK+ . This is done to test whether or not it is intuitive to learn and use the syntax. It has not been possible to get a user test conducted. This is due to lack of time, discussed in section 6.3.

# Chapter 6

## Discussion

This chapter will discuss the development of the PEAK+ compiler, as well as reflect on the extent to which PEAK+ meets the requirements.

### 6.1 Language Revisions

The goal with PEAK+ 's syntax is to balance the readable concepts from block-based languages, with the industry standards. The three criteria we followed in regard to designing a programming language were as described in section 3.1: Readability, Writability, and Reliability.

#### **Readability:**

Based on the analysis, it was decided that PEAK+ should have high readability because new concepts are harder to understand for beginners. After reflection, it can be asserted that this goal was met with PEAK+ because the syntax described in the Syntax Design section 3.4, is arguably highly readable because it is closely related to natural language. This is highlighted in table 6.1, where simplicity, Data Types, and Syntax design all are check-marked, under the readability column.

However, it could be argued that some parts of PEAK+ could be more readable. Because the goal for PEAK+ is to become a language for transitioning to industry standard languages, some of the syntax is less readable.

Taking the list helper function into consideration, the "*list*:add(*x*)" is more readable and related to the English language if it is instead written as: "add *x* to *list*". Another example of less readable syntax is the use of curly brackets to define scopes in PEAK+ . Alternatively, a more readable approach would have been to use "begin" and "end", like used in Quroum. However, our own experience when learning to program was that using curly brackets for defining scopes, was not difficult to understand.

Most popular programming language (except Python) uses curly brackets [10], and therefore curly brackets were included in PEAK+ .

### Writability:

The aim for PEAK+ was to have fairly low writability, as high writability is often associated with shorter syntax and faster-to-write features, which can lead to confusion for beginners. This goal was arguably met, as much of the syntax in PEAK+ was not compromised in terms of simplifying certain features. An example of this is the implementation of function declarations seen in semantic definition 3.20, where keywords such as *function* and *return* are used, in order to increase readability, but compromise writability. In the criteria table 6.1 below, the lack of writability is shown in the majority of crosses under the writability column.

### Reliability

We wanted high reliability in PEAK+ because a clear understanding of what is happening when programming will help beginner programmers develop. In order to achieve this, complex concepts such as pointers and exception handling, are excluded from PEAK+ . Additionally PEAK+ uses a static type binding, which promotes scope rule understanding. In general, these features were implemented as set out. Table 6.1 highlights that reliability was significant under multiple categories, as seen by the checkmarks shown under Type Checking, Restricted Aliasing, Simplicity, etc.

Based on these reflections the following criteria table 6.1 was created, which highlights how PEAK+ turned out in terms of the different criteria. The table was created based on how the PEAK+ turned out. A checkmark (✓) means PEAK+ expresses a high significance within that category, and a cross (X) means that PEAK+ lacks implementation or priority within that category.

	Readability	Writability	Reliability
Simplicity	✓	✓	✓
Orthogonality	X	X	X
Data Types	✓	X	✓
Syntax Design	✓	X	✓
Support for Abstraction		✓	✓
Expressivity		X	X
Type Checking			✓
Exception Handling			X
Restricted Aliasing			✓

**Table 6.1:** Language Criteria for PEAK+

Throughout the process, PEAK+ went through two iterations. The first iteration

was much simpler, but still took the majority of the time, simply because there was a learning curve to developing a compiler. Once the first iteration was finished, a template existed for implementing many of the additional features that were needed to meet the requirements set out in table 3.1.

### 6.1.1 Compiler revisions

#### Compiler Language

The first iteration compiled to C# as the target code. This was because initially we had more experience with C#, but through the process, it became more clear that C was a better language to target. This was realized with instances such as the static binding rules in C, as well as the general flexibility. Therefore the second iteration was developed with C as the target language. With these reflections in mind, the following subsection will describe in detail to what extent the requirements of PEAK+ have been met.

#### Decorated AST

In section 4.2 of the implementation chapter, it was stated that the PEAK+ compiler did not ever decorate the AST. When testing string concatenation in the emitter, this needed to change because of an issue. Adding two *text* values in an addition node has to be handled differently than adding numbers in the emitter. Therefore infix-expression nodes in the AST needed to store the type to which the node evaluates to, resulting in the AST being decorated. This happens in the type checker.

#### Decimal to float

During testing, we discovered, that values of type *decimal* with many decimal numbers, would be rounded down to 7 decimals in the target code. This is because when trying to store a *decimal* value in PEAK+ in the AST, it gets stored as a float, which automatically is rounded up to 7 decimals. Alternatively, it could have been stored as a double instead. Float was chosen as PEAK+ is not designed for creating large precise calculations. However, choosing double could have increased PEAK+ 's reliability as it might not be clear to the user that the *decimal* value is rounded up.

## 6.2 Requirement evaluation

All requirements in table 3.1 will be evaluated in order to assess whether PEAK+ has successfully achieved all of the desired requirements.

### Must have requirements evaluation

In this section, all the must-have requirements from table 3.1 will be assessed in order to evaluate if they have been fulfilled and that PEAK+ has all the features deemed essential.

- M1 - Declaration and assignment of number, decimal, text, and boolean variables.

The must-have requirement *M1* has been fulfilled by making it possible to declare and assign variables of the types: *number*, *decimal*, *text*, and *boolean*. An example of this can be seen in the code example section 5.4.2. In this code example, a *number*, *text*, and *decimal* variable are declared and give the correct output. For this reason requirement *M1* has been fulfilled.

- M2 - Display output and take keyboard input in the console.

This must-have requirement states that PEAK+ should be able to display output in the console and receive inputs. This has been done by creating the "*call output()*" and "*call input()*" methods. An example of this can be seen in listing 5.6 which shows that both input and output works as intended. Because of this the must-have requirement *M2* has been fulfilled

- M3 - Basic iterative and selective control structures

This must-have requirement says that PEAK+ should contain both iterative and selective control structures. The iterative control structures was added through the implementation of a while loop and for loop. An example of this can be seen in the appendix chapter in section E.1.1. The selective control structure has been implemented by the use of if-else statement. An example of an if-else statement can also be seen in the appendix chapter in section E.1.2. Because PEAK+ has both iterative and selective control structures the *M3* requirement has been fulfilled.

- M4 - Basic arithmetic & logical operations.

This must-have requirement requires PEAK+ to include basic mathematical operations. This is done by implementing the infix operators "+, -, \*, /" and parenthesis



operators "(" and ")" , and the logical operators "<, >, <=, >=, and, or, is, is not" (as mentioned in section 3.4.4. An example program of mathematical operations can be seen in the appendix chapter E.1 in figure E.1.5. In this code example all the mathematical operations have been implemented. This can be seen in the result of figure E.1.5. For this reason, the *M3* must-have requirement has been fulfilled.

- *M5* - Syntax which resembles concepts and terminology used in high school education.

As mentioned in table 3.1, the syntax has to resemble concepts and terminology used in high school, for example the use of fully spelled words and basic math symbols. This can be seen in the code examples in section 5.4 and the appendix chapter E.1. PEAK+ uses fully spelled words as *number* or *text*. By doing this the must-have requirement has been fulfilled.

- *M6* - Error messages at compile time.

This must-have requirement states that PEAK+ must have error messages at compile time. This has been done by outputting error-messages in the terminal during compilation. An example of this can be seen in the appendix chapter E.1 in figure E.1.6, where an error message is being displayed in the terminal due to a missing semicolon. Another example can also be seen in the appendix chapter E.1 in the figure E.1.7, displaying a type error, due to the type being *text*, but the value is of type *number*. Because of this the must-have requirement *M5* has been fulfilled.

### Should have requirement evaluation

In this section, all the should-have requirements from table 3.1 will be evaluated.

- *S1* - A for each loop.

This requirement states that PEAK+ should contain a way to create a for each loop similar to the one used in a language like C#. In PEAK+ , this has been done by using the keyword *repeat* followed by *for each*. The way it was implemented in C can be seen in the implementation chapter in section 4.5.2. An example of a for each loop in PEAK+ can be seen in the appendix chapter E.1 in figure E.1.1. By implementing the *for each* loop in PEAK+ , the should-have requirement *S1* has been fulfilled.

- *S2* - String concatenation.

This requirement states that PEAK+ should contain the possibility to do string concatenation. An example of how string concatenation has been achieved in PEAK+

can be seen in the appendix chapter E.1 in figure E.1.8. By being able to do string concatenation, the *S2* requirement has been fulfilled.

- *S3* - Declaration and assignment of lists

To complete this should-have requirement *PEAK+* should have the feature of declaring and assigning lists. The way the declaration and assignments of lists have been implemented can be seen in the implementation chapter 4 in section 4.5.1. Some code examples for declaring and assigning lists can be seen in the appendix chapter E.1 in figure E.1.1. As can be seen in the figure, by being able to declare and add values to a list, the should-have requirement *S3* has been fulfilled.

