

An Execution Environment for Algorithm Pseudocode



Colm Murphy

Final Year Project

Supervisor: Dr. Kieran Herley
Second Reader: Professor Michel Schellekens

Department of Computer Science
University College Cork

April 2025

Abstract

Introduction to Algorithms [2] by *Cormen, Leiserson, Rivest, and Stein* (CLRS) is one of the most widely used algorithms textbooks. Algorithms in this textbook and others are mainly presented in a pseudocode format. Executing pseudocode programs requires them to be translated to a high-level programming language first. This can act as a barrier for students to understand and reason about an algorithm. This project aims to develop a software system to facilitate the seamless execution of CLRS pseudocode and provide additional tooling expected of a high-level language.

The Pseudo-Code Compiler (PCC) is a transpiler that generates executable Python code from a pseudocode program. The software system provides additional tools to improve the experience of developing and executing pseudocode programs, including syntax highlighting, editor integration, and an interactive execution environment.

The resulting system allows users to write, run, and debug pseudocode programs without requiring prior knowledge of other programming languages, reducing the cognitive overhead associated with language translation and supporting a more intuitive, interactive learning process.

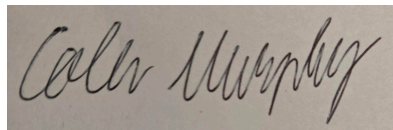
Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any education institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased, or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed:

A handwritten signature in dark ink on a light-colored rectangular background. The signature is written in a cursive style and appears to read "Calin Murphy".

Date: 2025/04/17

Contents

1	Introduction	1
1.1	Motivation	2
2	Background	3
2.1	What is Pseudocode?	3
2.2	Transpiler	4
2.3	Tree-Walk Interpreter	4
2.4	Abstract Syntax Tree	5
2.5	Formal Grammar	6
2.6	Lark	6
2.7	python-web-pdb	6
2.8	Unicodeitplus	6
2.9	Interegular	8
3	Analysis	9
3.1	Objectives	9
3.2	Language Analysis	10
3.3	Feasibility Analysis	10
3.4	System Requirements	11
3.4.1	User Stories	11
3.4.2	Functional Requirements	12
3.4.3	Non-Functional Requirements	13
4	Design	14
4.1	Architectural Diagrams	14
4.1.1	System Use-Case Diagram	14
4.1.2	Activity Diagram	15
4.2	Language Grammar	16
4.3	Output Language	16
4.4	Lexer	17

4.4.1	Whitespace Indentation	17
4.4.2	Handling of Non-ASCII characters	18
4.4.3	Naming Collisions With Python Reserved Identifiers	18
4.5	Parser	19
4.6	Approaches	20
4.6.1	Tree-Walk Interpreter	20
4.6.2	Transpilation	20
4.7	Transpiler	21
4.7.1	Generating Debug-Friendly Transpiled Code	21
4.8	REPL	22
4.9	Debugger	24
4.10	Interactive Execution Environment	25
4.11	Command-Line Interface	25
4.12	Standard Library	27
5	Implementation	29
5.1	Lexer	29
5.1.1	PostLex	30
5.1.2	Chaining multiple PostLex operations	30
5.1.3	Whitespace Indentation	31
5.1.4	Support for non-ASCII characters	31
5.1.5	Collisions With Python Keywords	33
5.2	Parser	34
5.2.1	Parsing Algorithm	35
5.2.2	Disambiguating Hyphens in CLRS-Style Pseudocode	37
5.2.3	REPL Parser	38
5.3	Transpiler	39
5.3.1	Bottom-Up Transpiler	40
5.3.2	Debug Transpiler	41
5.4	Debugger	42
5.4.1	Web-Based Debugger	43
5.5	Public API	44
5.6	IDE Support for Pseudocode Programming	45
5.7	Installation Script	47
6	Evaluation	48
6.1	Testing	48
6.1.1	Unit Testing	48
6.1.2	End-to-End Testing	49
6.2	Parsing Benchmark	50

6.3	User Evaluation	51
6.4	Appraisal	55
6.5	Limitations	56
6.5.1	Lack of Learning Resources	56
6.5.2	Limitations in LaTeX Symbol Translation	56
7	Conclusions	58
7.1	Reflection	58
7.2	How the Project was Conducted	59
7.3	Future Work	60
7.3.1	Support For Alternative Pseudocode Dialects	60
7.3.2	CLRS Pseudocode as a General-Purpose Program- ming Language	62
7.3.3	Collection of Performance Statistics and Runtime Com- plexity Estimation	62
7.3.4	Improvements to the Installation Process	63
7.3.5	Visualisation of Data Structures	64
8	Appendix	68
8.1	External Libraries Used	68
8.2	Pseudocode Compiler Grammar	68

Chapter 1

Introduction

Pseudocode is a widely used tool for teaching algorithms. It allows programmers and students to express the structure and logic of an algorithm in a clear format without becoming distracted by the syntactic style or implementation details of a specific programming language. However, pseudocode cannot normally be executed by a computer, which limits its practicality for experimentation, testing, and interactive learning.

This project sets out to bridge that gap by creating a software system capable of translating pseudocode programs into executable code in a high-level programming language. The primary goal was to enable users to seamlessly write and execute pseudocode algorithms without needing to translate them manually into another language, such as Python or Java. This makes it possible to experiment with pseudocode programs in a real programming environment while preserving their simple, educational style.

To achieve this, the project involves designing a pseudocode language matching the style used in the well-known algorithms textbook *Introduction to Algorithms* [2] by Cormen, Leiserson, Rivest, and Stein (CLRS). A compiler-like system is developed to automate the process of translating pseudocode programs into their equivalent Python code. The system includes additional features for syntax highlighting and interactive execution to enhance the user experience.

Overall, the system demonstrates that executable pseudocode environments are both practical and beneficial in educational contexts. This project lays the foundation for future enhancements such as performance analysis tools and visualisation features for data structures, further supporting learning and exploration in the study of algorithms.

1.1 Motivation

Algorithms are a fundamental part of computer science education and are typically introduced using pseudocode: a language-neutral notation designed to express algorithmic ideas without the syntactic overhead of real programming languages. However, pseudocode cannot be executed directly, and students are often required to manually translate it into a high-level language such as Python, Java, or C++ in order to test and explore its behaviour.

This translation process introduces several challenges: differences in language syntax, memory management, and naming conventions can obscure the underlying logic of an algorithm, making it harder for students to reason about its correctness and efficiency. The shift in context between an abstract pseudocode description and its concrete implementation often creates a disconnect that impedes learning.

This project is motivated by the idea of reducing the cognitive barrier and enabling students to execute and interact with pseudocode directly without needing to translate it manually. By providing a collection of tools that bridges the gap between pseudocode and executable programs, this system aims to support algorithm education in a more intuitive and accessible way.

Chapter 2

Background

2.1 What is Pseudocode?

Pseudocode is a way of writing algorithms using a structured, human-readable format that resembles programming languages but does not follow strict syntax rules. It serves as an intermediate step between natural language and actual code, making it easier to plan, communicate, and understand algorithms without worrying about language-specific syntax. Pseudocode makes use of common programming constructs, such as loops, conditionals, and function calls. Pseudocode is suitable for situations where clarity and readability are deemed more important than execution.

```
INSERTION-SORT( $A, n$ )
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3      // Insert  $A[i]$  into the sorted subarray  $A[1 : i - 1]$ .
4       $j = i - 1$ 
5      while  $j > 0$  and  $A[j] > key$ 
6           $A[j + 1] = A[j]$ 
7           $j = j - 1$ 
8       $A[j + 1] = key$ 
```

Figure 2.1: Pseudocode implementation of insertion sort. [2]

2.2 Transpiler

Transpilation is a term describing a compiler whose output code is the source code of some other high-level language. Rather than writing a compiler that translates code in the source language to architecture-specific machine code, a transpiler will produce a string of valid source code in some target language. This is also known as a *source-to-source compiler* or *transcompiler*.

A transpiled language has the benefit of being able to reuse existing tools of the target language, such as debuggers and runtimes for various platforms. TypeScript is a prime example of a transpiled language. The TypeScript compiler will parse its source code, perform semantic analysis and type checking before transpiling the input program to a string of valid JavaScript code. This technique allows developers to write and deploy TypeScript code without requiring a TypeScript interpreter on the end user's machine.

This project involved the development of a transpiler with pseudocode as its input language and Python as its target language. Transpiling pseudocode to Python brings the advantage of allowing pseudocode to execute on any platform with a Python runtime.

2.3 Tree-Walk Interpreter

A tree-walk interpreter is a technique used in developing interpreted programming languages in which code is executed immediately after parsing it to an Abstract Syntax Tree (AST)^{2.4}. To evaluate the code, the interpreter traverses a tree, one branch and leaf at a time, evaluating each node as it goes. This implementation style is not widely used for general-purpose languages since it tends to be slow.

2.4 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a structured, hierarchical representation of a program's syntax that abstracts away surface-level details such as formatting and punctuation. Each non-leaf node in the tree corresponds to a language construct, such as a statement, expression, or control structure, and stores some associated metadata. Leaf nodes represent atomic values like identifiers, literals, or operators. The AST forms the central intermediate representation shared across all stages of the compilation pipeline.

Figure 2.2 shows a simplified AST produced from parsing a polynomial expression in pseudocode.

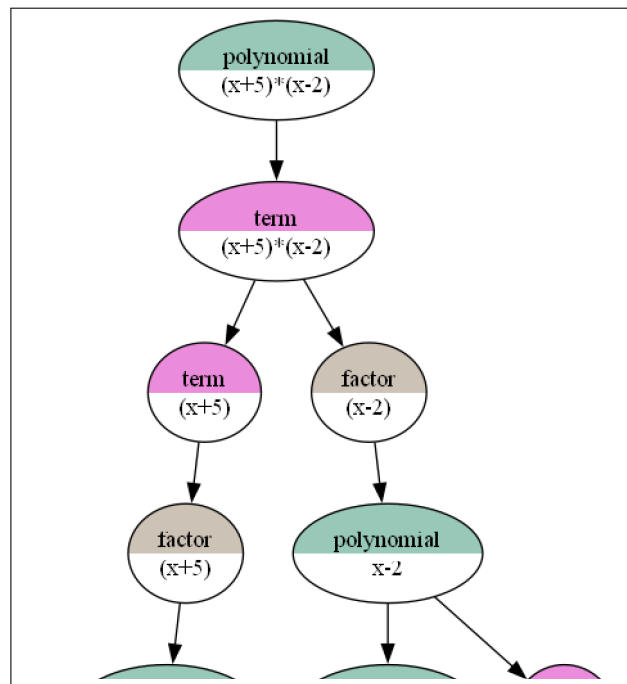


Figure 2.2: A partial AST produced from parsing a polynomial expression [7]

2.5 Formal Grammar

A **formal grammar** describes which strings from a formal language's *alphabet* are considered valid according to the language's syntax. A formal grammar can be defined as a set of production rules for strings in a formal language.

A formal grammar is a set of rules for rewriting strings along with a starting symbol, from which rewriting begins. A formal grammar can be thought of as a language generator.

If a grammar has the property that its production rules can be applied to any non-terminal symbol regardless of its context, the grammar is said to be a *context-free grammar*.

2.6 Lark

Lark[14] is a popular, general-purpose parsing library for Python. Lark can efficiently parse any context-free grammar 2.5. Lark accepts a formal grammar in Extended Backus-Naur Form defining the structure of a language, and produces a parser that can transform a string in the provided language to its AST 2.4 representation.

2.7 python-web-pdb

python-web-pdb[8] is a wrapper around Python's native debugger, `Pdb`[15]. *python-web-pdb* provides a web-based graphical interface to `Pdb`. This library was forked and adapted for use with the pseudocode debugger developed for this project. Figure 2.3 shows `python-web-pdb` being used to debug a simple program.

2.8 Unicodeitplus

Unicodeitplus [3] is a Python package used to convert simple \LaTeX expressions to their Unicode approximation. Table 2.4 shows a handful of examples of how the library may be used.

