Bachelor's thesis

TINYC COMPILER FRONTEND

Mykhailo Anisimov

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Petr Máj, Ph.D.
May 16, 2025

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	TinyC Compiler Frontend
Student:	Mykhailo Anisimov
Supervisor:	Ing. Petr Máj, Ph.D.
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2025/2026

Instructions

The aim of the project is to design universal compiler frontend for the TinyC programming language as used in the NI-GEN course that can be given to its students so they can focus on the middle- and back-end work. The frontend should be implemented in C++. It should parse the TinyC language into an abstract syntax tree whose representation should follow established Object Oriented Programming principles. It should be available either as a library with the AST classes directly usable by students, or as a standalone executable that will output the parsed AST in a standardized JSON format (including source location information).

The thesis should:
1) Analyze the landscape of language parsers and language agnostic AST representations (such as babel/parser for JavaScript)
2) Design and document AST representation for TinyC and its JSON format.
3) Design, document, implement and test the TinyC parser.
4) Discuss further development of the project.

Electronically approved by Ing. Michal Valenta, Ph.D. on 12 February 2025 in Prague.

Czech Technical University in Prague
Faculty of Information Technology
© 2025 Mykhailo Anisimov. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Anisimov Mykhailo. *TinyC Compiler Frontend*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2025.

I would like to sincerely thank my supervisor, Ing. Petr Máj, Ph.D., for his guidance and invaluable support throughout this thesis. My deepest gratitude also goes to my parents, whose continuous support and encouragement have provided me with the opportunity to study abroad and pursue my academic and personal growth.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

I declare that I have used AI tools during the preparation and writing of my thesis. I have verified the generated content. I confirm that I am aware that I am fully responsible for the content of the thesis.

In Prague on May 16, 2025

Abstract

This thesis presents a compiler frontend for the `tinyC` programming language used in the NI-GEN course, designed to facilitate backend implementation in diverse programming languages. The frontend offers both a standardized JSON interface for language-agnostic backend development and a direct C++ library option. Based on analysis of parser technologies and AST representations, the implementation features a recursive descent parser with LL(1) grammar, source location tracking, and a defined JSON schema. With its clean architecture and documentation, this frontend serves as an educational tool while maintaining extensibility for future enhancements.

Keywords compiler frontend, AST, recursive descent parser, `tinyC`, OOP, JSON serialization, language-agnostic design

Abstrakt

Tato práce představuje překladový frontend pro programovací jazyk `tinyC` používaný v kurzu NI-GEN, navržený k usnadnění implementace backendu v různých programovacích jazycích. Frontend nabízí jak standardizované JSON rozhraní pro jazykově nezávislý vývoj backendu, tak i přímé použití C++ knihovny. Na základě analýzy technologií parserů a reprezentací AST implementace zahrnuje rekurzivní sestupný parser s LL(1) gramatikou, sledování pozic ve zdrojovém kódu a definované JSON schéma. Díky své přehledné architektuře a dokumentaci slouží tento frontend jako výukový nástroj při zachování rozšiřitelnosti pro budoucí vylepšení.

Klíčová slova překladový frontend, AST, LL1 parser, `tinyC`, OOP, serializace JSON, jazykově nezávislý návrh

Contents

List of Figures

List of Tables

List of code listings

List of abbreviations

ADT	Algebraic Data Type
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
CBOR	Concise Binary Object Representation
CFG	Context-Free Grammar
CFL	Context-Free Language
DFA	Deterministic Finite Automaton
DTD	Document Type Definition
IDE	Integrated Development Environment
IR	Intermediate Representation
JSON	JavaScript Object Notation
LL(1)	Left-to-right, Leftmost derivation, 1 token lookahead
LALR	Look-Ahead LR parser
LR	Left-to-right, Rightmost derivation
LSP	Language Server Protocol
NFA	Non-deterministic Finite Automaton
OOP	Object-Oriented Programming
PEG	Parsing Expression Grammar
SLR	Simple LR parser
XML	eXtensible Markup Language
XSLT	XML Stylesheet Language Transformations

Chapter 1
Introduction

The primary aim of this thesis is to provide a compiler frontend for the `tinyC` language that simplifies backend implementation for students in the NI-GEN course. By delivering a standardized frontend capable of generating a language-agnostic abstract syntax tree (AST) in a JSON format, students can focus exclusively on backend implementation, optimization techniques, and code generation in their preferred programming language.