- *S4* - List helper functions like add and remove from the lists.

To complete the should-have requirement *S4*, list helper functions like *Add()* or *Remove()* should be added. Like the should-have requirement *S3*, the implementation for this can be seen in the implementation chapter 4 in section 4.5.1. In figure E.1.1 in the appendix chapter E.1, it is possible to see the helper function *Add()* in the code example. Under development, it was decided to replace the *Remove()* helper function with a *Replace()* instead and add the *ValueOf()* and *IndexOf()* operators. The reason for not including a *Remove()* operator was because of the difficulty of deleting an element in the middle of a list. This is because in *PEAK+* no value for null exists, so it was unclear what to do with a missing element, even though, a *Remove()* operator is very intuitive and well-suited for a beginner programmer. By having the *Add()*, *Replace()*, *ValueOf()*, and *IndexOf()* list helper functions are implemented in *PEAK+*. However, since the *Remove()* is not implemented the should-have requirement *S4* has not been fully completed.

### Could have requirement evaluation

- *C1* - Support for abstraction in the form of methods

This could-have requirement states that *PEAK+* could include the feature of being able to create methods. The implementation of methods can be seen in the implementation chapter 4 in section 4.5.3. An example of a method being created in *PEAK+* can be seen in figure 5.4.1 in the acceptance test chapter 5.4. For this reason the could-have requirement *C1* has been fulfilled.

Requirements overview						
Must have	M1	M2	M3	M4	M5	M6
Should have	S1	S2	S3	S4		
Could have	C1					

**Table 6.2:** An overview of each individual requirement.

Through this requirement evaluation all must-have, should-have & could-have requirements have been evaluated as well as fulfilled, except for the S4 requirement. By having most requirements fulfilled in the above-mentioned requirements it is deemed that PEAK+ has the most essential features implemented and usable in the language.

However, it is important to consider the development of these requirements was based on the problem analysis. Even though our own experiences also have been taken into consideration, as experienced programmers, we have a bias towards what we believe to be hard and easy concepts to understand in programming. This could result in us having created a language that is not actually beginner-friendly, as our understanding of a beginner-friendly language may differ from an actual beginner's perspective on what they find more educational. Because of this, it can be argued that we have no confirmation on the claim that PEAK+ is a good tool for beginner programmers to learn to code. Therefore user testing would be fairly wanted, however, this was not conducted due to time loss, as touched upon in section 5.4.3 and 6.5.1.

## 6.3 Time Management

One of the first decisions that we made was to use ANTLR for scanning and parsing the initial program. ANTLR's intuitive introduction and graphical interface caught our eye and seemed to be the best tool for first-time compiler developers. However, ANTLR turned out to be more difficult than expected to get used to and time consuming. In general, we needed to weigh the options on whether it was more worth to develop the parser ourselves or use a tool. In both scenarios, there was a learning curve with the process.

The decision to use ANTLR also led to the need of converting our CFG to LL(k) grammar. Ultimately the LL(k) parsing approach is less powerful than an LR(K) approach, however, it was easier to design the grammar so that it met the requirements for LL(k), which saved us time.

We decided to use an iterative model for the development of the compiler, and it was split into 2 iterations. Overall it was a good decision since it led us to a much stronger language. It was also a very natural decision in the context of developing a compiler since a compiler has concrete phases that all need to be updated to handle

new feature or fixes. However, the 2nd iteration did take more time than we initially wanted, which led to some shortcomings that we had to be aware of. Mainly it was the testing of the compiler, which had to be pushed back. This inevitably led us to have a less complete testing section than we would have wanted, meaning not everything in the compiler ended up being tested. It would have been smarter to leave out some less important features of the language, which delayed the completion time of the 2nd iteration. In general, it was hard to plan ahead in detailed time increments, because of how new everything within this project was.

## 6.4 Tests

Testing was done after the compiler was finished with the 2nd iteration, which is not the most effective testing strategy. Optimally we would have used a Test-driven development (TDD) design process in order to be able to more definitively ensure that the code written was correct, and also in order to be able to test that the changes made to the existing constructs would not break existing functionality [17]. Testing this way would give defined parameters to develop within, which potentially could have saved time. However, this idea of writing tests first is hard to do, because it requires a more structured approach, which can be difficult when developing a compiler for the first time. If tests were written beforehand, they would likely have taken just as long, if not longer to accurately write, and there is a good chance they would have had to be rewritten as the process moved along.

## 6.5 Future Work

As this project is subject to deadlines, some aspects of the project will not be fulfilled to perfection. This section will cover the future works of this project, and what particular components could be added or worked on, to improve the overall result.

### 6.5.1 User Test

A form of testing we did not get to cover was user testing. In order to better conclude that PEAK+ was suitable for the set-out target audience, it would have been advantageous to let a group of beginner programmers, preferably high school students, test PEAK+.

Regarding which users to test on, first-semester software students of Aalborg University seemed evident. This is because these students are motivated to learn programming and have gone to high-school, as well as having just started to learn to code. The test that should be conducted is explained in 5.4.3.

## 6.5.2 Code Functionality

While developing and testing the PEAK+ compiler, some of the inconveniences with PEAK+ were discovered, as well as missing features which were planned at first but were forgotten early on. These features made some of the programmer tasks we originally wanted beginner programmers to execute, impossible. This included:

### Indexing strings:

While writing tests, an attempt of making a program that was able to detect whether a given string input was a palindrome was made. This programming task came from when we were learning to program ourselves in the "imperative programming" course [27]. This required a for-loop to go through the string and check whether a character in the front, matched a character at the end of the string. String indexing is often used in other languages like Python, C#, or C and gives access to multiple ways of string manipulation. Alternatively, some of the list operations could have been made to also apply to PEAK+ 's *text* type, like *IndexOf()*.

### Built-in Operations:

In PEAK+ , many components have to be built from the ground up. An example of this is getting the length of a list, which in PEAK+ requires a for-each loop in order to iterate through the list and count the number of elements. For beginner programmers, this can seem tedious as many additional small functions are often needed in a program, which by default exist as constructs in other languages like C#, python and C. Examples of built-in operations could be: *number:Max()*, *text:Length()*, *list(text):Length()*, *Decimal:Round()*.

### Structs:

In order to teach the beginner programmer about object-like programming, structs were considered before making the first iteration. Structs would have made it possible for the beginner programmer to store values of certain types together. This would have made it possible for the beginner programmer to solve programming tasks such as creating a family tree, where each family member is of a "Person" struct. Alternatively, these kinds of tasks could be solved in PEAK+ using multiple lists and indexes. For example a *list(text) family* where [0] = "your name", [1] = "mom's name" and [2] = "dad's name" and a separate list of *list(boolean) isMale* where [0] = true, [1] = false.

### Typecasting:

The ability to typecast is also something that was discussed early on. This was one of the features which was only going to be added if any extra time was left. In PEAK+

, you are not able to convert for example *decimal* to *numbers* or *text* to *numbers*. Being able to cast a string to an integer is often used in programming tasks. When learning about programming in the "imperative programming" course, we had a task, about calculating the overall scores of football teams. As input, a large text file about football scores was provided, which had to be converted to numbers in order to calculate the total scores [27]. In order to solve this task, typecasting would need to be available to the programmer. Alternatively, you can sometimes work around explicit typecasting in PEAK+ . For example, by adding a *number* together with "0.0", you would receive a *decimal* value.

### **Recursive functions:**

Recursive functions were never discussed but are something that is often used in beginner tasks in order to teach the concept of recursion. Given more time, the recursive functions should be added to PEAK+ , as it is an important concept in programming. The palindrome task mentioned earlier when talking about indexing strings also applies here. In the original task from the "imperative programming" course [27], the goal was to solve this task first without using recursion, and then with, in order to see the difference.

### **Arithmetic Operators:**

Including a modulo arithmetic operator was never discussed and therefore forgotten. As modulo is an important tool in programming it should have been included, as the alternative is to create a *mod()* function, calculating the modulo of two numbers. This is an extra step in PEAK+ but is provided in other languages like C#, C, and Python. An exponent operator using the symbol  $\hat{}$  was discussed shortly, as when the group recalled using math tools in high school, this way of creating exponents was natural for them. With this, the transition from high-school maths to programming would be smoother.

# Chapter 7

## Conclusion

This report details the development of PEAK+ a programming language aimed towards introducing beginner programmers to text-based programming and some of the concepts of the industry standards, in an educational format. PEAK+ has fundamental features, such as basic arithmetic, control structures, abstraction through functions, etc. with high readability. The challenge of designing PEAK+ was a balancing act between using readable concepts from block-based languages, whilst still designing features that introduce users to the industry standards for programming languages. Based on this initial problem, the following problem statement was defined.