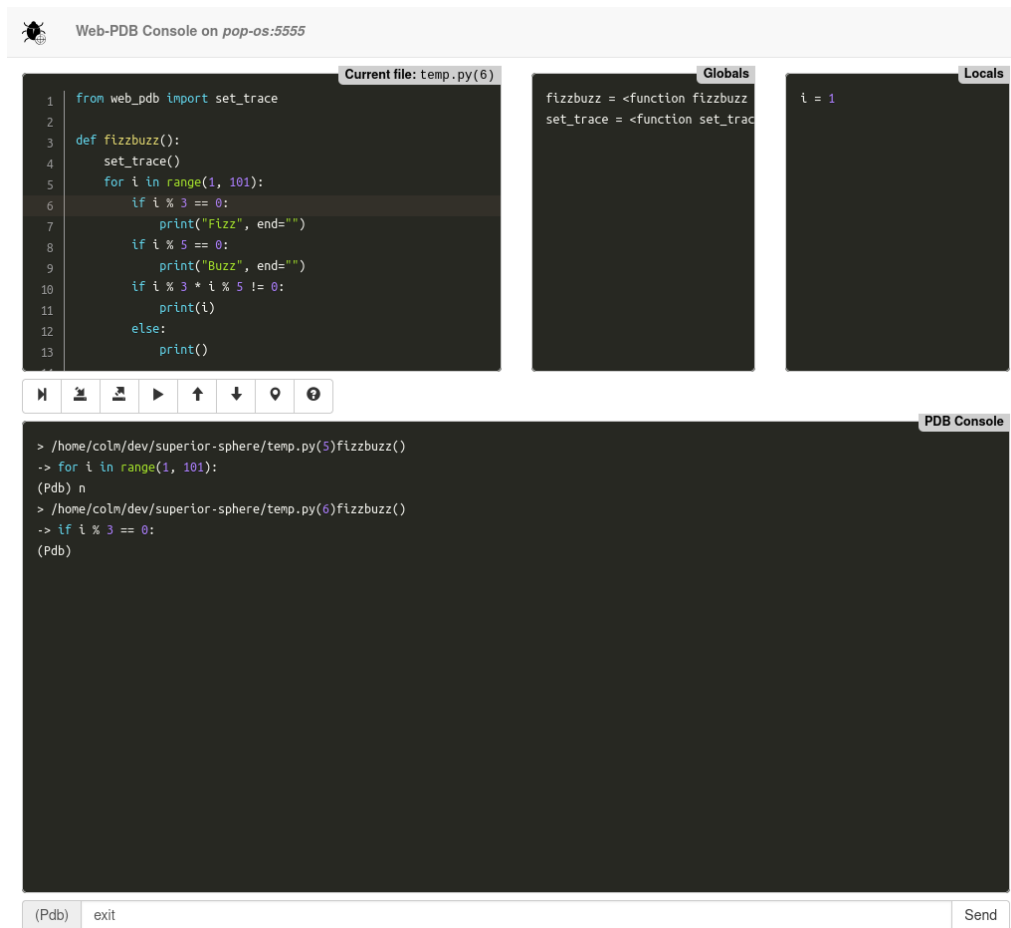


Figure 2.3: Python-web-pdb interface

Input	Unicodeitplus formatted output
π	π
$\alpha\beta - \text{prune}$	$\alpha\beta - \text{prune}$
K^0_S	K^0_S

Figure 2.4: An example of how unicodeitplus transforms \LaTeX to its Unicode representation

```
NAME: /[a-zA-Z0-9]+/
NUMBER: /[1-9][0-9]*/
```

Figure 2.5: Example of two colliding regular expressions.

2.9 Interegular

Interegular is a Python package for detecting collisions in regular expressions. This package was used to identify potential inconsistencies in the pseudocode language’s grammar and pre-emptively address defects in the parsing logic.

A regular expression collision occurs when two different regular expressions can match the same input string in a way that causes ambiguity or unintended matching behaviour. This is particularly important in lexical analysis, where regex patterns are assigned to token types and matched against the input.

Figure 2.5 shows an example of a collision where certain numeric strings (e.g. 123) may be matched as either a **NAME** or a **NUMBER** token. Collisions can be addressed by modifying one or both regular expressions or by assigning a priority to the tokens. In the example shown, the conflict can be avoided by forbidding a **NAME** token to begin with a digit.

Chapter 3

Analysis

3.1 Objectives

This project aims to facilitate the seamless execution of pseudocode algorithms by creating a compiler to translate human-readable pseudocode to a machine-readable representation. This should allow for a pseudocode algorithm to be executed without the mental context shift to another programming language. An end user of the system should not be required to have prior knowledge of Python or other programming languages. The pseudocode language implementation should hide as many implementation details as possible, such as manual memory management and type checking. Additionally, the outputted Python code should be understandable and easily modifiable by the programmer if desired.

3.2 Language Analysis

This step involved a comprehensive review of *Introduction to Algorithms* [2], identifying the syntactic structure and behaviour of the pseudocode language used. This included language keywords, control flow constructs, naming patterns, built-in functions, and more. This review helped to discover the language features present and define a formal language specification upon which to build a compiler.

The review of the text highlighted numerous instances of ambiguous procedures in pseudocode algorithms. One example of ambiguous logic is in the procedure `INTERVAL-SEARCH(T i)` [2, Section 14, 4]. The statement `x \neq T.nil and i does not overlap x.int` is ambiguous and cannot be faithfully executed without further context. Given the presence of this ambiguity, it is not possible to create a system to accurately translate all algorithms defined in the text to a machine-executable representation

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x  $\neq$  T.nil and i does not overlap x.int
3      if x.left  $\neq$  T.nil and x.left.max  $\geq$  i.low
4          x = x.left // overlap in left subtree or no overlap in right subtree
5      else x = x.right // no overlap in left subtree
6  return x
```

Figure 3.1: Example of ambiguous logic in pseudocode.

3.3 Feasibility Analysis

CLRS pseudocode does not have a strict definition. A detailed, comprehensive survey of the features of the pseudocode notation should be undertaken to identify a subset of the language that can be faithfully executed without requiring modifications or sacrificing the program's correctness. This should then be used to define attainable expectations for the project.

Most C-like languages heavily restrict the set of valid strings to denote a variable or function identifier to Latin letters, digits and underscores. The grammar of CLRS pseudocode is much more permissive to the point of introducing ambiguity to the language. For example, the strings

"alpha-beta-prune" and " $\alpha - \beta - \text{prune}$ " are considered valid identifiers for a function or variable, or may also be interpreted as an arithmetic expression with three operands. Such limitations must be considered when implementing the compiler 5.2.2.

3.4 System Requirements

3.4.1 User Stories

User stories are an essential part of requirements analysis because they help define what the user needs and why in a clear, concise, and structured manner. For this project, a set of user stories was created to determine the features and capabilities a user would want from such a system.

1. "As a student learning algorithms, I want to write pseudocode in a structured manner and execute it without manually converting it to a high-level language."
2. "As a reader of *Introduction to Algorithms*[2], I want to execute a pseudocode algorithm described in the textbook without manually translating it to another language."
3. "As a programmer, I want to write and execute algorithms using a variety of non-Latin characters in variable and function identifiers."
4. "As a programmer, I want to insert non-Latin characters into my program with a standard English keyboard."
5. "As a student learning algorithms, I want to write pseudocode in a structured format and have it executed as Python code, so that I can test and verify my algorithm without manually converting it."
6. "As a pseudocode programmer, I want to receive clear error messages when my pseudocode has syntax errors, so that I can quickly correct my mistakes and learn from them."
7. "As a programmer debugging complex logic, I want to set breakpoints at specific lines in my pseudocode, so that execution pauses where I need to inspect values."
8. "As a user debugging my pseudocode, I want to see the current values of all variables at any point, so that I can understand how my algorithm is manipulating data."

9. “As a new user installing the software, I want to have a simple and guided installation process, so that I can set up the system quickly without technical difficulties.”

Based on the language analysis 3.2, feasibility analysis 3.3, and user stories 3.4.1 described above, a set of functional and non-functional requirements was defined for the project.

3.4.2 Functional Requirements

1. Pseudocode Execution

- (a) The system must allow users to input pseudocode in a predefined format.
- (b) The system must translate valid pseudocode into its equivalent in a high-level programming language.
- (c) The system must detect and report syntax errors in the pseudocode.

2. Support for non-ASCII Characters

- (a) The system must allow for the usage of non-ASCII characters in variable and function identifiers.
- (b) The system must allow users to input special characters with a standard English keyboard, using a \LaTeX -like syntax (e.g. $\backslash\alpha \rightarrow \alpha$).

3. GUI Debugger

- (a) The system must provide a Graphical User Interface for debugging pseudocode.
- (b) The debugger must allow step-by-step execution of pseudocode.
- (c) Users must be able to set breakpoints in their pseudocode program.
- (d) The system must display variable values at each step of execution.
- (e) The system must allow users to pause, reset and resume execution.

4. Installation Script

- (a) The system must provide an installer for Windows, macOS and Linux.
- (b) The installer must install and verify all required dependencies.
- (c) The installer must prompt the user to install optional dependencies.

3.4.3 Non-Functional Requirements

1. Performance and Efficiency

- (a) The time needed to translate a pseudocode program should correlate linearly with the number of characters in the program.
- (b) The debugger must update variable values and step execution within 200 milliseconds of a user action
- (c) The debugger process must not consume more than 200 MB of RAM when debugging a program.

2. Usability

- (a) The system must provide a syntax highlighting engine for Visual Studio Code.
- (b) Error messages must be clear and informative to the user.

3. Reliability and Availability

- (a) The debugger must never cause execution to hang indefinitely.
- (b) The system must not crash when handling long pseudocode programs (10,000+ lines).

4. Compatibility and Portability

- (a) The system must be platform-independent. The system should be compatible with all major desktop operating systems.
- (b) The Python implementation must be compatible with Python 3.10 and later
- (c) The system must support Windows 10+, macOS 14+ (Sonoma) and Ubuntu 22.04+

Chapter 4

Design

4.1 Architectural Diagrams

4.1.1 System Use-Case Diagram

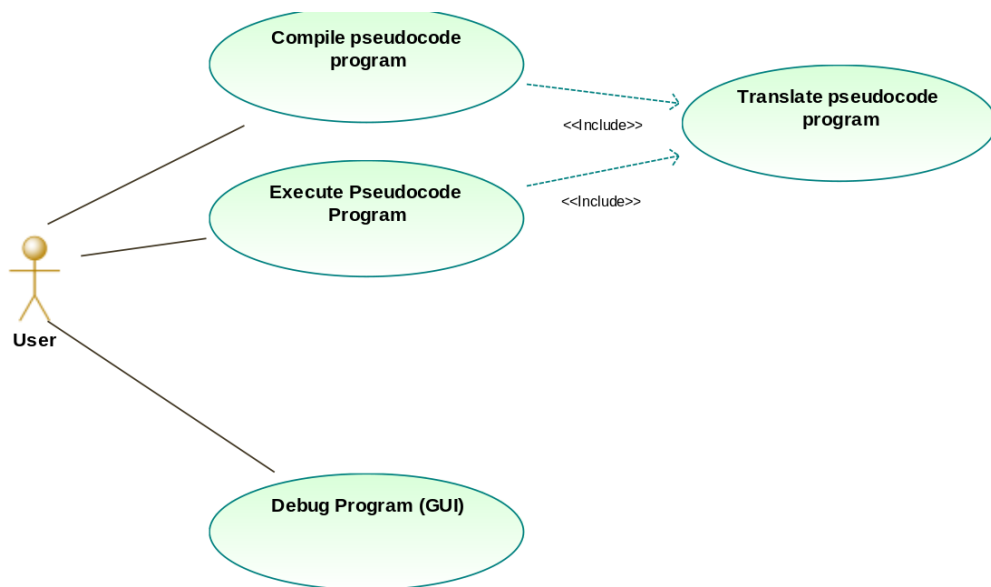


Figure 4.1: System use-case diagram.

4.1.2 Activity Diagram

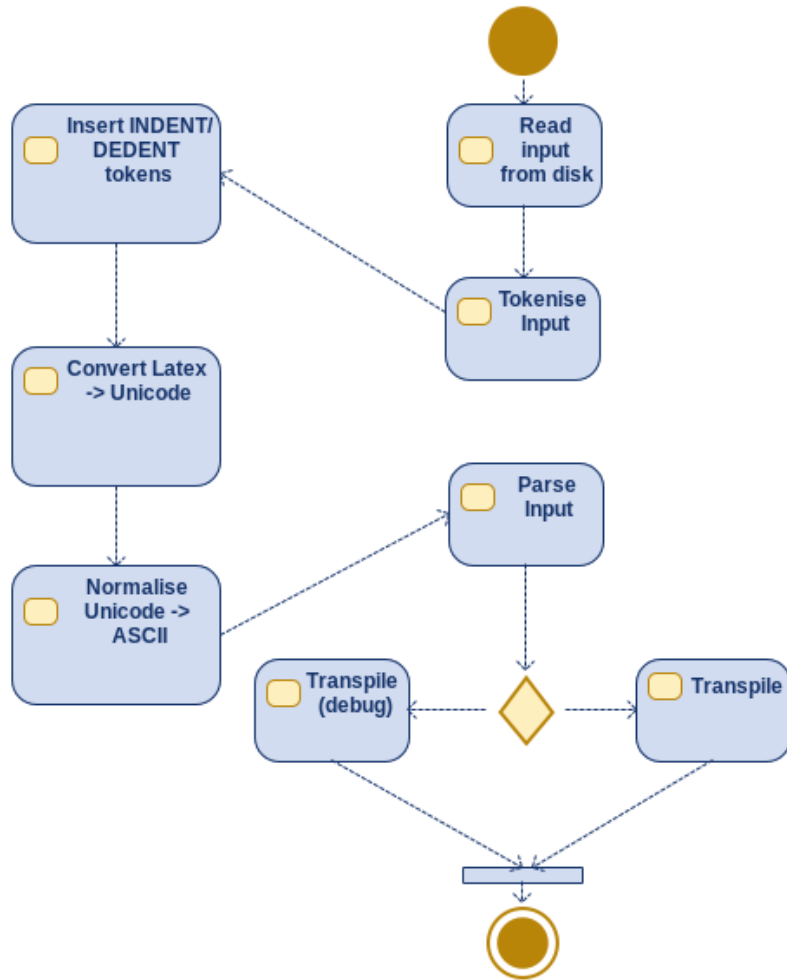


Figure 4.2: High-level transpiler activity diagram.

4.2 Language Grammar

The syntactic structure of the pseudocode language is defined using a formal grammar 2.5 specification 8.1. This grammar distinguishes between two types of elements: *terminals* and *non-terminals*.

Terminals represent the atomic units of syntax, such as keywords, operators, identifiers, and literals, which are recognised by the lexer during the lexical analysis phase. These are the basic building blocks from which the language is constructed.

Non-terminals, on the other hand, define combinations of terminals and other non-terminals that form valid syntactic constructs in the language, such as expressions, control structures, and function declarations. These rules are used by the parser to build a structured representation of the program, typically in the form of an Abstract Syntax Tree 2.4.

Together, the grammar’s terminal and non-terminal definitions provide a complete specification of the language’s valid syntax and serve as the foundation for both the lexer and parser components of the system.

4.3 Output Language

In designing the translation system, a central decision involved selecting the target programming language to which pseudocode would be translated. Two primary candidates were evaluated: Kotlin and Python.

Kotlin [4] is a modern, statically typed programming language. It can be compiled to bytecode targeting the Java Virtual Machine (JVM) or to native binaries on select platforms. Kotlin also benefits from an ecosystem of tools for programmatic source-code generation, most notably, libraries such as KotlinPoet¹ that simplifies the construction of structured source code. Additionally, the JVM and Java Runtime Environment (JRE) are widely available across platforms, offering a platform-agnostic execution environment, fulfilling the system’s requirements. However, Kotlin’s static type system introduces challenges when translating from dynamically typed pseudocode. This mismatch introduces additional challenges to the translation system. Kotlin has some limited metaprogramming capabilities, namely the ability to generate and execute Kotlin script files at runtime.

In contrast, Python is a high-level, dynamically typed, interpreted lan-

¹KotlinPoet is a Kotlin API for generating .kt source files programmatically. See: <https://github.com/square/kotlinpoet>

guage that emphasises readability and minimal syntactic overhead. Its syntax closely resembles common pseudocode conventions, simplifying the translation process and making the generated code more accessible to students and beginner programmers. Python’s dynamic typing eliminates the complications in translating pseudocode to a statically-typed programming language. Python supports robust metaprogramming features such as runtime code evaluation and introspection, which provide flexibility in constructing and executing translated programs. Python’s metaprogramming capabilities are particularly useful for the development of a pseudocode debugger system. Additionally, Python was chosen as the *implementation* language of the translation system. Thus, using Python as the system’s output language would simplify the development process and minimise the system’s dependency requirements.

Given these considerations, Python was selected as the output language for the system. Its dynamic typing, high readability, and minimal setup requirements make it well-suited for translating pseudocode algorithms into executable code with minimal friction.

4.4 Lexer

The pseudocode lexer accepts the input string of source code and produces a sequence of token objects as output. Tokens are the atomic units of syntax, each representing a minimal, meaningful component of the language, such as keywords, identifiers, literals, or operators. These tokens form the basic building blocks for the parser, which uses them to construct higher-level syntactic structures in the form of a syntax tree.

4.4.1 Whitespace Indentation

A block statement is a syntactic construct that groups any number of assignments, declarations or other statements into a single statement. Traditionally, C-like syntaxes use curly braces to define a block statement. Pseudocode, as defined in *Introduction to Algorithms*[2] uses leading whitespace to denote that a line of code belongs to a block statement. This convention mirrors other high-level languages such as Python and Scala.

Handling of whitespace indentation presents the problem where some occurrences of whitespace should be ignored by the compiler, while other occurrences add semantic meaning to the program and must be handled appropriately by the parser. As a solution, the compiler iterates over the

token stream produced by the lexer and analyses the use of whitespace in the program. Where appropriate, `_INDENT` and `_DEDENT` tokens are inserted into the stream to mark the beginning and end of a block statement. All other non-semantic whitespace is discarded.

4.4.2 Handling of Non-ASCII characters

While the source language supports a subset of the Unicode alphabet to be used in variable and function identifiers in the form of \LaTeX expressions, the target language (Python) does not allow the full Unicode character set to be used in its syntax. Per the system requirements 1, a user should be able to insert Unicode characters into a program with a standard English keyboard. Before transpilation, \LaTeX elements in identifiers will be transformed to their Unicode approximation using `unicodeitplus` 2.8. The system will produce a modified version of the pseudocode source with all instances of \LaTeX fragments substituted with their Unicode approximation. Before translating the input pseudocode to code in the target language, all Unicode characters will be replaced with their ASCII. This step is crucial to avoid producing a syntactically incorrect program in the target language.

4.4.3 Naming Collisions With Python Reserved Identifiers

Python 3.14 reserves a set of 35 unique identifiers 4.3 for use as keywords within the language. As a consequence, a pseudocode program that uses one of Python's reserved words as an identifier will not yield a faithful translation if the identifier is not modified in the outputted Python code. Per the system's requirements, the user must not be exposed to implementation details of the compiler's target language. Thus, it is not an appropriate design choice to forbid the use of Python keywords for use as pseudocode identifiers. To address this problem without exposing implementation details to the user, the compiler will internally modify any identifier tokens whose values match a Python keyword. This process is invisible to the user. Figure 4.4.3 shows a pseudocode snippet that declares a variable `class` whose name is a reserved word in Python.