1.1 Background and Motivation

The NI-GEN: Code Generators course at the Czech Technical University is designed to teach both theoretical and practical aspects of compiler backend development. According to the course materials, students are expected to "become acquainted with both theoretical and practical aspects of backend of an optimizing programming language compiler." Emphasizing hands-on experience, the course notes that "the beauty of compilers comes in part from the fact that here, the saying that 'the devil lies in the details' is more pronounced than in many other parts of CS."

To fulfill course requirements, students implement a compiler for a small language called `tinyC`, covering aspects such as:

- Syntax and semantic descriptions
- Parser
- Type checker
- Translation to LLVM IR
- Optimizations
- Code generation for the idealized t86 target

While this approach places considerable demands on students to develop both frontend and backend components within a single term. Given that the course predominantly emphasizes backend development, providing a standardized, high-quality frontend would greatly benefit students, allowing them to concentrate on backend-specific tasks such as optimization and code generation.

1.2 Problem Statement

The current structure of the NI-GEN course assignments, which requires students to implement both frontend and backend components, poses several challenges:

1. Students are provided with an existing parser written in C++, which is of poor quality and contains implementation errors. Consequently, students are forced not only to debug and improve this parser but also to implement the middle and backend in C++, a language in which many may lack sufficient proficiency.
2. Limited time within a single-term course restricts in-depth exploration of backend optimization techniques.
3. Implementing a parser and AST diverts focus from backend-centric learning objectives related to optimization and code generation.

Thus, there is a clear necessity for a standardized frontend implementation for the `tinyC` language, facilitating backend development independently of the programming language used.

1.3 Project Goals

This thesis addresses these challenges by developing a compiler frontend for the `tinyC` language. The key objectives are:

- Developing a lexer and parser that conform precisely to the defined `tinyC` grammar.
- Designing a well-structured and extensible AST.
- Providing a library interface suitable for direct integration primarily into C++ projects, along with a standalone executable outputting the AST in a standardized, language-agnostic JSON format.
- Including detailed source location information for improved debugging and error reporting.

- Ensuring testing, clear documentation, and maintainable implementation.

With these objectives, the project aims to streamline frontend tasks, enabling students to allocate more resources towards mastering backend compiler techniques.

1.4 Expected Contributions

The thesis will provide several significant contributions:

- An extensive analysis of existing parser technologies and language-agnostic AST formats.
- A robust, object-oriented AST design promoting easy traversal and modification.
- A standardized and language-agnostic JSON representation of the AST, enabling flexibility in backend implementation.
- A thoroughly tested lexer and parser implementation for `tinyC` in C++.
- Comprehensive documentation and examples to facilitate integration into students' projects.

1.5 Thesis Structure

The remaining chapters of this thesis are structured as follows:

- **Chapter 2:** *Theoretical Background* introduces fundamental compiler theory, particularly lexical analysis, parsing methods, and AST structures relevant to `tinyC`.
- **Chapter 3:** *Analysis of Existing Solutions* reviews current parser technologies and AST representations, and evaluates how well they meet the needs of this project.
- **Chapter 4:** *Design Requirements & Architecture* details the architecture of the `tinyC` compiler frontend, including AST class definitions, the visitor pattern, and the JSON format.
- **Chapter 5:** *Evaluation* compares the developed frontend with existing solutions, evaluating its effectiveness, usability, and overall improvements.
- **Chapter 6:** *Conclusion* summarizes the project's achievements and outlines possible future enhancements and extensions.

Chapter 2

Theoretical Background

This chapter presents the essential theoretical concepts that form the foundation for compiler frontend development. We begin with an overview of formal languages and grammars that provide the mathematical framework for defining programming languages. Next, we examine the compilation process, with particular focus on the frontend phases. We then explore lexical analysis (the process of converting source code into tokens) and syntax analysis (parsing tokens according to the language grammar). Finally, we discuss Abstract Syntax Trees (ASTs) as the key intermediate representation produced by the frontend for use in subsequent compiler phases. These concepts directly inform the design decisions made in our `tinyC` compiler frontend implementation.

2.1 Formal Languages

2.1.1 The Significance of Formal Language Theory

Formal language theory provides the mathematical tools and methodologies for specifying and implementing lexical analysers (via regular expressions and finite automata) and parsers (via context-free grammars and parsing techniques). It guides the selection of appropriate formalisms, ensuring that compiler design proceeds systematically and reliably.