*How can a text-based programming language be developed for beginner programmers, using readable concepts from block-based languages, while focusing on facilitating the transition to programming languages used in the industry?*

Based on the requirements described in the problem statement section 2.5, which aims towards creating a successful compiler for PEAK+ , it can be concluded, that the parts and phases of a compiler that are required have been successfully implemented, tested and described. This includes but is not limited to the creation of a lexer, parser, AST design, correct scope rules, successful type checking, symbol table creation, code generation, testing as well as defining operational semantics for PEAK+ .

Finally, It can be concluded that PEAK+ has successfully met most requirements defined in section 3.3 table 3.1. Unfortunately, because of the lack of user testing, PEAK+ 's ability to serve as a beginner-friendly language cannot be entirely concluded as successful in a practical manner. Although the requirements are built upon the analysis, it is difficult to disregard the bias of an experienced programmer, compared to the mind of a beginner programmer. However, the analysis provides a blueprint of how PEAK+ should be designed in order to cater to beginner programmers in high school. Hence it can be concluded that PEAK+ has successfully fulfilled the problem statement.





# Bibliography

- [1] Ihechikara Vincent Abba. *Double VS Float in C++ – The Difference Between Floats and Doubles*. URL: <https://www.freecodecamp.org/news/double-vs-float-in-cpp-the-difference-between-floats-and-doubles/>.
- [2] *About Scratch*. URL: <https://scratch.mit.edu/about>.
- [3] Adobe. *Project Sprints*. URL: <https://business.adobe.com/blog/basics/sprints>.
- [4] *All About Scratch - The World's Most Popular Programming Language For Kids*. URL: <https://www.codetigers.com/post/all-about-scratch-the-world-s-most-popular-programming-language-for-kids>.
- [5] Atlassian. *ChatGPT*. URL: <https://www.atlassian.com/software/jira>.
- [6] Atlassian. *What is version control*. URL: <https://www.atlassian.com/git/tutorials/what-is-version-control>.
- [7] Giorgio Bacci. "Syntax and Semantics Basic principles of Operational Semantics". 2023.
- [8] Giorgio Bacci. "Syntax and Semantics Blocks and Procedures". 2023.
- [9] Giorgio Bacci. "Syntax and Semantics Type Systems". 2023.
- [10] Stephen Cass. *Top Programming Languages 2022*. URL: <https://spectrum.ieee.org/top-programming-languages-2022>.
- [11] Richard J. LeBlanc Jr. Charles N. Fischer Ron K. Cytron. *Crafting a Compiler*. Addison Wesley, 2009.
- [12] Chin Soon Cheah. *Factors Contributing to the Difficulties in Teaching and Learning of Computer Programming: A Literature Review*. URL: <https://www.cedtech.net/download/factors-contributing-to-the-difficulties-in-teaching-and-learning-of-computer-programming-a-8247.pdf>.
- [13] CS-23-SW-4-01. *P4-project GitHub Repository*. URL: <https://github.com/thomasbjeldbak/P4-project>.
- [14] DoIt. *Quorum: An Accessible Programming Language*. URL: <https://www.washington.edu/doit/videos/index.php?vid=76>.

- [15] FunTechBlog. *How Does Block-Based Programming Make Coding Easier?* URL: <https://funtech.co.uk/latest/how-does-block-based-programming-make-coding-easier>.
- [16] GeekForGeeks. *Difference Between Imperative and Declarative Programming*. URL: <https://www.geeksforgeeks.org/difference-between-imperative-and-declarative-programming/>.
- [17] Matt Heusser. *test-driven development (TDD)*. URL: <https://www.techtarget.com/searchsoftwarequality/definition/test-driven-development>.
- [18] Hans Hüttel. *Transitions and Trees*. Cambridge University Press, 2010.
- [19] IBM. *Metacharacters*. URL: <https://www.ibm.com/docs/en/informix-servers/14.10?topic=matching-metacharacters>.
- [20] Vasyl Lagutin. *Why Are There So Many Programming Languages?* URL: <https://www.freecodecamp.org/news/why-are-there-so-many-programming-languages/>.
- [21] Eagan Location. *WHAT IS PEMDAS?* URL: <https://www.mathnasium.com/eagan/news/what-pemdas-e>.
- [22] Majed Marji et al. *Learn to program with Scratch: A Visual Introduction to programming with games, Art, science, and math*. No Starch Press, 2014. URL: <https://nostarch.com/download/samples/Learn-Scratch-05.pdf>.
- [23] Stephan Miller. *20 Code Challenges To Put What You're Learning to the Test*. URL: <https://www.codecademy.com/resources/blog/20-code-challenges/>.
- [24] Mozilla. *Dynamic typing*. URL: [https://developer.mozilla.org/en-US/docs/Glossary/Dynamic\\_typing](https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing).
- [25] ngwes. *Programming Language Evaluation Criteria Part 2: Writability, Reliability, Cost*. URL: <https://medium.com/@thisisfordeveloper/programming-language-evaluation-criteria-part-2-writability-reliability-cost-6c3029c9d957>.
- [26] NUnit. URL: <https://nunit.org/>.
- [27] Kurt Nørmark. *Imperativ Programmering Efteråret 2021*. URL: <https://www.moodle.aau.dk/course/view.php?id=39338#section-0>.
- [28] Python. *The Python Language Reference*. URL: <https://docs.python.org/3/reference/index.html>.
- [29] PythonBasics. *Learn Python Programming*. URL: <https://pythonbasics.org/exercises/>.
- [30] Quorum. *The Quorum Programming Language*. URL: <https://quorumlanguage.com/evidence.html>.
- [31] Scratch. *Scratch User Interface*. URL: [https://en.scratch-wiki.info/wiki/User\\_Interface](https://en.scratch-wiki.info/wiki/User_Interface).

- [32] Robert W Sebesta. *Concepts of programming languages*. Pearson Education Limited, 2016.
- [33] Charles P. Shelton. *Exception Handling*. URL: [https://users.ece.cmu.edu/~koopman/des\\_s99/exceptions/](https://users.ece.cmu.edu/~koopman/des_s99/exceptions/).
- [34] Amrit Pal Singh. *5 Reasons Why You Should Learn C As Your First Language*. URL: <https://levelup.gitconnected.com/5-reasons-why-a-you-should-learn-c-as-your-first-language-7db054620bc0>.
- [35] University of Texas. *Why Learn Python? Five Reasons to Start Programming With Python*. URL: <https://techbootcamps.utexas.edu/blog/why-learn-python-get-started-programming/>.
- [36] Bent Thomsen. "Languages and Compilers, Context Free Grammars". 2023.
- [37] Bent Thomsen. "Languages and Compilers, The ac language and compiler". 2023.
- [38] Gabriele Tomassetti.
- [39] *Top 10 Block Programming tools*. URL: <https://thetechnoknowledge.com/top-10-block-programming-tools/>.
- [40] Erik Trautman. *Why Learning to Code is So Damn Hard*. URL: <https://www.thinkful.com/blog/why-learning-to-code-is-so-damn-hard/>.
- [41] *Unit test basics*. URL: <https://learn.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2022>.
- [42] Vanshinka. *Why is Programming important? The importance of computer programming explained*. URL: <https://codedamn.com/news/programming/why-is-programming-important-2>.
- [43] Michael Lvov Vladyslav Kruglyk. *Choosing the First Educational Programming Language*. URL: <https://ceur-ws.org/Vol-848/ICTERI-2012-CEUR-WS-paper-37-p-188-198.pdf>.
- [44] W3resource. *Python Syntax*. URL: <https://www.w3resource.com/python/python-syntax.php>.
- [45] David Weintrop. *Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms*. URL: <https://www.sciencedirect.com/science/article/pii/S036013151930199X#bib15>.
- [46] Wikipedia.
- [47] Wikipedia. *C programming language*. URL: [https://en.wikipedia.org/wiki/C\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/C_(programming_language)).
- [48] Wikipedia. *Functional Programming Paradigms*. URL: [https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming).

- [49] Wikipedia. *Overleaf*. URL: <https://en.wikipedia.org/wiki/Overleaf>.
- [50] Wikipedia. *Programming Paradigms*. URL: [https://en.wikipedia.org/wiki/Programming\\_paradigm](https://en.wikipedia.org/wiki/Programming_paradigm).