```
class = "maths"  
grade = 76
```

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Figure 4.3: Python 3.14 reserved words [16]

4.5 Parser

The parser consumes the sequence of tokens produced by the lexer and constructs a structured, tree-based representation of the program using the language’s grammar. This representation is known as the *Abstract Syntax Tree* (AST) 2.4.

In the AST, each non-leaf node corresponds to a production rule from the grammar, representing higher-level syntactic constructs such as expressions, statements, or function definitions. Leaf nodes, by contrast, correspond to terminal symbols, such as identifiers, numbers, or string literals defined in the grammar.

For this project, the pseudocode parser is generated by the Lark parsing toolkit [14]. The use of a parser generator has many benefits over a bespoke, handwritten parser. Namely, the ability to quickly adapt to changes and additions to the language’s grammar while minimising the likelihood of introducing bugs to the parser’s internal logic. Although parsers produced by code generators such as Lark are generally slower than their handwritten counterparts, they maintain the same worst-case runtime complexity. Additionally, pseudocode programs tend to be relatively small. For this reason, the loss in performance was deemed acceptable.

Each AST node will also carry additional metadata, including the name of the production rule it represents, along with the exact line and column numbers from the source text where the rule was applied. This metadata allows the reconstruction of the original pseudocode segment associated

with a given node, which is useful for debugging and error reporting.

Traditional compilers will apply some optimisations to the tree representation of the user’s program. Examples of such optimisations are dead code elimination and compile-time evaluation or simplification of arithmetic expressions. One of this project’s key objectives is to produce an executable program that matches the provided pseudocode as closely as possible. Another requirement is that a user should be able to modify the outputted program in the compiler’s target language if desired. For this reason, it was decided not to apply optimisations that may alter the user’s provided pseudocode.

4.6 Approaches

Two implementation strategies were considered for executing pseudocode programs: a tree-walk interpreter 2.3 and a transpiler 2.2 to a high-level language. Each approach was evaluated with respect to the system requirements outlined in Section 3.4.2.

4.6.1 Tree-Walk Interpreter

The first approach involves interpreting the program directly by traversing its Abstract Syntax Tree (AST) at runtime. This method is relatively simple to implement and removes the dependency on any external output language and its compilation infrastructure. However, tree-walk interpreters tend to be inefficient for larger programs, as the AST must be evaluated repeatedly on each execution. Additionally, this approach would require implementing custom debugging and execution tools specifically for the pseudocode language, increasing development complexity.

4.6.2 Transpilation

The second approach is to transpile pseudocode to executable code in a high-level target language. While this introduces a dependency on the selected output language and its toolchain, it enables the system to reuse existing compilers, debuggers, and runtime environments. This significantly reduces the need to implement low-level execution and debugging features from scratch. Moreover, transpiled programs typically benefit from the performance and maturity of the target language’s ecosystem.

Given the trade-offs outlined, the transpilation approach was chosen as the basis for the system implementation due to its extensibility, performance advantages, and alignment with the project’s requirements and objectives.

4.7 Transpiler

The transpiler is responsible for converting the program’s AST representation into executable source code in the target language. This process involves traversing the AST and generating corresponding code fragments based on the structure and content of each node.

The transpiler will define a mapping from parse tree nodes to strings of code in the output language. The mapping for a given tree node will be determined by the name of the production rule it represents, stored in the node’s metadata. If the transpiler encounters a node for which no mapping exists, the error will be reported, and a default mapping will be applied to the node. The transpiler may traverse the AST either from the bottom-up 5.3.1 or top-down 5.3.2.

A bottom-up transpiler will begin by mapping leaf nodes to their string representations. The resulting string(s) will be provided as arguments to the translation functions of their respective parent nodes. One drawback to this method is that the metadata associated with a node will be stripped as it is transpiled. Alternatively, a top-down transpiler will begin its traversal at the root node and explicitly call the translation functions of its children as required. This approach preserves the metadata of nodes in the parse tree.

4.7.1 Generating Debug-Friendly Transpiled Code

For the purpose of debugging pseudocode, the compiler must produce an output with additional data, relating elements of the output code to fragments of the pseudocode input. When the system is invoked with the debug flag set, the output will contain metadata symbols along with the Python translation of the input program.

Each AST node contains additional metadata, including the name of the production rule it represents, as well as the exact line and column numbers from the source text where the rule was applied. To produce a transpiled output containing metadata for use by the debugger, each Python statement in the output program contains a comment denoting the line number in

```
1 | COUNT()
2 |     for i = 1 to 10
3 |         print i
4 |
5 | COUNT()
6 |

1 | def COUNT(): # 1:1
2 |     for i in range(1, 10 + 1): # 1:2
3 |         print(i) # 1:3
4 | COUNT() # 1:5
5 |
```

Figure 4.4: Example showing how lines in the output program are related to lines of the source program by inserting metadata-containing comments to the output program.

the source program from which the outputted line originates. Figure 4.4 highlights an example of metadata used by the debugger.

4.8 REPL

To support interactive program development, the system includes a Read-Evaluate-Print Loop (REPL) for executing pseudocode statements incrementally. The REPL allows users to input single lines of pseudocode, which are immediately parsed, transpiled, and executed within a persistent environment. Each iteration of the REPL performs four key steps:

1. **Read:** The system reads a line or block of input from the user.
2. **Parse:** The input is passed through the lexer and parser to generate an AST representation.
3. **Transpile and Execute:** The AST is transpiled into target language code, which is then executed in the context of a maintained runtime environment.
4. **Print:** Any return values or output generated by the executed code are displayed to the user.

The REPL maintains a shared execution context across inputs, allowing variables, functions, and control flow to persist between entries. This design enables users to incrementally build and test algorithms without restarting the interpreter or rewriting prior code.

To enhance usability, the REPL also provides basic error reporting. Syntax errors encountered during parsing are caught and reported. Runtime errors in the executed code are similarly surfaced, allowing users to debug and correct issues interactively. This interactive environment is especially well-suited for experimentation, instructional use, and iterative development workflows.

4.9 Debugger

The system includes an integrated debugger to support introspection and step-by-step execution of pseudocode programs. This custom debugger provides users with a familiar interface for tracing program execution, inspecting variables, and diagnosing logic errors within the translated pseudocode.

The debugger is tightly integrated with the execution environment of the translated code. When invoked, the debugger hooks into the program's runtime and allows users to pause execution at arbitrary points via breakpoints. Once paused, users can step through individual lines of pseudocode, examine the state of variables, and resume execution as needed.

The debugger will offer a graphical user interface to clearly display the program's current execution state by highlighting the current code location and values of defined variables.

Overall, the debugger enhances the usability of the system for students, educators, and developers by offering familiar and powerful debugging tools tailored to the pseudocode abstraction.

Command	Alias	Function
step	z	Execute the current line. Calls to functions and routines defined outside of the file scope will be skipped. All other calls will be stepped into.
continue	c	Continue execution until a breakpoint is reached or the program terminates.
break	b	Set a breakpoint at the selected line.
clear	cl	Clear all breakpoints at the given line.
globals		Display all global variables as a collection of key-value pairs.
locals		Display all local variables as a collection of key-value pairs.

4.10 Interactive Execution Environment

In addition to the command-line debugger, the system provides a Graphical User Interface (GUI) to the debugger. The graphical interface will feature the same commands and functionality as its CLI counterpart while aiming to provide a more accessible and improved user experience.

The interface includes a source code viewer that displays the pseudocode source with syntax highlighting. As execution progresses, the currently executing line is visually highlighted, allowing users to follow the program's control flow in real time. Breakpoints can be set by clicking in the margin beside a line of code, and a call stack view shows the current function context.

A live variable inspector displays the values of all local and global variables in scope, updating reactively after each action. This allows users to observe how the program state changes over time. An integrated output console shows print statements, error messages, and other runtime feedback, ensuring all relevant information is centralised within the interface.

The interactive execution environment is well-suited for learners, as it bridges the gap between abstract pseudocode and concrete program behaviour.

4.11 Command-Line Interface

The system provides a Command-Line Interface (CLI) to facilitate the translation of pseudocode files. The CLI serves as the primary entry point to the transpilation tool, allowing users to specify input files, choose output formats, and control the behaviour of the transpiler using structured command-line arguments.

The CLI is designed to be intuitive and consistent with common command-line interface conventions. It follows established patterns for argument parsing, flag naming, and help message formatting, making it accessible to users familiar with typical developer tools. This approach reduces the learning curve and ensures the interface behaves as users expect, whether invoked directly from the terminal or integrated into shell scripts and development workflows. A full list of supported command-line arguments is shown in figure 4.5

Usage information and help text are automatically generated and dis-

Argument name	Alias	Function
<code>--help</code>	<code>-h</code>	Display a message to the terminal outlining the supported CLI options and usages.
<code>--version</code>	<code>-v</code>	Show the current version of the transpiler
<code>--debug</code>	<code>-d</code>	Indicates that the transpiled output should have debug symbols attached.
<code>--output</code>	<code>-o</code>	Override the filesystem path to which the translated output will be saved.
<code>--output-rendered-source</code>	<code>-r</code>	Boolean-valued flag indicating whether a formatted pseudocode file should be produced with \LaTeX fragments rendered to their Unicode equivalents. True by default

Figure 4.5: Supported command-line arguments.

played when invalid arguments are passed or the `--help` flag is provided, improving accessibility for new users.

Overall, the CLI provides a lightweight yet powerful interface for manual or automated use cases.

4.12 Standard Library

To support common patterns and reduce boilerplate, the system includes a standard library: a collection of predefined functions and data structures available to all pseudocode programs. This standard library provides familiar constructs for manipulating data while maintaining the simplicity and readability of pseudocode.

The standard library is implemented as a predefined set of functions and classes implemented in the transpiler's output language. The standard library will, by default, be imported into any program produced by the transpiler. The standard library will be packaged alongside the transpiler system itself. This ensures that all generated code remains self-contained and portable.

By design, functions defined in the output language's standard library will also be callable from a pseudocode program.

The standard library includes:

- **Basic data structures:** A complete list of available data structures is outlined in 4.12.
- **Mathematical utilities:** Functions for computing minimums, maximums and other numeric operations that are frequently used in algorithmic pseudocode.
- **String utilities:** Functions for concatenation, slicing, searching, and formatting text.
- **Random number generators:** Utility functions for the generation of pseudorandom numbers.

Below is a list of all data structures provided in the system's standard library.

- Array
- Binary Tree
- Graph
- Heap
- Linked List

- Doubly-Linked List
- Queue
- Stack

All standard library functions are designed to match the semantics typically implied by textbook pseudocode, prioritising readability over language-specific optimisations. This consistency allows users to focus on expressing algorithmic logic clearly, without being burdened by low-level implementation details.

The standard library is structured modularly, allowing new functions or structures to be added incrementally as needed.

Chapter 5

Implementation

In the previous chapter, the architecture of the pseudocode compiler was outlined. In this chapter, the details on how this was implemented will be discussed.

5.1 Lexer

The system's lexer is responsible for transforming the pseudocode source string into a stream of token objects that represent the atomic syntactic units of the language. The lexer is automatically generated using the Lark parsing toolkit [14], based on a formally defined grammar 8.1.

Each token corresponds to a terminal symbol defined in the pseudocode language grammar (see Listing 8.1). These include keywords, operators, literals, and punctuation. The lexer also preserves source metadata such as line and column numbers, which are attached to each token and propagated through later compilation stages to support features like error reporting and debugging.

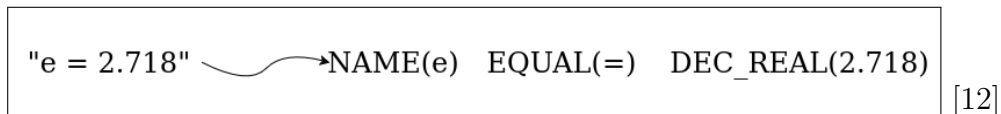


Figure 5.1: Pseudocode tokenisation example.

5.1.1 PostLex

Immediately after the input pseudocode string is tokenised, a series of transformations are applied to the resulting token stream. These transformations are encapsulated in classes that extend from the abstract class `PostLex`. Each `PostLex` subclass implements the `process` method, which defines how the input token stream is filtered, modified, or annotated.

This post-processing step allows the lexer to remain simple and declarative while handling more complex or contextual adjustments separately. For example, transformations can be used to:

- Insert or remove tokens based on surrounding context (e.g., remove comment tokens).
- Replace or modify tokens (e.g., replace whitespace indentation with explicit `_INDENT` and `_DEDENT` tokens).
- Annotate tokens with additional metadata required by later stages in the translation pipeline.

The definition of the `PostLex` abstract class is shown below:

```
class PostLex(ABC):
    @abstractmethod
    def process(self, stream: Iterator[Token]) -> Iterator[
Token]:
        pass
```

5.1.2 Chaining multiple PostLex operations

In order to support multiple `PostLex` operations, a wrapper class is used to perform multiple transformations on the token stream sequentially.

```
class PostLexPipeline(PostLex):
    def __init__(self, postlexers: list[PostLex]):
        super().__init__()
        self.postlexers: list[PostLex] = postlexers

    def process(self, stream: Iterator[Token]) -> Iterator[
Token]:
        output: Iterator[Token] = stream
        for postlexer in self.postlexers:
            output = postlexer.process(output)
```

```
return output
```

5.1.3 Whitespace Indentation

The pseudocode language uses significant whitespace to denote block structure 4.4.1, similar to Python. Identifying appropriate positions in the token stream for `_INDENT` and `_DEDENT` tokens requires additional context regarding leading whitespace on the current and surrounding lines that cannot be implemented with a traditional Lexer. To address this, the lexer uses a `PostLex` transformation that analyses line-level indentation and replaces whitespace tokens with `_INDENT` and `_DEDENT` tokens where appropriate.

This transformation scans the token stream line by line, tracks changes in indentation level, and indentation tokens to reflect increases or decreases in indentation depth. These synthetic tokens are not part of the original input but are essential for constructing the correct hierarchical structure in the parser.

By separating indentation logic from the core grammar and lexer, the system simplifies grammar design and improves maintainability. The use of `PostLex` for this purpose makes the lexer pipeline both extensible and language-aware.

```
MEANING-OF-LIFE()####_#_<-non-significant_whitespace
#####return_42
#^^^significant_whitespace
```

5.1.4 Support for non-ASCII characters

As discussed in 4.4.2, the system allows for non-ASCII characters to be used in variable and function identifiers by enclosing a `LATEX` expression between two `$` characters (e.g. 2.4). Immediately after tokenising the input, and before parsing, a `PostLex 5.1.1` object will iterate over all tokens produced by the lexer, and extract any fragments of `LATEX`-like syntax.

```
from unicodetplus import replace

class UnicodeFragment:
    def __init__(self, source: str):
        self._ascii: str = source
    @property
    def ascii(self) -> str:
```

```

        return self._ascii
    @property
    def transformed(self) -> str:
        return replace(self._ascii)

class Renderer(PostLex):
    identifiers: set[tuple[str, str]]

    def process(self, stream: Iterator[Token]) -> Iterator[Token]:
        for token in stream:
            if token.type != "NAME":
                yield token
                continue
            orig_value: str = token.value
            fragments = split_unicode_fragments(token.value)
            transformed: list[str] = list(map(
                lambda x: x.transformed if isinstance(x,
UnicodeFragment) else x, fragments
            ))
            formatted = "".join(transformed)
            self._identifiers.add((orig_value, formatted))
            yield token

```

With all instances of Unicode fragments extracted from the source code, two actions are taken

- All \LaTeX expressions in the source code are replaced with their Unicode approximation. The resulting pseudocode source file is saved to disk under the filename `rendered_source.pc`. This file is not guaranteed to be syntactically correct, but may be used for presentation or demonstration purposes. The pseudocode string produced at this stage will be shown in the debugger 5.4 interface.
- All \LaTeX expressions in the source are normalised to conform to the target language's naming syntax. This involves stripping all non-alphanumeric characters except for underscore characters. As the transpiler's target language (Python) does not support the full Unicode alphabet, this step is necessary to avoid producing a syntactically invalid Python program. Edge cases, such as where a normalised identifier begins with a digit, are accounted for.

```
// input pseudocode program
$\alpha$ = 1.5
```

```
// rendered_source.pc
 $\alpha$  = 1.5
```

```
# output Python program
alpha = 1.5
```

5.1.5 Collisions With Python Keywords

As discussed in 4.4.3, it is possible to declare pseudocode variables whose names are reserved words in the target language (Python). To address this issue without exposing implementation details of the target language, a post-processing routine is applied to append the Unicode character `_` (undertie) to any identifier whose name is a Python keyword. The undertie character is not typable in \LaTeX math mode without external packages and thus cannot be inserted into a pseudocode program.

```
class KeywordNormalizer(PostLex):
    UNDERTIE = '\u203F'
    PY_KEYWORDS = {"assert", "async", "await", "class", ...}
    def _normalize(self, token: Token) -> None:
        if token.value not in self.PY_KEYWORDS:
            return
        orig = token.value
        token.value = orig + self.UNDERTIE
    def process(self, stream: Iterator[Token]) -> Iterator[
Token]:
        for token in stream:
            if token.type == "NAME":
                self._normalize(token)
            yield token
```

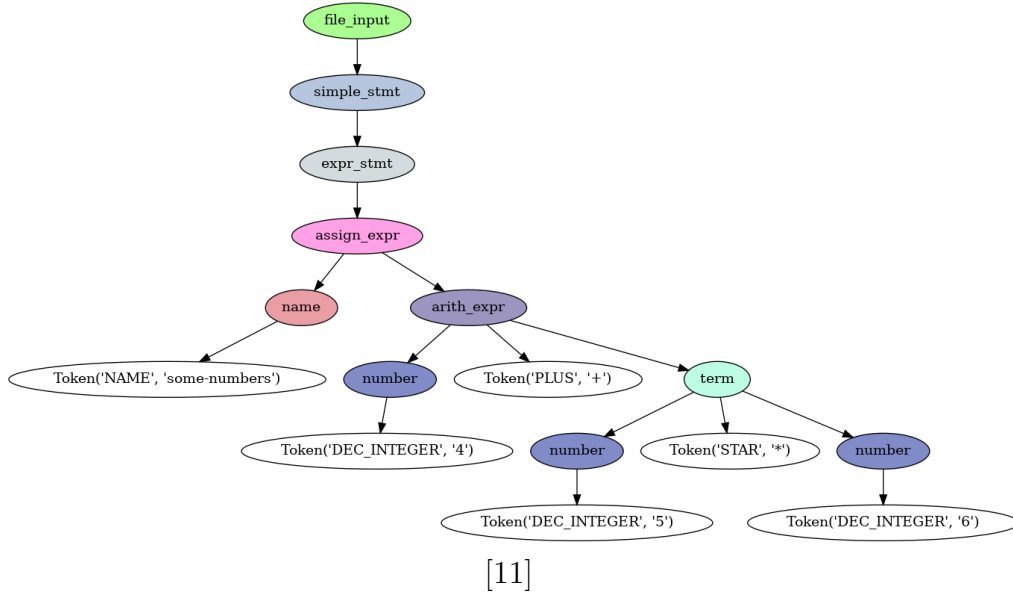


Figure 5.2: Parse tree produced from the string `some-numbers = 4 + 5 * 6`.

5.2 Parser

The pseudocode parser is generated using the Lark parser generator [14], which constructs a concrete parser from a context-free grammar 8.1 defined in Extended Backus–Naur Form (EBNF). The parser consumes the stream of tokens produced by the lexer and constructs a structured representation of the program in the form of a syntax tree. This tree captures the syntactic structure of the input while abstracting away lexical details. To simplify downstream processing, the grammar applies tree-shaping transformations via the inlining and aliasing features provided by Lark. In particular, certain rules that produce a single non-terminal child are inlined, effectively collapsing them in the final parse tree. This results in a more compact and semantically meaningful tree structure that is easier to traverse and transform in later stages of the compilation pipeline.

Figure 5.2 demonstrates the parser and its tree-shaping transformations. Certain nodes (`expr`, `factor`, etc.) with only one child have been inlined to reduce the tree size and reduce the complexity of downstream processing.

5.2.1 Parsing Algorithm

Lark supports a variety of parsing algorithms that offer different time and memory trade-offs. There are the CYK, Earley and LALR parsing algorithms.

- The **CYK** parsing algorithm can parse any context free grammar 2.5. The algorithm's worst-case runtime complexity is $O(n^3 \cdot |G|)$, where n is the length of the input string and $|G|$ is the size of the grammar, in terms of the number of production rules. CYK is the most computationally expensive, but it offers the advantage of being able to parse any input, including grammars with ambiguity. Although the CYK parsing algorithm is theoretically capable of handling any context-free grammar, it proved impractical in the context of this project. When used with Lark[14], the CYK parser exhibited unstable behaviour, producing parsing errors non-deterministically and failing to process input files exceeding approximately 1000 characters in length. These issues stem from CYK being included in Lark primarily for legacy support and is not actively recommended for production use. As such, it was excluded from consideration.
- The **LALR** (Look-Ahead LR) parsing algorithm is a deterministic, bottom-up parsing technique used to parse a subset of context-free grammars. It combines the parsing power of canonical LR parsers with the efficiency of a smaller parse table, making it well-suited for programming languages with well-structured, unambiguous grammars. The LALR algorithm operates in linear time, with a worst-case runtime complexity of $O(n)$, where n is the length of the input string. Among the parsers supported by Lark, LALR offers the best performance but requires the grammar to be free of ambiguity.
- The **Earley** parsing algorithm can parse any context-free grammar 2.5, including those that are ambiguous or left-recursive. Its worst-case runtime complexity is $O(n^3)$ for arbitrary grammars, where n is the length of the input string. However, for unambiguous grammars or those with certain constraints, it can perform more efficiently, achieving $O(n^2)$ or even $O(n)$ in best-case scenarios. Among the parsers offered by Lark, Earley provides a good balance between expressiveness and performance, making it suitable for grammars with complex or unpredictable structure.