# Appendix A

## CFG

```
1 Program -> Cmds
2 Cmds -> Cmd Cmds | EPSILON
3 Cmd -> Stmt | Dcl
4 Dcl -> Type Id Ass ;
5 Ass -> = Expr | EPSILON
6 Stmt -> Id = Expr ; | CtrlStrct | ListStmt ; | FuncDef |
    FuncCall ; | CommentStmt ; | return Type ;
7 Expr -> LogicOr OprOr
8 OprOr -> or LogicOr OprOr | EPSILON
9 LogicOr -> LogicAnd OprAnd
10 OprAnd -> and LogicAnd OprAnd | EPSILON
11 LogicAnd -> Equal OprEq1
12 OprEq1 -> is Equal OprEq1 | is not Equal OprEq1 | EPSILON
13 Equal -> Bool OprBool
14 OprBool -> > Bool OprBool | < Bool OprBool | >= Bool OprBool |
    <= Bool OprBool | EPSILON
15 Bool -> Term OprExpr
16 OprExpr -> + Term OprExpr | - Term OprExpr | EPSILON
17 Term -> Factor OprTerm
18 OprTerm -> * Factor OprTerm | / Factor OprTerm | EPSILON
19 Factor -> ( Expr ) | FuncCall | ListOprExpr | Num | Decimal |
    String | Id | Boolean
20 Block -> { Cmds }
21 CommentStmt -> comment: StringChars
22 CtrlStrct -> IfStmt | Loop
23 IfStmt -> if ( Expr ) Block ElseIfStmt
24 ElseIfStmt -> else if ( Expr ) Block ElseIfStmt | Else |
    EPSILON
25 Else -> else Block
26 Loop -> repeat Loops
27 Loops -> LoopStmt | WhileStmt | ForeachStmt
```

```

28 LoopStmt -> ( Expr ) times Block
29 WhileStmt -> while ( Expr ) Block
30 ForeachStmt -> for each ( Type Id in Id ) Block
31 ListStmt -> ListOpr | ListOprExpr
32 ListOpr -> Id : Add ( ArgList ) | Id : Replace ( ArgList )
33 ListOprExpr -> Id : IndexOf ( Expr ) | Id : ValueOf ( ArgList )
34 FuncCall -> call Id ( ArgList ) | call output ( ArgList ) |
    call Type input ( ArgList )
35 FuncDef -> function Id ( ParamList ) FuncReturn Block
36 FuncReturn -> return FuncReturnType
37 FuncReturnType -> Type | nothing
38 ParamList -> Param ParamTail | EPSILON
39 ParamTail -> , Param ParamTail | EPSILON
40 Param -> Type Id
41 ArgList -> Expr ArgTail | EPSILON
42 ArgTail -> , Expr ArgTail | EPSILON
43 Boolean -> true | false
44 Type -> boolean | text | number | decimal | list ( Type )
45 Id -> Letter Chars
46 String -> ' ' StringChars ' '
47 StringChars -> StringChar StringChars | EPSILON
48 StringChar -> Char | SpecialChar
49 Chars -> Char Chars | EPSILON
50 Char -> Digit | Letter
51 Letters -> Letter Letters | EPSILON
52 Letter -> a ... z | A ... Z | _
53 Decimal -> Num . Num
54 Num -> Digit Digits
55 Digits -> Digit Digits | EPSILON
56 Digit -> 0 ... 9
57 SpecialChar -> ! | @ | # |   |   | $ | % | & | / | ( | ) | {
    | } | [ | ] | = | ? | + | - | ` | ^ | ~ | * | , | . | ~ |
    |

```

**Listing A.1:** Context-free grammar for PEAK+ in BNF form

# Appendix B

## ANTLR CFG + LEXER

```
1  program: cmds;
2  cmds: cmd cmds | /*epsilon*/;
3  cmd: stmt | dcl;
4  dcl: type ID ass SEMI;
5  ass: ASSIGN expr | /*epsilon*/;
6  stmt: ID ASSIGN expr SEMI |
7  ctrlStrct |
8  listStmt SEMI |
9  funcDef |
10 funcCall SEMI |
11 commentStmt SEMI |
12 RETURN type SEMI;
13 expr: logicOr oprOr;
14 oprOr: OR logicOr oprOr | /*epsilon*/;
15 logicOr: logicAnd oprAnd;
16 oprAnd: AND logicAnd oprAnd | /*epsilon*/;
17 logicAnd: equal oprEql;
18 oprEql: EQUAL equal oprEql | NOT equal oprEql | /*epsilon*/;
19 equal: bool oprBool;
20 oprBool: GREAT bool oprBool | LESS bool oprBool |
21 GREATERQL bool oprBool | LESSEQL bool oprBool | /*epsilon*/;
22 bool: term oprExpr;
23 oprExpr: ADD term oprExpr | SUB term oprExpr | /*epsilon*/;
24 term: factor oprTerm;
25 oprTerm: MUL factor oprTerm | DIV factor oprTerm | /*epsilon*/;
26 factor: LPAREN expr RPAREN | funcCall | listOprExpr | INT | DEC
    | STR | ID | boolean;
27 block: LCURLY cmds RCURLY;
28 commentStmt: COMM;
29 ctrlStrct: ifStmt | loop;
30 ifStmt: IF LPAREN expr RPAREN block elseIfStmt;
```

```

31 | elseIfStmt: ELSE IF LPAREN expr RPAREN block elseIfStmt | else
    |
32 | /*epsilon*/;
33 | else: ELSE block;
34 | loop: REPEAT loops;
35 | loops: loopStmt | whileStmt | foreachStmt;
36 | loopStmt: LPAREN expr RPAREN TIMES block;
37 | whileStmt: WHILE LPAREN expr RPAREN block;
38 | foreachStmt: FOREACH LPAREN type ID IN ID RPAREN block;
39 | listStmt: listOpr | listOprExpr;
40 | listOpr: ID COLON LISTADD LPAREN argList RPAREN |
41 | ID COLON LISTDEL LPAREN argList RPAREN;
42 | listOprExpr: ID COLON LISTIDXOF LPAREN argList RPAREN |
43 | ID COLON LISTVALOF LPAREN argList RPAREN;
44 | funcCall: CALL ID LPAREN argList RPAREN |
45 | CALL PRINT LPAREN argList RPAREN |
46 | CALL type SCAN LPAREN argList RPAREN;
47 | funcDef: FUNCTION ID LPAREN paramList RPAREN funcReturn block;
48 | funcReturn: RETURN funcReturnType;
49 | funcReturnType: type | NOTHING;
50 | paramList: param paramTail | /*epsilon*/;
51 | paramTail: COMMA param paramTail | /*epsilon*/;
52 | param: type ID;
53 | argList: expr argTail | /*epsilon*/;
54 | argTail: COMMA expr argTail | /*epsilon*/;
55 | boolean: TRUE | FALSE;
56 | type: BOOL | TEXT | NUM | LIST LPAREN type RPAREN | DECIMAL;

```

**Listing B.1:** Context-free grammar for PEAK+ used in ANTLR**Listing B.2:** Lexer for PEAK+ used in the ANTLR CFG

```

1 | OR: 'or';
2 | AND: 'and';
3 | EQUAL: 'is';
4 | NOT: 'is not';
5 | GREAT: '>';
6 | LESS: '<';
7 | GREATEREQ: '>=';
8 | LESSEQL: '<=';
9 |
10 | ASSIGN : '=' ;
11 | COMMA : ',' ;
12 | SEMI : ';' ;
13 | COLON : ':' ;
14 | LPAREN : '(' ;

```



```

15 RPAREN : ')' ;
16 LCURLY : '{' ;
17 RCURLY : '}' ;
18 TRUE: 'true';
19 FALSE: 'false';
20
21 ADD: '+';
22 SUB: '-';
23 MUL: '*';
24 DIV: '/';
25 BOOL: 'boolean';
26 TEXT: 'text';
27 NUM: 'number';
28 DECIMAL: 'decimal';
29 NOTHING: 'nothing';
30 LIST: 'list';
31 QUOTE: '"';
32 IF: 'if';
33 ELSE: 'else';
34 REPEAT: 'repeat';
35 TIMES: 'times';
36 WHILE: 'while';
37 FOREACH: 'for each';
38 IN: 'in';
39 FUNCTION: 'function';
40 RETURN: 'return';
41 CALL: 'call';
42 PRINT: 'output';
43 SCAN: 'input';
44 COMMENT: 'comment: ';
45
46 LISTADD: 'Add';
47 LISTIDXOF: 'IndexOf';
48 LISTREPLACE: 'Replace';
49 LISTVALOF: 'ValueOf';
50
51 COMM: 'comment: '~(';'')*';
52 STR: '"" (~'')* ' ';
53 DEC: ('+' | '-')? [0-9]+'.'[0-9]+;
54 INT: ('+' | '-')? [0-9]+ ;
55 ID: [a-zA-Z_][a-zA-Z_0-9]* ;
56 WS: [ \t\n\r\f]+ -> skip ;