```

parameters: name ("," name)*
funcdef: name "(" [parameters] ")" _NEWLINE function_body

arguments: expr ("," expr)*
funcall: name "(" [arguments] ")"

SUM-ARRAY(A, n)
    total = 0
    for i = 1 to n
        total = total + A[i]
    print total

nums = [1, 2, 3]
n = 3
SUM-ARRAY(nums, n)
print "done"

```

Figure 5.3: Example pseudocode program highlighting parsing ambiguity.

LALR is the most efficient parsing algorithm supported by Lark. To parse a language using LALR, its grammar must be free of ambiguity. This posed several challenges in parsing pseudocode programs, given the language’s inherent ambiguity.

Figure 5.3 shows a pseudocode program and a snippet of the pseudocode grammar for parsing function calls and definitions. The second-to-last line highlights one example of unavoidable ambiguity in the pseudocode language as described in *Introduction to Algorithms* [2], where it is not clear whether the statement should be parsed as a function *call* or as a function *declaration*.

Given the difficulties in adapting the pseudocode language to a LALR-compatible grammar and the limited support for Lark’s CYK parser, the Earley algorithm was chosen to implement the pseudocode parser. Despite its poor worst-case time complexity, it is shown that the Earley runs in approximately linear time when parsing pseudocode programs 6.2 described in *Introduction to Algorithms* [2].

```

def __init__(self, grammar: str):
    super().__init__()
    self._grammar = grammar
    self._renderer = Renderer()
    self._lark = Lark(self._grammar, propagate_positions=True,

```

```

start="file_input", postlex=PostLexPipeline([
    PythonIndenter(), self._renderer, UnicodeFormatter(),
    KeywordNormalizer()
]), parser="earley")
def parse(self, source_code: str) -> Tree:
    # append trailing newline if not present
    if source_code[-1] != "\n":
        source_code += "\n"
    ast = self._lark.parse(source_code)
    # replace latex expressions with unicode to use in debugger
    # GUI and other presentations
    # `ast` has already transformed latex expressions to ascii
    identifiers: set[tuple[str, str]] = self._renderer.
identifiers
    self._rendered_source = copy(source_code)
    for orig, rendered in identifiers:
        self._rendered_source = self._rendered_source.replace(
orig, rendered)

    return ast

```

5.2.2 Disambiguating Hyphens in CLRS-Style Pseudocode

CLRS pseudocode frequently uses hyphens in function and variable identifiers. This presents a challenge in parsing, where it is not always clear if a hyphen is used as a character in an identifier or as an arithmetic operator. Figure 5.4 highlights an example of this behaviour, where the `if` condition could be parsed as either a variable reference or as an arithmetic operation.

```

if foo-bar > 0
    return TRUE

```

Figure 5.4: Unclear usage of the hyphen character in CLRS pseudocode.

To resolve this ambiguity, the language specification requires that arithmetic operators be surrounded by whitespace, whereas hyphens within identifiers must appear without whitespace. This disambiguation rule ensures that expressions like `foo - bar` are correctly parsed as arithmetic operations, while identifiers such as `INSERTION-SORT` or `foo-bar` remain valid

and distinct. This convention is also consistent with most programming language style guides, which recommend the use of surrounding whitespace in binary operations to improve code clarity and readability. Enforcing this rule in the lexer and grammar effectively eliminates the parsing ambiguity without significantly impacting the expressiveness or readability of the pseudocode.

5.2.3 REPL Parser

The Read-Evaluate-Print Loop (REPL) component uses a dedicated parser that closely mirrors the main pseudocode parser. It is constructed from the same grammar definition 8.1 and shares the same lexer and parsing rules. However, to support the interactive nature of REPL input, the parser is configured with a different start symbol: `single_input` instead of `file_input`.

This alternate start symbol allows the REPL to accept and evaluate partial or stand-alone statements, such as individual expressions, assignments, or control structures, without requiring a complete source file. It enables the parser to return valid results for incomplete programs or multi-line inputs incrementally entered by the user.

Aside from the modified entry point, the REPL parser is functionally identical to the main parser and produces the same abstract syntax tree structure. This shared architecture ensures consistency between interactive and file-based program execution.

```
class ReplParser(Parser):
    def __init__(self, grammar: str):
        self._lark = Lark(
            grammar,
            start="single_input",
            postlex=PostLexPipeline([
                PythonIndenter(),
                UnicodeFormatter(),
                KeywordNormalizer(),
            ]))

    def parse(self, source_code: str) -> Tree:
        return self._lark.parse(source_code)
```

```

class Transpiler(Transformer):
    def block_stmt(self, args: list[str]) -> str:
        block = self._indent(args)
        return "\n".join(block)

    def stmt(self, args: list[str]) -> str:
        return args[0] + ";\n"

    def expr(self, args: list[str]) -> str:
        return args[0]

    def term(self, args: list[str]) -> str:
        return " + ".join(args)

    def DEC_INTEGER(self, tok: Token) -> str:
        return str(tok.value)

```

Figure 5.5: Excerpt implementation of a bottom-up AST to Python transpiler.

5.3 Transpiler

The transpiler implementation accepts the abstract syntax tree (AST) produced by the parser 5.2 and recursively reduces each sub-tree to a corresponding string of code in the target language (Python). These code fragments are then combined into a single string of executable Python code.

The lexer 5.1 and parser 5.2 together comprise the *frontend* of the compilation pipeline, while the transpiler functions as the *backend*. This frontend–backend distinction is made with the goal of allowing alternative compiler frontends to reuse the same backend implementation. In practice, this makes it possible for multiple dialects of the pseudocode language, or even entirely different languages, to target a shared backend and produce consistent output. This concept is discussed further in Section 7.3.1.

The transpiler is implemented as a Python class. For each production rule and terminal symbol in the grammar, the transpiler defines a public method, representing the mapping from nodes in the syntax tree to strings of Python code. In this project, two implementations of the transpiler are defined due to technical debt and time constraints.

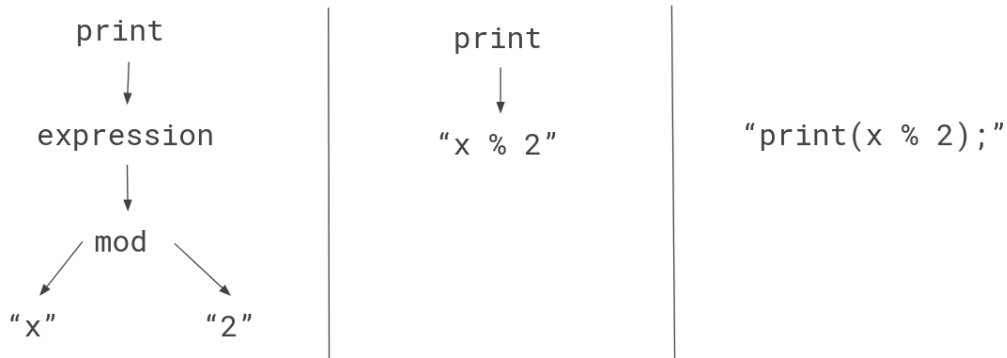


Figure 5.6: Demonstration of a bottom-up transpiler.

5.3.1 Bottom-Up Transpiler

The initial implementation of the transpiler traverses the tree beginning with the leaf nodes and works up to the root of the AST. As mentioned previously 4.5, leaf nodes in the syntax tree correspond to terminal symbols that can be trivially mapped to their equivalents in the target language. The mapping function for non-leaf nodes will take in as input the strings produced by transpiling its children and transform them into a single string of syntactically valid Python code. This process will be continued until all nodes in the syntax tree have been mapped to Python code, and the resulting string will be the output of the transpiler subsystem.

Figure 5.5 shows a demonstration of how this is implemented in Python. Figure 5.6 provides a visual aid to demonstrate the transformation from an abstract syntax tree to a Python program.

5.3.2 Debug Transpiler

While the bottom-up transpiler described in the previous section effectively generates syntactically valid Python code, it is not well-suited for debugging purposes. The debugger subsystem requires additional metadata, such as line numbers and source positions, to be inserted into the generated code. The bottom-up approach, which relies on recursively reducing child nodes to strings before combining them, makes it difficult to inject such metadata at the correct points in the transpilation process.

To address this limitation, a separate transpiler was implemented specifically for use in debug mode. This *debug transpiler* uses a top-down visitor pattern, traversing the abstract syntax tree starting from the root node and visiting each child node recursively. This traversal strategy provides more control over the transpilation process, allowing metadata and instrumentation code to be inserted as each node is encountered and before its children are processed. As a result, it becomes possible to maintain an accurate mapping between the original pseudocode source and the generated Python code, enabling the debugger to display meaningful runtime information and source code references.

The decision to adopt a top-down approach for the debug transpiler was influenced by both technical debt and time constraints. Retrofitting the bottom-up transpiler to support debug metadata would have required significant refactoring. Although this introduces duplication with two implementations of the transpiler, the logic can be refactored in the future to adhere to best practices for code duplication.

```

class DebugTranspiler(Interpreter, PccTranspiler):
    def __line_marker(self, tree: Tree) -> str:
        return f" # l:{tree.meta.line} "

    def block_stmt(self, tree: Tree) -> str:
        return self._indent_all_lines("\n".join(self.
visit_children(tree)))

    def stmt(self, tree: Tree) -> str:
        return self.visit(tree.children[0]) + ";\n"

    def expr(self, tree: Tree) -> str:
        return self.visit(tree.children[0])

    def funcdef(self, tree: Tree) -> str:
        func_name, parameters, body = self.visit_children(tree)
        if parameters is None:
            parameters = ""
        return \
            f"def {func_name}({parameters}):" + f"{self.
__line_marker(tree)}\n" \
            + body + "\n"

```

5.4 Debugger

The system's debugger, `PccDb`, is implemented as a subclass of Python's built-in debugger, `Pdb`[15]. The main challenge experienced when implementing the pseudocode debugger involved communicating with `Pdb`, and exchanging data between the debugger process and the program process. Rather than exposing a set of public methods to facilitate information exchange, `Pdb` accepts debugger commands to be issued via the process' standard input. The outputs from `Pdb` commands will then be written to the process' standard output. Alternatively, some commands may be issued to `Pdb` by modifying the private internal state of the `Pdb` object, however, this is not the recommended usage and is subject to change in future versions of Python.

The extension to `Pdb` acts as a wrapper around the built-in `Pdb` interface. A number of public methods are defined to manage the interactive execution of a pseudocode program. The public methods are then mapped to `Pdb` commands that will be issued to the debugger, either via the process' standard input or by inserting them into `Pdb`'s internal command queue.

The debugger is presented to the user in the form of a web-based graphical user interface. Upon invoking the debugger, a local web server will be spawned. The web server exposes several endpoints to provide static assets and exchange data between the client web browser and the underlying debugger system.

5.4.1 Web-Based Debugger

To implement a graphical user interface to the pseudocode debugger, an existing library, Web-Pdb 2.7, is used. For this project, Web-Pdb is forked and adapted to aid in debugging pseudocode programs. Upon launching the debugger, the user interface is opened in the user's browser, static HTML, CSS and JavaScript assets are received from the server, and a WebSocket¹ connection is established between the browser and the underlying debugger process. The WebSocket is the primary mode of communication between the client and the debugger.

Commands are issued by the client to Pdb via the WebSocket connection. Dynamic data is transferred from the server to the client via the endpoint `/frame-data`. The data shared through this endpoint includes:

- Source code listing.
- Current execution line.
- File and directory name of the executing program.
- Local and global variables and their current values.
- Line numbers of all active breakpoints.

¹WebSocket is a computer communications protocol, providing a simultaneous two-way communication channel over a single TCP connection.

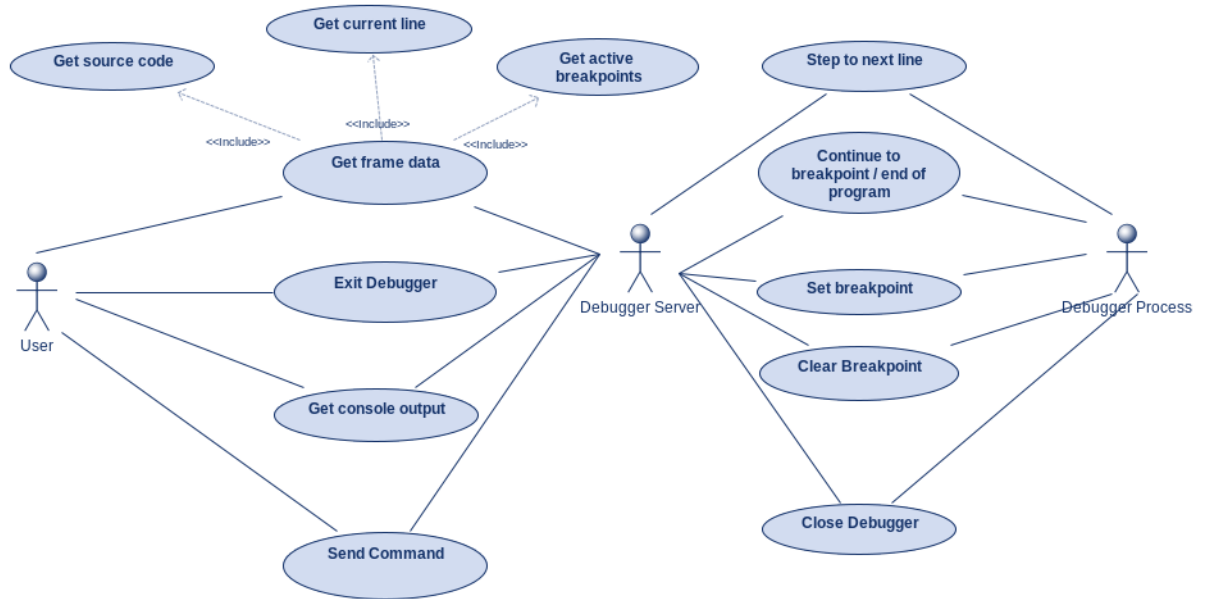


Figure 5.7: Pseudocode debugger use-case diagram

5.5 Public API

The system features a public API for invoking the transpiler. The API is implemented with the user experience in mind. Care is taken to follow widely accepted conventions for such command-line interfaces.

The CLI is implemented using a library called `click`[13]. Click is a Python package to create rich command-line interfaces by providing tools for argument parsing and automatic help message generation. The command line arguments and options implemented match those defined in 4.5.

```

@click.command(help=usage())
@click.option("-v", "--version", type=click.BOOL, default=False,
              is_flag=True)
@click.option("-h", "--help", type=click.BOOL, default=False,
              is_flag=True)
@click.option("-d", "--debug", type=click.BOOL, default=False,
              is_flag=True)
@click.option("-o", "--output", type=click.STRING, default="out/output.py")
@click.option("-r", "--output-rendered-source", type=click.BOOL,
              default=True)
@click.argument("source_file_path", type=click.STRING, default="")
  
```

```
def main(version: bool, help: bool, debug: bool, output: str,
source_file_path: str, output_rendered_source: bool):
    if help:
        print(usage())
        sys.exit(0)
    if version:
        print("Pseudo-Code Compiler, version 1.0.0")
        sys.exit(0)
    if not source_file_path:
        print("Error: No source file provided")
        print(usage())
        sys.exit(1)
    ...
```

```
# Example usage
python pcc.py -d src/linked_list.pc
```

5.6 IDE Support for Pseudocode Programming

To improve the user experience when writing and executing pseudocode programs, a Visual Studio Code extension was developed. The extension provides syntax highlighting, basic syntax awareness, and editor integration tailored specifically to the pseudocode language used in this project.

Syntax highlighting is implemented by defining regular expression patterns that capture the key syntactic constructs of the pseudocode language. These patterns are mapped to semantic token types, which are then styled by the editor according to the user's selected colour scheme.

```
{
  "comments": {
    "patterns": [{
      "name": "comment.line",
      "match": "\\s//\\s.*\\n"
    }]
  },
  "keywords": {
    "patterns": [{
      "name": "keyword.control.pseudocode",
      "match": "\\b(a|array|be|downto|else|for|if|let|new|print|repeat|return|to|until|while)\\b"
    }]
  },
  "numbers": {
```

```

    "patterns": [{
      "name": "constant.numeric",
      "match": "-?[0123456789]+"
    }]
  },
}

```

In addition to syntax highlighting, the extension integrates with several editor features to enhance the programming experience. These include automatic closing of brackets and quotation marks, support for inserting and removing line comments through editor shortcuts, and bracket pair highlighting.

```

{
  "comments": {
    // symbol used for single line comment.
    "lineComment": "//",
  },
  // symbols used as brackets
  "brackets": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"]
  ],
  // symbols that are auto closed when typing
  "autoClosingPairs": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"],
    ["\\\"", "\\\""]
  ],
  // symbols that can be used to surround a selection
  "surroundingPairs": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"],
    ["\\\"", "\\\""]
  ]
}

```

5.7 Installation Script

To assist users in installing the transpiler and its dependencies, a shell script is provided to automate the installation process. In order to ensure compatibility with all major operating systems, as outlined in the system requirements 3.4.2, two installation scripts were developed: a PowerShell script for Windows systems and a POSIX-compatible shell script for macOS and Linux platforms.

The actions taken by the installation script are as follows:

- Ensure Python version 3.10 or later is installed on the host machine.
- Clone the latest release of the pseudocode compiler repository from GitHub.
- Prompt the user to optionally create a Python virtual environment to encapsulate the project's dependencies
- Install any runtime dependencies for the project.
- Prompt the user to install the Visual Studio Code extension 5.6 to enable pseudocode syntax highlighting.
- If desired, clone the extension from GitHub to VSCode's extensions directory.

Chapter 6

Evaluation

6.1 Testing

For the purpose of verifying the correctness and reliability of the system, two primary testing methods are employed: unit testing 6.1.1 and end-to-end testing 6.1.2. Unit testing is used to validate the behaviour of individual components in isolation. These tests ensure that each part of the system functions correctly under a variety of controlled input scenarios. End-to-end testing, on the other hand, is used to assess the behaviour of the system as a whole by running complete pseudocode programs through the entire compilation and execution pipeline to verify that the final outputs match expected results. Together, these complementary methods provide confidence in both the internal correctness of the system's components and its overall functionality in realistic usage scenarios.

6.1.1 Unit Testing

Many unit tests are implemented to test the functionality of the software system and its components. The unit testing suite for this project is implemented using Python's built-in `unittest` framework. This decision was made to prioritise simplicity, portability, and ease of integration within the existing Python ecosystem. As `unittest` is included in the Python standard library, it eliminates the need for an additional third-party dependency, ensuring that the project remains lightweight and easily installable across different systems. Generally, unit tests are defined on a per-class basis, where each class defined in the implementation will have an associated suite of unit tests.