```



# Appendix C

## Type System

This appendix acts as a continuation of the Type system defined in section 3.5.2

### C.1 Expression

$$(NUMEXP_{TS}) E \vdash n : \text{number} \quad (C.1)$$

$$(DECEXP_{TS}) E \vdash d : \text{decimal} \quad (C.2)$$

$$(TEXTEXP_{TS}) E \vdash txt : \text{text} \quad (C.3)$$

$$(BOOLEXP1_{TS}) E \vdash \mathbb{T} : \text{boolean} \quad (C.4)$$

$$(BOOLEXP2_{TS}) E \vdash \mathbb{F} : \text{boolean} \quad (C.5)$$

$$(VAREXP_{TS}) \frac{E(x) = T}{E \vdash x : T} \quad (C.6)$$

$$(PAREXP_{TS}) \frac{e : B}{E \vdash (e) : B} \quad (C.7)$$

$$(LISTSEXP_{TS}) \frac{E \vdash e : B}{E \vdash \text{list}(B) : \text{ok}} \text{ for } e \in L \quad (C.8)$$

$$(DEC * 1_{TS}) \frac{E \vdash e_1 : \text{decimal} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 * e_2 : \text{decimal}} \quad \text{where } * \in \{+, *, -, /\} \quad (C.9)$$

$$(DEC * 2_{TS}) \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{decimal}}{E \vdash e_1 * e_2 : \text{decimal}} \quad \text{where } * \in \{+, *, -, /\} \quad (C.10)$$

$$(NUMDIV_{TS}) \frac{E \vdash e_1 : \text{number} \quad E \vdash e_2 : \text{number}}{E \vdash e_1 / e_2 : \text{decimal}} \quad (C.11)$$

$$(BOOL*_{TS}) \frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \star e_2 : \text{boolean}} \quad \text{where } \star \in \{\text{and, or, is, is not}\} \quad (C.12)$$

$$(TXT_{TS}) \frac{E \vdash e_1 : \text{text} \quad E \vdash e_2 : \text{text}}{E \vdash e_1 + e_2 : \text{text}} \quad (C.13)$$

## C.2 Statements

$$(LISTSTM1_{TS}) \frac{E \vdash x : \text{list}(B) \quad E \vdash (x_1, \dots, x_k) : B}{E \vdash x : \text{Add}(x_1, \dots, x_k) : \text{ok}} \quad (C.14)$$

$$(LISTSTM2_{TS}) \frac{E \vdash x : \text{list}(B) \quad E \vdash x_1 : B \quad E \vdash n : \text{number}}{E \vdash x : \text{Insert}(x_1, n) : \text{ok}} \quad (C.15)$$

$$(LISTSTM3_{TS}) \frac{E \vdash x : \text{list}(B) \quad E \vdash n : \text{number}}{E \vdash x : \text{ValueOf}(n) : B} \quad (C.16)$$

$$(LISTSTM4_{TS}) \frac{E \vdash x : L \quad E \vdash e : B}{E \vdash x : \text{IndexOf}(e) : \text{number}} \quad (C.17)$$

$$(LISTSTM4_{TS}) \frac{E \vdash x : L \quad E \vdash n : \text{number}}{E \vdash x : \text{Delete}(n) : \text{ok}} \quad (C.18)$$

$$(IFSTM1_{TS}) \frac{E \vdash e : \text{boolean} \quad E \vdash S : \text{ok}}{E \vdash \text{if}(e) \{S\} : \text{ok}} \quad (C.19)$$

$$(IFSTM2_{TS}) \frac{E \vdash e : \text{boolean} \quad E \vdash S_1, S_2 : \text{ok}}{E \vdash \text{if}(e) \{S_1\} \text{ else } \{S_2\} : \text{ok}} \quad (C.20)$$

$$(REPEATSTM1_{TS}) \frac{E \vdash n : \text{number} \quad E \vdash S : \text{ok}}{E \vdash \text{repeat}(n) \text{ times } \{S\} : \text{ok}} \quad (C.21)$$

$$(REPEATSTM2_{TS}) \frac{E \vdash e : \text{boolean} \quad E \vdash S : \text{ok}}{E \vdash \text{repeat while}(e) \{S\} : \text{ok}} \quad (C.22)$$

$$(REPEATSTM3_{TS}) \frac{E \vdash x_1 : \text{list}(B) \quad E \vdash x : B \quad E \vdash S : \text{ok}}{E \vdash \text{repeat foreach}(B \text{ in } x_1) \{S\} : \text{ok}} \quad (C.23)$$

$$(RETSTM1_{TS}) \frac{E \vdash f : (e_1 : T_1, \dots, e_k : T_k \mapsto T) \quad E \vdash e : T}{E \vdash \text{return } e : \text{ok}} \quad (C.24)$$

$$(RETSTM2_{TS}) \frac{E \vdash f : (e_1 : T_1, \dots, e_k : T_k \mapsto \varepsilon)}{E \vdash \text{return} : \text{ok}} \quad (C.25)$$

# Appendix D

## Structural Operational Semantics

### Expressions

$$(NUM_{BS}) \frac{n \rightarrow_{\mathbf{Num}} v}{\sigma \circ Env_v \vdash n \rightarrow_{\mathbf{Exp}} v} \quad (D.1)$$

$$(VAR_{BS}) \frac{\sigma(Env_v(x)) = v}{\sigma \circ Env_v \vdash x \rightarrow_{\mathbf{Exp}} v} \text{ if } Env_v[x \rightarrow l] \text{ and } \sigma[l \rightarrow v] \quad (D.2)$$

$$(\star_{BS}) \frac{\sigma \circ Env_v \vdash e_1 \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash e_2 \rightarrow_{\mathbf{Exp}} v_2}{\sigma \circ Env_v \vdash e_1 \star e_2 \rightarrow_{\mathbf{Exp}} v_1 \star v_2} \quad (D.3)$$

Where  $\star \in (*, -, >, \geq, \leq, <, \text{and}, \text{or}, \text{is}, \text{is not})$

$$(PLUSNUM_{BS}) \frac{\sigma \circ Env_v \vdash e_1 \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash e_2 \rightarrow_{\mathbf{Exp}} v_2}{\sigma \circ Env_v \vdash e_1 + e_2 \rightarrow_{\mathbf{Exp}} v_1 + v_2} \quad (D.4)$$

Where  $v_1, v_2 \in \mathbf{Num} \cup \mathbf{Dec}$

$$(PLUSTXT_{BS}) \frac{\sigma \circ Env_v \vdash e_1 \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash e_2 \rightarrow_{\mathbf{Exp}} v_2}{\sigma \circ Env_v \vdash e_1 + e_2 \rightarrow_{\mathbf{Exp}} v_1 v_2} \quad (D.5)$$

Where  $v_1, v_2 \in \mathbf{Txt}$

$$(DIV_{BS}) \frac{\sigma \circ Env_v \vdash e_1 \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash e_2 \rightarrow_{\mathbf{Exp}} v_2}{\sigma \circ Env_v \vdash e_1 / e_2 \rightarrow_{\mathbf{Exp}} v_1 / v_2} \quad (D.6)$$

Where  $v_2 \neq 0$

$$(PAR_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v}{\sigma \circ Env_v \vdash (e) \rightarrow_{\mathbf{Exp}} v} \quad (D.7)$$

$$(LISTVALUEOF_{BS}) \frac{n \rightarrow_{\mathbf{Num}} v_1 \quad \sigma \circ Env_v \vdash x(v_1) = v_2}{\sigma \circ Env_v \vdash x : \text{valueof}(n) \rightarrow_{\mathbf{Exp}} v_2} \quad (D.8)$$

Where  $x(n)$  is a function that returns a value given the index of the list of  $x$

$$(LISTINDEXOF_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v_1 \quad \sigma \circ Env_v \vdash x(v_1) = v_2}{\sigma \circ Env_v \vdash x : indexof(e) \rightarrow_{\mathbf{Num}} v_2} \quad (D.9)$$

Where  $x(e)$  is a function that returns a natural number  
given an expression which is contained in the list  $x$

### Declarations

$$(FUNCDEC_{BS}) \frac{Env_v, Env_f[f \mapsto (S, x_1, \dots, x_k, Env_v)] \vdash_l \langle S_2, \sigma \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle \text{function } f(T_1x_1, \dots, T_kx_k) \text{ return } T\{S_1; \text{return } e\}; S_2, \sigma \rangle \rightarrow \sigma'} \quad (D.10)$$

Where  $S = S_1; \text{return } e$

$$(VARDEC1_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v \quad Env_v[x \mapsto l], Env_f \vdash_{nxt(l)} \langle S, \sigma[l \mapsto v] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle T x = e; S, \sigma \rangle \rightarrow \sigma'} \quad (D.11)$$

$$(VARDEC2_{BS}) \frac{Env_v[x \mapsto l], Env_f \vdash_{nxt(l)} \langle S, \sigma[l \mapsto d(T)] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle T x; S, \sigma \rangle \rightarrow \sigma'} \quad (D.12)$$