```

class TestUnicodeFragment(unittest.TestCase):
    def test_split_unicode_fragment(self):
        source: str = r"alpha-$\beta$-gamma-$\delta$"
        fragments: list[str | UnicodeFragment] =
split_unicode_fragments(source)
        self.assertEqual([
            "alpha-",
            UnicodeFragment("\\beta"),
            "-gamma-",
            UnicodeFragment("\\delta"),
        ], fragments)

```

Figure 6.1: Example unit test with Python’s *unittest* module.

The pseudocode parser 5.2 is created using the Lark [14] package, which provides its own suite of comprehensive unit tests. As such, no additional unit tests are defined for the parser. Instead, the parser is evaluated in the system’s end-to-end testing 6.1.2.

6.1.2 End-to-End Testing

In addition to unit testing individual components, a series of end-to-end tests 6.2 are provided to verify the correctness of the system as a whole. These tests validate the entire workflow, from reading pseudocode input to generating executable Python code.

Each end-to-end test consists of a pseudocode algorithm described in *Introduction to Algorithms* [2] and a hand-written Python translation of the algorithm. The hand-written Python implementation is taken to be correct for all possible inputs. The testing framework invokes the transpiler on the pseudocode file to produce a Python translation. Both the hand-written and automatically generated Python programs are evaluated on a collection of test inputs. It is expected that the programmatically generated Python code should produce the same output as the hand-written program for all inputs. The test case is reported as a failure if the two programs differ on any input.

By simulating real-world usage scenarios, these tests provide a high level of confidence in the correctness and robustness of the system’s behaviour. This approach ensures that changes to individual components do not introduce regressions or break the overall functionality of the system.

Algorithm	Reference	Pass/Fail
Insertion Sort	[2][Chapter 2.1]	Pass
Matrix Multiply	[2][Chapter 4.1]	Pass
Heap Sort	[2][Chapter 6.4]	Pass
Quick Sort	[2][Chapter 7.1]	Pass
Counting Sort	[2][Chapter 8.2]	Pass
Minimum	[2][Chapter 9.1]	Pass
Heap Push/Pop	[2][Chapter 6.3]	Pass
Queue Operations	[2][Chapter 10.1.3]	Pass
Linked List Operations	[2][Chapter 10.2]	Pass
Hash Table (Chained Hashing)	[2][Chapter 11.2]	Pass
Binary Search Tree	[2][Chapter 12]	Pass
Depth-First Search	[2][Chapter 20.3]	Pass
Breadth-First Search	[2][Chapter 20.2]	Pass
Dijkstra's Algorithm	[2][Chapter 22.3]	Pass

Figure 6.2: Algorithms used for end-to-end testing

6.2 Parsing Benchmark

The system's non-functional requirements 3.4.3 define that the system must parse a pseudocode program in linear time with respect to the number of characters in the source. A series of benchmarks were conducted to evaluate the performance of the parser. The benchmark measures the time taken to parse pseudocode programs of varying sizes, providing insight into the efficiency of the selected parsing strategy and its suitability for different input categories.

The benchmarks were performed using a collection of pseudocode programs taken from *Introduction to Algorithms* [2], ranging from small examples containing only a single statement, to larger programs consisting of thousands of characters. Each program was parsed multiple times to obtain an average parsing time and account for variability in system performance.

The benchmark was performed using both the **CYK** and **Earley** parsing algorithms. However, the CYK parser would fail to parse larger programs with more than approximately one thousand characters. Figure 6.2 shows the results of the benchmark on both parsing algorithms.

The results of the benchmark indicate the CYK parser being slightly more performant on small programs, although it becomes inefficient on larger inputs and fails outright to parse sufficiently large programs. Addi-

tionally, the CYK implementation bundled in Lark [14] is not recommended for production use, as discussed previously 5.2.1. For these reasons, the CYK parsing algorithm was not used in the system implementation.

The benchmark validates the practicality of the Earley parser for this system, verifying its conformance to the system requirements and its ability to handle real-world pseudocode programs efficiently within the intended usage context.

6.3 User Evaluation

In addition to technical validation through automated tests, a user evaluation was conducted to gather qualitative feedback on the usability and functionality of the system from a practical perspective. The goal of this evaluation was to assess how effectively the system supports users in writing, executing, and debugging pseudocode programs.

A small group of participants, consisting of computer science students, were invited to use the system and perform a set of predefined tasks. These tasks included writing simple pseudocode programs, executing them using the transpiler and interactive execution environment, and exploring the integrated Visual Studio Code extension for syntax highlighting and editor integration. Participants were given a brief tutorial on the system’s usage and documentation, after which they completed the tasks independently.

Participants were asked to provide feedback on their experience through an informal questionnaire and discussion. Questions asked included the ease of writing and running pseudocode, the clarity of error messages and debug output, and the system’s overall usability. The full list of questions asked is provided below. Unless specified otherwise, questions were multiple-choice, with the options *Strongly Disagree*, *Disagree*, *Neutral*, *Agree*, *Strongly Agree*

- "Have you taken, or are you currently taking a course on introductory programming?" (Yes/No)
- "Have you taken, or are you currently taking a university-level course on algorithms?" (Yes/No)
- "Have you ever worked professionally in software development or a related field?" (Yes/No)
- "The software was easy to install"

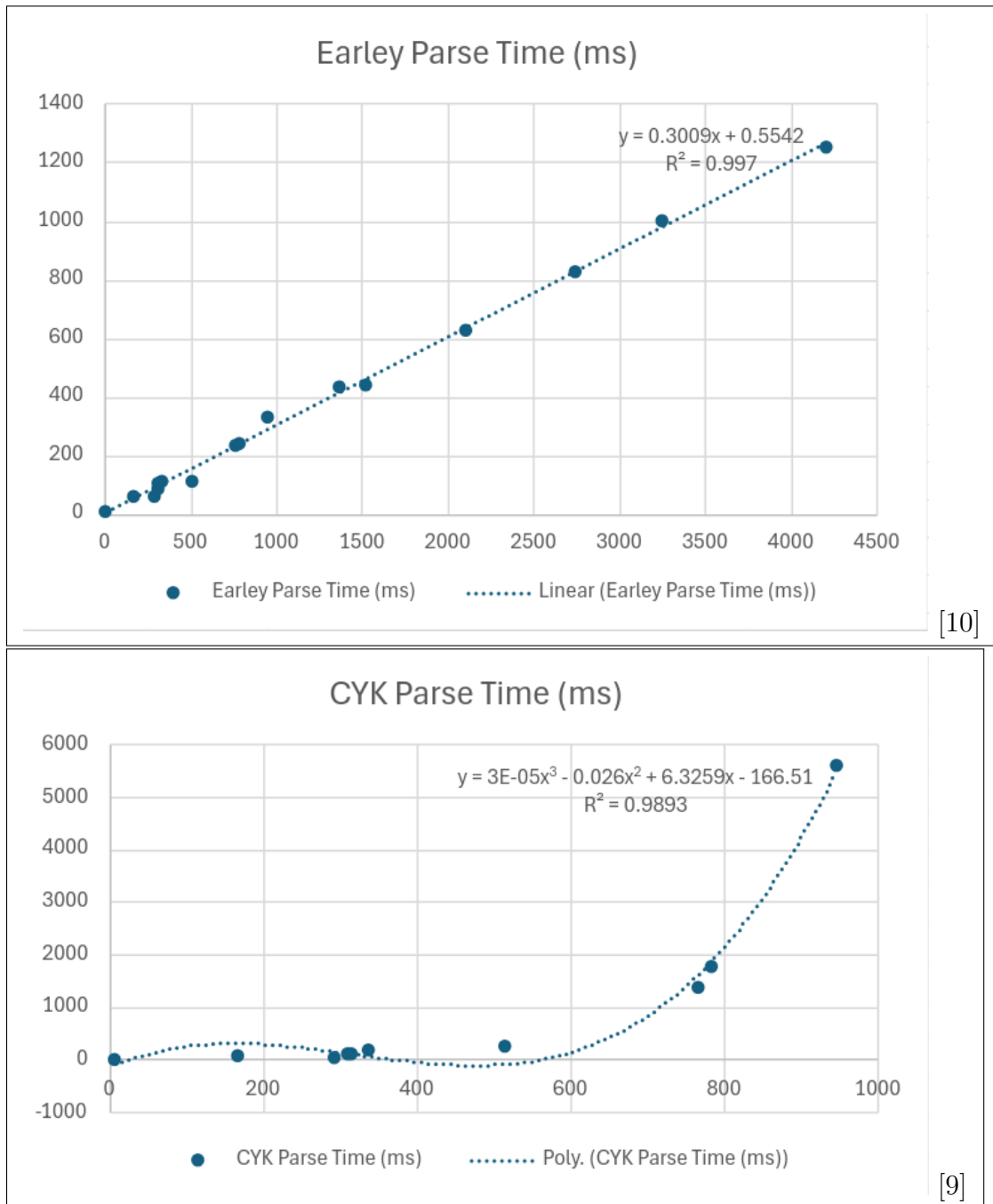


Figure 6.3: Time (ms) to parse a pseudocode program with respect to its character count.

- "Did you use a VSCode extension to provide syntax highlighting for *.pc files?" (Yes/No)
- "The syntax highlighting helped in reading and reasoning about pseudocode." (optional)
- "The syntax highlighting was helpful in identifying syntactical errors in a pseudocode program." (optional)
- "The pseudocode language was intuitive to understand."
- "The pseudocode language was intuitive to program with."
- "The pcc command-line interface was easy to work with."
- "The Interactive Execution Environment was easy to use."
- "The Interactive Execution Environment was helpful in understanding algorithms described in the textbook."
- "Are there any missing features you believe would improve the compiler or execution environment?" (short paragraph, optional)
- "Criticisms of the software system." (short paragraph, optional)
- "General comments on your experience with the software system." (short paragraph, optional)

Figure 6.4 shows the responses received for a subset of the questionnaire.

When users were asked about missing features they believed would improve the software system, two responses were given.

- "Cheat sheet of all pseudocode syntax"
- "The installation script of the environment works extremely well, a quick setup guide to problem-solve any existing issues might be an added bonus"

When asked for general comments on the software system, the following responses were received from users.

- "Very interesting, and impressive"

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
"The software was easy to install."	1	0	0	2	2
"The syntax highlighting helped in reading and reasoning about pseudocode."	0	0	1	2	0
"The syntax highlighting was helpful in identifying syntactical errors in a pseudocode program."	0	0	0	3	2
"The pseudocode language was intuitive to understand."	0	0	1	3	1
"The pseudocode language was intuitive to program with."	0	0	3	2	0
"The pcc command-line interface was easy to work with."	0	0	2	2	1
"The Interactive Execution Environment was easy to use."	0	0	1	3	1
"The Interactive Execution Environment was helpful in understanding algorithms described in the textbook."	0	0	0	4	1

Figure 6.4: Responses given to a subset of questions in the user feedback survey.

- “Was able to create sorting algorithms with pseudocode, & the IEE helped a lot with understanding the code”
- “Slight learning curve to understand how to use the system, however once past the curve I found the system intuitive to use”
- “Was quite cool to go through the motions for displaying the responses in a different way”

6.4 Appraisal

While the sample size was limited, the user evaluation offered valuable practical validation of the system’s design and confirmed its suitability for educational and instructional use cases. Overall, the feedback indicated that users found the system beneficial as a learning tool. However, some participants encountered difficulties during installation, both when attempting manual setup and when using the provided installation script 5.7. These issues primarily stemmed from their systems’ default Python interpreters being linked to older versions that were not supported by the software.

This feedback highlighted an important area for improvement and was taken into account when identifying opportunities for future work. In particular, it suggests the potential value of enhancing the installation process to include automatic version detection and improved logic for reporting and troubleshooting installation errors.

The results of the unit testing 6.1.1, end-to-end testing 6.1.2, and user evaluation 6.3 suggest that the software system successfully achieves the system’s requirements 3.4.2 and the objectives outlined in Section 3.1. Unit tests confirm the correctness of individual components, ensuring that the core compilation pipeline reliably transforms pseudocode into executable Python code. End-to-end tests validated the overall workflow.

Feedback from the user evaluation further reinforced these findings. Participants were able to write and run pseudocode programs without prior knowledge of Python, and expressed that the syntax and layout conventions felt intuitive for algorithmic work. Additionally, the generated Python code was found to be readable and modifiable, meeting the objective of producing understandable output for users who wished to inspect or modify the transpiled code.

While minor criticisms and difficulties were reported, these did not directly affect the system’s ability to meet its core functional goals. Overall,

the combination of technical testing and practical user feedback indicates that the system satisfies its intended purpose and serves as a functional, accessible tool for executing pseudocode algorithms.

6.5 Limitations

6.5.1 Lack of Learning Resources

During user testing, one participant reported difficulties in learning the syntax conventions of the CLRS-style pseudocode language 6.3. Another user reported a learning curve to understanding and effectively using the system 6.3. This sentiment was further echoed in informal discussions, where some users expressed uncertainty about specific language constructs and formatting requirements. Although the syntax closely follows established conventions from *Introduction to Algorithms* [2], it became clear that additional guidance would improve the user experience, particularly for those without prior exposure to CLRS pseudocode notation.

This highlights a limitation in the system’s current documentation and onboarding support. Addressing this would involve developing clearer, beginner-friendly learning materials such as interactive examples, tutorials, or video walkthroughs, making the system more approachable for a wider audience.

6.5.2 Limitations in LaTeX Symbol Translation

One of the requirements of the system is to allow programmers to embed LaTeX expressions within pseudocode identifiers 2, which are then translated into visually similar Unicode or ASCII equivalents in the transpiled output. This feature was designed to enhance the readability and expressiveness of pseudocode programs, particularly in domains where mathematical notation is commonly used.

However, a limitation of this approach is that not all \LaTeX symbols have direct Unicode or ASCII equivalents. In particular, certain characters, such as the superscript uppercase $\text{\textbf{S}}$, do not have a Unicode equivalent. Other characters (e.g. superscript lowercase $\text{\textbf{q}}$) have equivalents in the Unicode alphabet that are not supported in many fonts. In such cases, the system substitutes an approximate ASCII representation.

This inconsistency can lead to the occasional loss of visual clarity or accuracy in the final translated program, especially when users rely on specific notational conventions. Addressing this limitation would require expanding the translation system by providing warnings for unsupported symbols or integrating optional LaTeX rendering in the output environment. While this feature is effective in most common use cases, it currently lacks comprehensive coverage for the full range of \LaTeX expressions that may be encountered in pseudocode.

Chapter 7

Conclusions

7.1 Reflection

I have learned many things throughout the design and implementation of this project. This project has been a significant learning experience, offering both technical and personal development opportunities. One of the most valuable outcomes was gaining a practical understanding of compiler theory and programming language design. While I had previously encountered these topics in a theoretical context, implementing a working transpiler from scratch allowed me to engage with real-world challenges and improve my knowledge in the area.

Throughout the project, I also developed a much deeper knowledge of Python than I had learned in previous programming modules, particularly in its modern language features and metaprogramming capabilities. Working extensively with the Lark parsing toolkit [14] provided insight into parser generation and offered the opportunity to contribute enhancements to the open-source project. This experience improved both my technical abilities and my confidence in contributing to collaborative software projects, which I intend to continue in the future.

In addition to technical skills, this project highlighted some areas for personal growth, particularly in time management and project planning. Some stages of the project involved more time than anticipated, which impacted the available time for other features, such as advanced debugger integration and runtime complexity estimation 7.3.3. I did not allocate enough time to the report for this project, leading to stress in the days leading up to the submission deadline. If I were to begin this project again, I would ensure that work on the report was done iteratively alongside the

implementation. This experience has emphasised the importance of early prototyping, milestone setting, time management, and realistic scheduling when managing complex software projects.

Overall, this project has enhanced my technical, problem-solving, and organisational skills while providing valuable exposure to advanced Python programming concepts. These skills and lessons will be highly applicable in my future career.

7.2 How the Project was Conducted

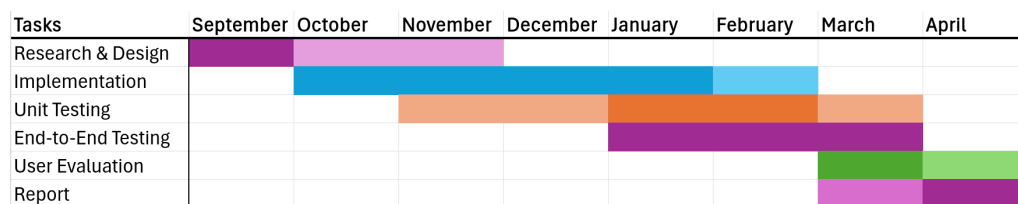


Figure 7.1: A Gantt chart of the time taken to do each general task.

Work on the project began with an investigation into the pseudocode language described in *Introduction to Algorithms*. Alongside the analysis of the pseudocode language, research was conducted on the various approaches to implementing the task at hand.

After the research phase, implementation of the project’s core components began. This included implementing the parser 5.2, transpiler 5.3, and debugger 5.4. Unit testing 6.1.1 was performed incrementally alongside the core implementation, while end-to-end testing 6.1.2 and user evaluation 6.3 were left until a *minimum viable product* had been developed.

Towards the end of the development cycle, work began on the report. After the SCSIT open day, all work done was focused on the report. If this project were repeated, the report would be done alongside the implementation and testing stages to reduce the stress involved in writing the report in April.

7.3 Future Work

7.3.1 Support For Alternative Pseudocode Dialects

The transpiler system can be divided into a logical frontend and backend. The frontend logic involves the tokenisation and parsing of code in the input language (Pseudocode), while the compiler backend transforms the program's AST representation to an executable program in the output language.

There is no standardised format for pseudocode. Sources will often use differing notations to express the same logical concepts. As an example, figure 7.2 highlights the syntactic differences between two dialects of pseudocode.

Given that the compiler backend accepts an abstract syntax tree produced by parsing a program, it can be considered language-agnostic. If one were to develop a language frontend targeting another dialect of pseudocode, it would be possible to reuse the existing backend infrastructure developed as part of this project.

This technique is common in programming language implementations. For example, the LLVM project [5] provides a modular, reusable backend that accepts an intermediate representation (IR) generated by different language frontends, such as Clang for C/C++ or Swiftc for the Swift language. Each frontend is responsible for parsing its source language and producing a compatible intermediate representation, which is then processed by the shared LLVM backend for optimisation and code generation. Adopting a similar frontend-backend separation in this project opens opportunities for supporting multiple pseudocode dialects in the future without requiring modifications to the core backend logic.

```

INSERTION-SORT(A, n)
  for i = 2 to n
    key = A[i]
    // Insert A[i] into the sorted subarray A[1:i-1]
    j = i - 1
    while j > 0 and A[j] > key
      A[j + 1] = A[j]
      j = j - 1
    A[j + 1] = key

```

```

i <- 1
while i < length(A)
  x <- A[i]
  j <- i
  while j > 0 and A[j - 1] > x
    A[j] <- A[j - 1]
    j <- j - 1
  end while
  A[j] <- x
  i <- i + 1
end while

```

Figure 7.2: Insertion Sort, as defined in *Introduction to Algorithms*[2], and on Wikipedia [1]

7.3.2 CLRS Pseudocode as a General-Purpose Programming Language

Another direction for future work is the extension of the pseudocode language, originally inspired by the style presented in *Introduction to Algorithms* [2], into a fully-featured, general-purpose programming language. While the current system is primarily designed to support algorithmic exercises and educational programs, expanding its capabilities would open up new applications beyond educational use.

To achieve this, the pseudocode language could be extended with a richer standard library, incorporating built-in routines for interacting with the operating system, performing disk and network input/output, handling cryptographic operations, and managing concurrency or parallelism. These additions would allow users to write practical programs entirely in pseudocode without relying on transpiled Python code for low-level system interaction.

Such enhancements would not only increase the expressive power of the language but also provide an opportunity to experiment with how algorithm-focused pseudocode can bridge the gap between high-level algorithm design and real-world application development. This would turn the pseudocode environment into a more versatile platform for both learning and rapid prototyping, particularly in educational contexts where simplicity and clarity are valued over raw performance efficiency.

Integrating these capabilities while maintaining the language’s minimalist and approachable nature would be a key design challenge, requiring careful language design decisions to balance expressive power, usability, and consistency with the original CLRS pseudocode style.

7.3.3 Collection of Performance Statistics and Runtime Complexity Estimation

An interesting opportunity for future work lies in extending the system to collect performance statistics and provide runtime complexity estimations for executed pseudocode programs. While the current implementation focuses on correctness and usability, integrating performance monitoring would enhance the system’s educational value by helping users develop an intuition for time complexity and big-O notation.

This could be achieved by instrumenting the transpiled Python code to measure key runtime metrics, such as the number of fundamental operations performed. These statistics could be collected during program execution and reported alongside the program's output, offering users immediate feedback on the practical cost of their algorithms.

Additionally, by analysing control flow structures, recursion depth, and loop iteration counts within the abstract syntax tree, the system could attempt to estimate the runtime complexity of a given program in Big O notation. Although such static analysis would have limitations to its accuracy and reliability, it could potentially provide valuable approximations in simple cases.

Incorporating these features would not only enhance the capabilities of the system but also align it more closely with the learning objectives of algorithm and data structure courses, where understanding both the correctness and efficiency of algorithms is essential.

7.3.4 Improvements to the Installation Process

It was reported during user evaluation 6.3 that a number of users found difficulties in installing and configuring the software. If more time were available, the installation script 5.7 would be rewritten as a single, cross-platform Python script. This would remove the requirement to maintain two scripts for both Windows and Unix-like systems. Additionally, a comprehensive troubleshooting guide would be created to aid in manual installation of the software, detailing common issues and their remedies, such as incompatible Python versions and missing dependencies.

Implementing these changes would result in a lower barrier to entry for users to obtain the software and an overall improved user experience.

7.3.5 Visualisation of Data Structures

An area of significant potential for future development is the visualisation of data structures within the Interactive Execution Environment. While the current implementation provides a graphical interface for inspecting variable values, it does not yet support graphical representations of complex data structures such as linked lists, trees, and graphs; instead, such structures are shown as a primitive string representation.

Incorporating data structure visualisation would greatly enhance the educational value of the system by providing users with intuitive, real-time graphical representations of their program's state. These dynamic visualisations would update interactively as the program executes, helping users better understand how data structures evolve during algorithm execution.

This capability would be particularly valuable for teaching algorithms that rely heavily on dynamic data structures, such as tree traversals, graph algorithms, and pointer-based manipulations. By making abstract structures visible and interactive, the system could improve comprehension and reduce the cognitive load associated with mentally tracking complex program states.

Acronyms

- API** Application Programming Interface. 44
- AST** Abstract Syntax Tree. 4, 5, 19–22, 40, 60
- CLI** Command-Line Interface. 25, 44
- GUI** Graphical User Interface. 12, 25
- JRE** Java Runtime Environment. 16
- JVM** Java Virtual Machine. 16
- PCC** Pseudo-Code Compiler. i
- REPL** Read-Evaluate-Print Loop. 22, 38

Bibliography

- [1] Wikipedia contributors. *Insertion Sort pseudocode*. Accessed: 2025-04-14. URL: https://en.wikipedia.org/wiki/Insertion_sort#Algorithm.
- [2] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th ed. The MIT Press, 2022. URL: <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.
- [3] Hans Dembinski. *Unicodeitplus*. Version 0.3.1. 2023. URL: <https://pypi.org/project/unicodeitplus/>.
- [4] JetBrains. *Kotlin*. URL: <https://kotlinlang.org/>.
- [5] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88.
- [6] MegaIng. *Interegular*. Version 0.3.3. 2024. URL: <https://pypi.org/project/interegular/>.
- [7] Jacob Minsky. *Abstract Syntax Tree produced from parsing a polynomial expression*. Accessed: 2025-04-05. 2025. URL: <https://github.com/lark-parser/lark/issues/1520>.
- [8] Roman Miroshnychenko. *python-web-pdb*. Version 1.6.3. Roman Miroshnychenko, 2024. URL: <https://pypi.org/project/web-pdb/>.
- [9] Colm Murphy. *CYK Parse Times*. Figure created by the author for use in this report. 2025.
- [10] Colm Murphy. *Earley Parse Times*. Figure created by the author for use in this report. 2025.
- [11] Colm Murphy. *Parse tree produced from the expression 'some-numbers = 4 + 5 * 6'*. Figure created by the author for use in this report. 2025.
- [12] Colm Murphy. *Pseudocode tokenisation example*. Figure created by the author for use in this report. 2025.

- [13] Pallets. *Click*. Version 8.1.8. 2024. URL: <https://pypi.org/project/click/>.
- [14] Erez Shinan. *Lark*. Version 1.2.2. 2024. URL: <https://pypi.org/project/lark/>.
- [15] Guido Van Rossum and Fred L. Drake. *Pdb*. 2024. URL: <https://docs.python.org/3/library/pdb.html>.
- [16] Guido Van Rossum and Fred L. Drake. *Python 3.14 list of reserved words*. Accessed: 2025-04-03. 2025. URL: https://docs.python.org/3.14/reference/lexical_analysis.html#keywords.

Chapter 8

Appendix

8.1 External Libraries Used

I used many external Python libraries to complete this project. Here is a complete list of them.

Library	License
Lark [14]	MIT
Interegular [6]	MIT
unicodeitplus [3]	BSD 3- Clause
click [13]	BSD
web-pdb [8]	MIT

8.2 Pseudocode Compiler Grammar

This is the grammar used to generate the pseudocode parser using Lark [14]. The grammar is described in Extended Backus-Baur Form ¹.

```
// Copyright: (c) 2025, Colm Murphy <colmmurphy016@gmail.com>
// GNU General Public License v3.0+ (see COPYING or https://www
    .gnu.org/licenses/gpl-3.0.txt)

single_input: _NEWLINE | simple_stmt | compound_stmt _NEWLINE
file_input: (_NEWLINE | stmt)*

funcdef: name "(" [parameters] ")" _NEWLINE function_body
```

¹The grammar contains some additional syntactic sugar provided by Lark. An overview of the syntax is provided in Lark's documentation

```

parameters: name ("," name)*

?stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (";" small_stmt)* [";"] _NEWLINE
?small_stmt: (expr_stmt | assign_stmt | flow_stmt | decl_stmt)
expr_stmt: test
assign_stmt: assign

?decl_stmt: struct_decl_stmt

struct_init_arguments: arith_expr ("," arith_expr)*
    | arith_expr (":" arith_expr)

struct_decl_stmt: single_struct_decl
    | multiple_struct_decl
struct_init: name "[" struct_init_arguments+ "]"
single_struct_decl: "let" struct_init name+ name
multiple_struct_decl: "let" struct_init ("and" struct_init)+
    name+ name

?flow_stmt: return_stmt | print_stmt | error_stmt |
    exchange_stmt
return_stmt: "return" test?
print_stmt: "print" test
error_stmt: "error" string
exchange_stmt: ("exchange" | "swap") atom_expr "with" atom_expr

?compound_stmt: if_stmt
    | while_stmt
    | for_stmt
    | repeat_stmt
    | funcdef

if_stmt: "if" test _NEWLINE block_stmt elifs else_?
elifs: elif_*
elif_: "else" "if" test _NEWLINE block_stmt
else_: else_block | else_inline
else_block: "else" _NEWLINE block_stmt
else_inline: "else" simple_stmt block_stmt?
while_stmt: "while" test _NEWLINE block_stmt
for_stmt: for_loop
    | for_iter
for_loop: "for" name "=" test range_op test _NEWLINE block_stmt
for_iter: "for" "each" name~1..2 "in" expr _NEWLINE block_stmt
    // first 'name' token should be discarded
!range_op: "to"
    | "downto"
repeat_stmt: "repeat" _NEWLINE block_stmt "until" test _NEWLINE

```

```

block_stmt: simple_stmt | _INDENT stmt+ _DEDENT

// alias for block_stmt
function_body: _INDENT stmt+ _DEDENT

assign: test "=" test

?test: comparison
      | assign_expr

assign_expr: name "=" test

?comparison: expr (_comp_op expr)*

?expr: or_expr

?or_expr: xor_expr ("or" xor_expr)*
?xor_expr: and_expr ("xor" and_expr)*
?and_expr: shift_expr ("and" shift_expr)*
?shift_expr: arith_expr (_shift_op arith_expr)*
?arith_expr: term (_add_op term)*
?term: factor (_mul_op factor)*
?factor: unary_op factor | power

!unary_op: "+" | "-" | "!"
!_add_op: "+" | "-" | "|"
!_shift_op: "<<" | ">>"
!_mul_op: "*" | "/" | "mod" | "\\\\" | "&"
!_comp_op: "<" | ">" | "==" | ">=" | "<=" | "!=" | "in" | "not"
           "in" | "is" | "is" "not"
!_power_op: "^" | "**"

?power: atom_expr (_power_op factor)*

?atom_expr: atom_expr "(" [arguments] ")" -> funcall
           | atom_expr "[" test "]" -> getitem
           | atom_expr "." name -> getattr
           | atom

?atom: name -> var
      | number
      | string
      | "{" [test ("," test)*] "}" -> set_literal
      | "[" test ("," test)* "]" -> array_literal
      | "(" test ")" -> grouping
      | "NIL" -> const_nil
      | "TRUE" -> const_true

```

```

    | "FALSE"          -> const_false

arguments: test ("," test)*

number: DEC_INTEGER | DEC_REAL
string: STRING

// other terminals

_NEWLINE: ( /\r?\n[\t ]*/ | COMMENT )+

%ignore /\t \f+/ // whitespace
%ignore /\[\t \f]*\r?\n/ // LINE_CONTINUATION
%declare _INDENT _DEDENT

// more terminals

!name: NAME
// NAME: /[a-zA-Z$][a-zA-Z0-9'\-_\^${}\]\]*/
NAME: /([a-zA-Z_]([a-zA-Z0-9'\-_\^${}\]\]+\$))*
    | (\$[a-zA-Z0-9'\-_\^${}\]\]+\$)/
COMMENT: /\[/\[/[^\n]*/

%import common.ESCAPED_STRING -> STRING

DEC_INTEGER: /[1-9][0-9]*/
    | /0+/
DEC_REAL: /[1-9][0-9]*\.[0-9]+/
    | /0+\.[0-9]+/

```

Listing 8.1: Pseudocode grammar (Lark)