Where

$d(\text{number})$	$=$	0
$d(\text{text})$	$=$	""
$d(\text{boolean})$	$=$	$\mathbb{F}$
$d(\text{decimal})$	$=$	0.0
$d(\text{list}(B))$	$=$	[]

**Table D.1:** Default values for all variable types

### Statements

$$(COMP_{BS}) \frac{Env_v, Env_f \vdash_l \langle S_1, \sigma \rangle \rightarrow \sigma' \quad Env_v, Env_f \vdash_l \langle S_2, \sigma' \rangle \rightarrow \sigma''}{Env_v, Env_f \vdash_l \langle S_1; S_2, \sigma \rangle \rightarrow \sigma''} \quad (D.13)$$

$$(ASS_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v}{Env_v, Env_f \vdash_l \langle x = e, \sigma \rangle \rightarrow \sigma[Env_v(x) \mapsto v]} \quad (D.14)$$

$$(LISTADD_{BS}) \frac{\sigma \circ Env_v \vdash e_i \rightarrow_{\mathbf{Exp}} v_i}{Env_v, Env_f \vdash_l \langle x : add(e_1, \dots, e_k), \sigma \rangle \rightarrow \sigma[Env_v(x) \mapsto L']} \quad (D.15)$$

Where  $i = 1 \dots k$  and  $Env_v[x \mapsto l]$   
 and  $L' = L$  where the values  $v_i$  is appended to the end of the list

$$(LISTREPLACE_{BS}) \frac{n \rightarrow_{\mathbf{Num}} v_1 \quad \sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} v_2}{Env_v, Env_f \vdash_l \langle x : replace(e, n), \sigma \rangle \rightarrow \sigma[Env_v(x) \mapsto L']}$$

Where  $L' = L$  where  $v_2$  is placed at the index of  $v_1$

(D.16)

$$(FUNCEXP_{BS}) \frac{Env_f(f) = (S, x_1, \dots, x_k, Env'_v) \quad \sigma \circ Env_v \vdash e_i \rightarrow_{\mathbf{Exp}} v_i \quad Env'_v[x_1 \mapsto l_1][x_2 \mapsto l_2] \dots [x_k \mapsto l_k], Env_f \vdash_{nxt(l)} \langle S, \sigma[l_1 \mapsto v_1] \dots [l_k \mapsto v_k] \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle x = call \ f(e_1, \dots, e_k), \sigma \rangle \rightarrow \sigma'}$$

(D.17)

Where  $i = 1 \dots k$   
 if  $l_1 = l$ ,  $l_{i+1} = nxt(l_i)$

$$(IFTRUE_{BS}) \frac{Env_v, Env_f \vdash \langle S, \sigma \rangle \rightarrow \sigma' \quad \sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} \mathbb{T}}{Env_v, Env_f \vdash \langle if(e) \{S\}, \sigma \rangle \rightarrow \sigma'} \quad (D.18)$$

$$(IFFALSE_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} \mathbb{F}}{Env_v, Env_f \vdash \langle if(e) \{S\}, \sigma \rangle \rightarrow \sigma} \quad (D.19)$$

$$(IFELSETRUE_{BS}) \frac{Env_v, Env_f \vdash \langle S_1, \sigma \rangle \rightarrow \sigma' \quad \sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} \mathbb{T}}{Env_v, Env_f \vdash \langle if(e) \{S_1\} Else \{S_2\}, \sigma \rangle \rightarrow \sigma'} \quad (D.20)$$

$$(IFELSEFALSE_{BS}) \frac{Env_v, Env_f \vdash \langle S_2, \sigma \rangle \rightarrow \sigma' \quad \sigma \circ Env_v \vdash e \rightarrow_{\mathbf{Exp}} \mathbb{F}}{Env_v, Env_f \vdash \langle if(e) \{S_1\} Else \{S_2\}, \sigma \rangle \rightarrow \sigma'} \quad (D.21)$$

$$(REPEATX-T_{BS}) \frac{\sigma \circ Env_v \vdash n \rightarrow_{\mathbf{Num}} v \quad Env_v, Env_f \vdash_l \langle S, \sigma \rangle \rightarrow \sigma' \quad Env_v, Env_f \vdash_l \langle repeat \ (v - 1) \ times \ \{S\}, \sigma' \rangle \rightarrow \sigma''}{Env_v, Env_f \vdash_l \langle repeat \ (n) \ times \ \{S\}, \sigma \rangle \rightarrow \sigma''} \quad (D.22)$$

Where  $n > 0$

$$(REPEATX-F_{BS}) \frac{\sigma \circ Env_v \vdash n \rightarrow_{\mathbf{Num}} 0}{Env_v, Env_f \vdash_l \langle repeat \ (n) \ times \ \{S\}, \sigma \rangle \rightarrow \sigma'} \quad (D.23)$$

Where  $n = 0$

$$(\text{REPEATWHILE-T}_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{Exp} \mathbb{T} \quad Env_v, Env_f \vdash_l \langle S; \text{repeat while } (e) \{ S \}, \sigma \rangle \rightarrow \sigma'}{Env_v, Env_f \vdash_l \langle \text{repeat while } (e) \{ S \}, \sigma \rangle \rightarrow \sigma'} \quad (\text{D.24})$$

$$(\text{REPEATWHILE} - \mathbb{F}_{BS}) \frac{\sigma \circ Env_v \vdash e \rightarrow_{Exp} \mathbb{F}}{Env_v, Env_f \vdash_l \langle \text{repeat while } (e) \{ S \}, \sigma \rangle \rightarrow \sigma'} \quad (\text{D.25})$$

$$(\text{REPEATWHILE-T}_{BS}) \frac{\sigma \circ Env_v \vdash x_2 \rightarrow L \quad Env_v, Env_f \vdash \langle S, \sigma[Env_v(x_1) \mapsto v_1] \rangle \rightarrow \sigma' \dots Env_v, Env_f \vdash \langle S, \sigma[Env_v(x_k) \mapsto v_k] \rangle \rightarrow \sigma''}{Env_v, Env_f \vdash_l \langle \text{repeat for each } (B \ x_1 \text{ in } x_2) \{ S \}, \sigma \rangle \rightarrow \sigma''} \quad (\text{D.26})$$



# Appendix E

## Testing

### E.1 Acceptance Testing

#### E.1.1 Code Example: Insertion-Sort

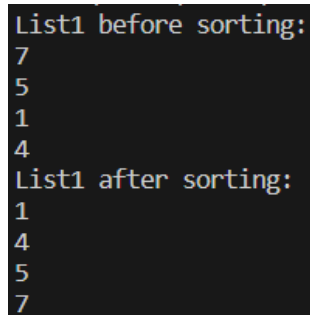
Source Code in PEAK+ :

```
1  comment: Function to sort a list of numbers;
2  function SortNumberList(list(number) listToSort) return list(
   number)
3  {
4      list(number) sortedList;
5      number bestIndex = 0;
6      number lowestVal = listToSort:ValueOf(0);
7      repeat for each(number i in listToSort){
8          repeat for each(number i in listToSort){
9              if(i < lowestVal){
10                 lowestVal = i;
11                 bestIndex = listToSort:IndexOf(i);
12             }
13         }
14         sortedList:Add(lowestVal);
15         lowestVal = listToSort:ValueOf(0);
16         listToSort:Replace(99999, bestIndex);
17         bestIndex = 0;
18     }
19     return sortedList;
20 }
21 comment: Function to print a list;
22 function PrintList(list(number) inputList) return nothing{
23     repeat for each(number elm in inputList){
24         call output(elm);
```



```
134         list = list->next;
135     }
136 }
137 return sortedList;
138 }
139 void PrintList(struct node *inputList)
140 {
141     {
142         struct node *list = inputList;
143         while (list != NULL)
144         {
145             int elm = *(int *)list->value;
146             printf("%d\n", elm);
147             list = list->next;
148         }
149     }
150 }
151 void main()
152 {
153     struct node *list1 = NULL;
154     AddToList(&list1, &(int){7}, sizeof(int));
155     AddToList(&list1, &(int){5}, sizeof(int));
156     AddToList(&list1, &(int){1}, sizeof(int));
157     AddToList(&list1, &(int){4}, sizeof(int));
158     printf("%s\n", "List1 before sorting:");
159     PrintList(list1);
160     printf("%s\n", "List1 after sorting:");
161     PrintList(SortNumberList(list1));
162 }
```

**result:**



```
List1 before sorting:
7
5
1
4
List1 after sorting:
1
4
5
7
```

**Figure E.1:** The result for the insertion-sort code example

**Code Example: Seconds Time-conversion****Source Code in PEAK+ :**

```

1  function mod(number a, number b) return number
2  {
3      number remainder = a;
4      repeat while (remainder >= b)
5      {
6          remainder = remainder - b;
7      }
8      return remainder;
9  }
10 number seconds_in_minute = 60;
11 number seconds_in_hour = seconds_in_minute * 60;
12 number seconds_in_day = seconds_in_hour * 24;
13 number seconds_in_week = seconds_in_day * 7;
14 number input_seconds = call number input();
15 decimal weeks = input_seconds / seconds_in_week;
16 input_seconds = call mod(input_seconds, seconds_in_week);
17 decimal days = input_seconds / seconds_in_day;
18 input_seconds = call mod(input_seconds, seconds_in_day);
19 decimal hours = input_seconds / seconds_in_hour;
20 input_seconds = call mod(input_seconds, seconds_in_hour);
21 decimal minutes = input_seconds / seconds_in_minute;
22 input_seconds = call mod(input_seconds, seconds_in_minute);
23 call output("Number of weeks:");
24 call output(weeks);
25 call output("Number of days:");
26 call output(days);
27 call output("Number of hours:");
28 call output(hours);
29 call output("Number of minutes:");
30 call output(minutes);

```

**Listing E.3:** Acceptance test input of the timeconversion code example**Target Code in C:****Listing E.4:** Acceptance test output of the timeconversion code example

```

106 ...
107 int mod(int a, int b)
108 {
109     int remainder = a;

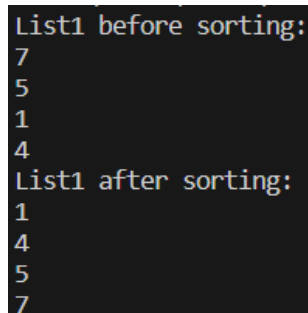
```

```

110     while ((remainder >= b))
111     {
112         remainder = (remainder - b);
113     }
114     return remainder;
115 }
116 void main()
117 {
118     int seconds_in_minute = 60;
119     int seconds_in_hour = (seconds_in_minute * 60);
120     int seconds_in_day = (seconds_in_hour * 24);
121     int seconds_in_week = (seconds_in_day * 7);
122     int input_seconds = *(int *)input("%d", sizeof(int));
123     float weeks = (input_seconds / seconds_in_week);
124     input_seconds = mod(input_seconds, seconds_in_week);
125     float days = (input_seconds / seconds_in_day);
126     input_seconds = mod(input_seconds, seconds_in_day);
127     float hours = (input_seconds / seconds_in_hour);
128     input_seconds = mod(input_seconds, seconds_in_hour);
129     float minutes = (input_seconds / seconds_in_minute);
130     input_seconds = mod(input_seconds, seconds_in_minute);
131     printf("%s\n", "Number of weeks:");
132     printf("%g\n", weeks);
133     printf("%s\n", "Number of days:");
134     printf("%g\n", days);
135     printf("%s\n", "Number of hours:");
136     printf("%g\n", hours);
137     printf("%s\n", "Number of minutes:");
138     printf("%g\n", minutes);
139 }

```

**result:**



```

List1 before sorting:
7
5
1
4
List1 after sorting:
1
4
5
7

```

**Figure E.2:** The result for the time-conversion code example

## E.1.2 Code Example: GDB Calculator

Source Code in PEAK+ :

```
1  number divisor;
2  number input_a;
3  number input_b;
4  number min;
5  number max;
6  number remainder;
7  text tryAgain = "y";
8  function mod(number a, number b) return number
9  {
10     number remainder = a;
11     repeat while (remainder >= b)
12     {
13         remainder = remainder - b;
14     }
15     return remainder;
16 }
17 function GetMax() return number
18 {
19     if (input_a > input_b)
20     {
21         return input_a;
22     }
23     else
24     {
25         return input_b;
26     }
27     return -1;
28 }
29 function GetMin() return number
30 {
31     if (input_a < input_b)
32     {
33         return input_a;
34     }
35     else
36     {
37         return input_b;
38     }
39     return -1;
40 }
```

```

41 repeat while (tryAgain is "y")
42 {
43     comment: Asks for two positive integers as input and in the
         case of an error, ask the user again;
44     call output("Enter two positive integers:");
45     boolean continue = false;
46     repeat while (continue is false)
47     {
48         input_a = call number input();
49         input_b = call number input();
50         if (input_a <= 0 or input_b <= 0)
51         {
52             call output("Please enter two positive integers:");
53             continue = false;
54         }
55         else
56         {
57             continue = true;
58         }
59     }
60     max = call GetMax();
61     min = call GetMin();
62     continue = false;
63     number i = min;
64     repeat while(i > 0 and continue is false)
65     {
66         if (call mod(max, i) is 0 and call mod(min, i) is 0)
67         {
68             divisor = i;
69             continue = true;
70         }
71         i = i - 1;
72     }
73     call output("The biggest divisor is:");
74     call output(divisor);
75     call output("Would you like to try again? (y/n)");
76     tryAgain = call text input();
77 }

```

Listing E.5: Acceptance test input of the GDB calculator code example

## Target Code in C:

Listing E.6: Acceptance test output of the timeconversion code example

```
106 ...
107 int mod(int a, int b)
108 {
109     int remainder = a;
110     while((remainder >= b))
111     {
112         remainder = (remainder - b);
113     }
114     return remainder;
115 }
116 int GetMax(int input_a, int input_b)
117 {
118     if((input_a > input_b))
119     {
120         return input_a;
121     }
122     else
123     {
124         return input_b;
125     }
126     return -1;
127 }
128 int GetMin(int input_a, int input_b)
129 {
130     if((input_a < input_b))
131     {
132         return input_a;
133     }
134     else
135     {
136         return input_b;
137     }
138     return -1;
139 }
140 void main(){
141     int divisor = 0;
142     int input_a = 0;
143     int input_b = 0;
144     int min = 0;
145     int max = 0;
146     int remainder = 0;
147     char * tryAgain = "y";
148     while(0 == strcmp(tryAgain, "y"))
149     {
```



```
150     printf("%s\n", "Enter two positive integers:");
151     int continue_ = 0;
152     while((continue_ == 0))
153     {
154         input_a = *(int *)input("%d", sizeof(int));
155         input_b = *(int *)input("%d", sizeof(int));
156         if(((input_a <= 0) || (input_b <= 0)))
157         {
158             printf("%s\n", "Please enter two positive integers:");
159             continue_ = 0;
160         }
161         else
162         {
163             continue_ = 1;
164         }
165     }
166     max = GetMax(input_a, input_b);
167     min = GetMin(input_a, input_b);
168     continue_ = 0;
169     int i = min;
170     while(((i > 0) && (continue_ == 0)))
171     {
172         if(((mod(max, i) == 0) && (mod(min, i) == 0)))
173         {
174             divisor = i;
175             continue_ = 1;
176         }
177         i = (i - 1);
178     }
179     printf("%s\n", "The biggest divisor is:");
180     printf("%d\n", divisor);
181     printf("%s\n", "Would you like to try again? (y/n)");
182     tryAgain = (char *)input("%s", sizeof(char *));
183     }
184 }
```

result:

```
Enter two positive integers:
5
-5
Please enter two positive integers:
20
5
The biggest divisor is:
5
Would you like to try again? (y/n)
y
Enter two positive integers:
```

Figure E.3: The result for the GDB-calculator code example

### E.1.3 Code Example: test input & repeat

Source Code in PEAK+ :

```
1  comment: Function to test repeat (x) times loop and input;
2  function StartFunction() return nothing{
3      text inputText;
4      boolean flag = true;
5      number repetitions;
6      repeat while(flag){
7          call output("");
8          call output("Write 'start' to start: ");
9          inputText = call text input();
10         if(inputText is "start"){
11             call output("");
12             call output("Write amount of repetitions: ");
13             repetitions = call number input();
14             repeat (repetitions) times{
15                 call output("This is a repeated message");
16             }
17         }
18     }
19 }
20 call StartFunction();
```

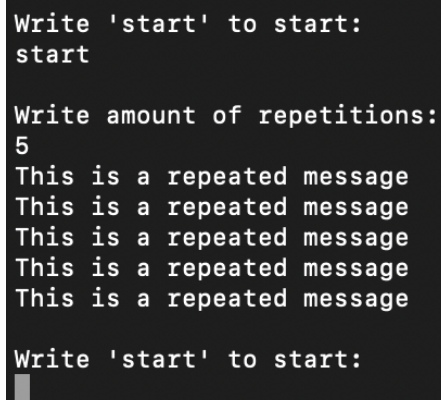
Listing E.7: Acceptance test input of the repeat loops code example

Target code in C:

**Listing E.8:** Acceptance test input of the repeat loops code example output in C

```
106 ...
107 void StartFunction()
108 {
109     char * inputText = "";
110     int flag = 1;
111     int repetitions = 0;
112     while(flag){
113         printf("%s\n", "");
114         printf("%s\n", "Write 'start' to start: ");
115         inputText = (char *)input("%s", sizeof(char *));
116         if(0 == strcmp(inputText, "start")){
117             printf("%s\n", "");
118             printf("%s\n", "Write amount of repetitions: ");
119             repetitions = *(int *)input("%d", sizeof(int));
120             for (int number = 0; number < repetitions; number
121                 ++){
122                 printf("%s\n", "This is a repeated message");
123             }
124         }
125     }
126 void main(){
127     StartFunction();
128 }
```

**result:**



```
Write 'start' to start:
start

Write amount of repetitions:
5
This is a repeated message
This is a repeated message
This is a repeated message
This is a repeated message
This is a repeated message

Write 'start' to start:
█
```

**Figure E.4:** The result for the repeat-input code example

### E.1.4 Code Example: Triangle area calculate.

Source Code in PEAK+ :

```
1 function triArea(number width, number height) return decimal
2 {
3     return (width * height) / 2;
4 }
5 call output(call triArea(2,3));
6 call output(call triArea(7,4));
7 call output(call triArea(10,10));
8 call output(call triArea(3,3));
```

Listing E.9: Acceptance test calculating triangle code examples

Target code in C:

Listing E.10: Acceptance test triangle code examples in C

```
1 float triArea(int width, int height)
2 {
3     return ((width * height) / 2);
4 }
5 void main(){
6     printf("%g\n", triArea(2, 3));
7     printf("%g\n", triArea(7, 4));
8     printf("%g\n", triArea(10, 10));
9     printf("%g\n", triArea(3, 3));
10 }
```

result:



```
3
14
50
4
```

Figure E.5: The result for the calculate triangle area code example

## E.1.5 Mathematical Expressions

### Source Code in PEAK+ :

```
1  number x = 14;
2  number y = 7;
3  number z = 10;
4
5  number A;
6  number B;
7  number C;
8  decimal D;
9  number E;
10
11 A = x + y;
12 B = x - y;
13 C = x * y;
14 D = x / y;
15 E = (x + y) * z;
16
17
18 call output(A);
19 call output(B);
20 call output(C);
21 call output(D);
22 call output(E);
```

Listing E.11: Acceptance test math expressions code examples

### Target code in C:

Listing E.12: Acceptance test math expressions code examples in C

```
1  void main(){
2      int x = 14;
3      int y = 7;
4      int z = 10;
5
6      int A = 0;
7      int B = 0;
8      int C = 0;
9      float D = 0;
10     int E = 0;
11
12     A = (x + y);
```

```

13     B = (x - y);
14     C = (x * y);
15     D = (x / y);
16     E = ((x + y) * z);
17
18     printf("%d\n", A);
19     printf("%d\n", B);
20     printf("%d\n", C);
21     printf("%g\n", D);
22     printf("%d\n", E);
23 }
```

**result:**

```

21
7
98
2
210
```

**Figure E.6:** The result for mathematical expressions code example

number x = 5; text y = "this is a test" text z = 5;

### E.1.6 Missing ";" error message

**Source Code in PEAk+ :**

```

1  number x = 5;
2  text y = "this is a test"
3  text z = 5;
```

**Listing E.13:** Acceptance test missing ";" code examples

**result:**

```

Syntax errors:
Error line 3: no viable alternative at input 'text' Caused by text.
```

**Figure E.7:** The result for missing semi colon code example

### E.1.7 Wrong type error message

Source Code in PEAK+ :

```
1 | text y = "a";  
2 | text z = 5;
```

Listing E.14: Acceptance wrong type code examples

result:

```
Type errors:  
Error line 2: Initialization of text 'z' does not match expression of type number
```

Figure E.8: The result for wrong type code example

### E.1.8 String Concatenation

Source Code in PEAK+ :

```
1 | text hej = "gem" + "fem";  
2 |  
3 | call output(hej);
```

Listing E.15: Acceptance test string conc code examples

Target code in C:

Listing E.16: Acceptance test string conc code examples in C

```
1 | char* concat(const char *str1, const char *str2)  
2 | {  
3 |     size_t len1 = strlen(str1);  
4 |     size_t len2 = strlen(str2);  
5 |     char *result = malloc(strlen(str1) + strlen(str2) + 1);  
6 |     strcpy(result, str1);  
7 |     strcat(result, str2);  
8 |     return result;  
9 | }  
10 |  
11 | void main()  
12 | {  
13 |     char * hej = concat("gem", "fem");
```

```

14 |     printf("%s\n", hej);
15 | }

```

**result:**



**Figure E.9:** The result for string concatenation code example

### E.1.9 Zero Division

**Source Code in PEAK+ :**

```

1 | number x = 5;
2 | number y = 0;
3 | decimal a;
4 | decimal b;
5 |
6 | a = x / y;
7 | b = y / x ;
8 |
9 |
10 | call output(a);
11 | call output(b);

```

**Listing E.17:** Acceptance test division by zero code examples

**Target code in C:**

**Listing E.18:** Acceptance test division by zero code examples in C

```

1 | void main()
2 | {
3 |     int x = 5;
4 |     int y = 0;
5 |     float A = 0;
6 |     float B = 0;
7 |
8 |     A = (divide(x, y));
9 |     B = (divide(y, x));
10 |

```



```
11     printf("%g\n", A);
12     printf("%g\n", B);
13 }
```

**result:**

**Error: Division by zero!**

**Figure E.10:** The result for division by zero code example

### E.1.10 Symbol Table scope-fix

**AddIdToFunctionBlock:**

```
232 public void AddIdToFunctionBlock(Symbol symbol, BlockNode
    blockNode)
233 {
234     //Only add the ID if the ID is contained in a functionBlock
235     //and has not already been declared within this
        functionBlock
236
237     FunctionBlockNode? fBlockNode = null;
238
239     var scope = _scopes[blockNode];
240
241     while (scope != null)
242     {
243         //Has functionBlock:
244         if (scope.Block is FunctionBlockNode)
245         {
246             fBlockNode = scope.Block as FunctionBlockNode;
247         }
248
249         //If the name is declared within the current scope, we
            don't add it
250         if (scope.Symbols.ContainsKey(symbol.Name))
251             break;
252
253         //If has functionBlock and has not been declared yet in
            any scopes,
254         //The ID must've been an identifier from outside the
            function
```

```

255         if (fBlockNode != null)
256         {
257
258             if (!fBlockNode.UsedVariables.Keys.Contains(symbol.
                Name))
259                 fBlockNode.UsedVariables.Add(symbol.Name,
                    symbol.Type);
260
261             //Exit because we've now met a functionBlockNode
262             break;
263         }
264
265         scope = scope.Parent;
266     }
267 }

```

**Listing E.19:** AddIdToFunctionBlock - 134-169 - CobraCompiler/SymbolTable.cs

### Visit Declaration-Node:

```

232 public override ASTNode? Visit(DeclarationNode node)
233 {
234     if (_reservedKeywords.Contains(node.Identifier.Name))
235         node.Identifier.Name = $"{node.Identifier.Name}_";
236
237     var sym = Lookup(node.Identifier.Name, _currentBlock);
238
239     string underscores = "";
240     while (sym != null)
241     {
242         underscores += "_";
243         sym = Lookup($"{node.Identifier.Name}{underscores}",
            _currentBlock);
244     }
245
246     node.Identifier.Name += underscores;
247     Insert(node.Identifier.Name, node.Identifier.TypeNode.Type,
        node);
248     Visit(node.Expression);
249     return null;
250 }

```

**Listing E.20:** Visit Declaration-Node - 232-250 - CobraCompiler/SymbolTable.cs

**Visit Identifier-Node:**

```
767 public void Visit(IdentifierNode node)
768 {
769     if (_reservedKeywords.Contains(node.Name))
770         node.Name = $"{node.Name}_";
771
772     var prevSym = Lookup(node.Name, _currentBlock);
773     var currSym = Lookup($"{node.Name}_", _currentBlock);
774
775     string underscores = "_";
776     while (currSym != null)
777     {
778         prevSym = currSym;
779         currSym = Lookup($"{node.Name}{underscores}",
780             _currentBlock);
781         underscores += "_";
782     }
783
784     node.Name = prevSym.Name;
785
786     if (prevSym != null)
787         AddIDToFunctionBlock(prevSym, _currentBlock);
788
789     if (prevSym == null)
790     {
791         SymbolError(node, $"{node.Name} is not found. Declare
792             your variable before use.");
793     }
794 }
```

**Listing E.21:** Visit Identifier-Node - 767-792 - CobraCompiler/SymbolTable.cs