▶ **Definiton 2.1.** ***Formal language*** *is a precisely defined set of strings composed from a finite alphabet, with their structure and validity governed by a specific set of rules known as a formal grammar [1].*

Unlike natural languages, which evolve organically and often contain ambiguities, formal languages are intentionally designed for specific applications, such as mathematics, chemistry, and, most importantly for our context, programming. The notation employed by mathematicians to express relationships between numbers and symbols, for instance, constitutes a formal language

adept at its intended purpose. Similarly, programming languages are formal languages meticulously crafted to express computations in an unambiguous manner [2].

The fundamental building blocks of a formal language include a finite set of symbols known as the alphabet, from which finite sequences (strings or words) are constructed, and a subset of these strings that constitutes the language itself. The syntax of a formal language determines which strings are well-formed according to a predefined set of production rules [1, 2]. The design of a programming language necessitates a precise syntax to guarantee each program has a unique interpretation by the compiler.

2.1.2 Types of Formal Languages

Formal languages can be categorised into different types based on the complexity of their defining grammars, often visualised through the Chomsky hierarchy illustrated in Figure 2.1. Within this hierarchy, regular languages and context-free languages hold particular significance for compiler frontends [1].

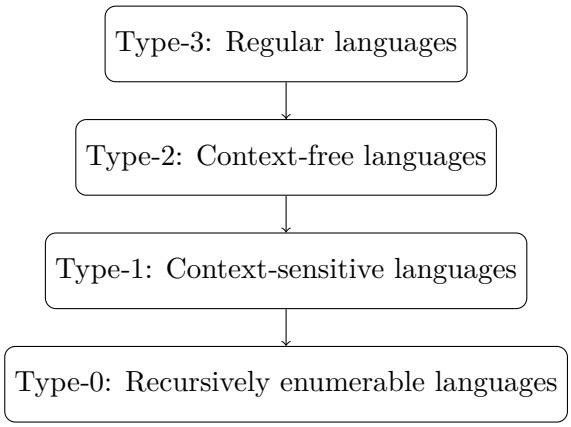


Figure 2.1 Chomsky hierarchy of formal languages

Regular Languages

Regular languages are the simplest in the hierarchy and can be defined using regular expressions, which offer a concise way to describe patterns of strings recognised by finite automata. In compilers, they are primarily used in lexical analysis to define tokens such as keywords, identifiers and operators [1].

Context-Free Languages (CFLs)

Context-free languages, defined by context-free grammars comprising production rules, are recognised by pushdown automata. The syntactic structure of programming languages—including hierarchical statements, expressions and

control constructs—is typically defined using CFGs, and parsing relies on these principles [3, 1].

While context-sensitive and recursively enumerable languages appear in the Chomsky hierarchy, their direct relevance to compiler frontends is limited. Regular languages suffice for token recognition, whereas the intricate syntactic structures of programming languages demand the expressive power of context-free languages.

2.1.3 Key Properties of Formal Languages

Understanding syntax, semantics and ambiguity is essential for comprehending how compilers process source code. Syntax governs token formation and program structure, semantics addresses the meaning and consistency checks during semantic analysis, and ambiguity—where a grammar permits multiple parse trees for the same string—is minimised in language design to ensure predictable compiler behaviour [1].

2.2 The Compilation Process

The process of compiling a program is typically structured as a sequence of distinct phases, each responsible for a specific transformation of the source code as it progresses towards becoming executable machine code. These phases generally include Lexical Analysis (Scanning), Syntax Analysis (Parsing), Semantic Analysis, Intermediate Code Generation, Code Optimisation, and Code Generation (see Table 2.1).

1. **Lexical Analysis (Scanning):** This initial phase reads the source code character by character and groups these characters into meaningful units called lexemes. For each lexeme, the lexical analyser produces a token, which represents a category of lexical units such as keywords, identifiers, operators, and literals [1]. It also typically removes whitespace and comments from the source code [1].

2. **Syntax Analysis (Parsing):** The parser takes the stream of tokens produced by the lexical analyser and checks if this sequence of tokens adheres to the grammatical rules of the programming language. If the syntax is correct, the parser constructs a parse tree or an Abstract Syntax Tree (AST) that represents the hierarchical structure of the program [1].

3. **Semantic Analysis:** This phase checks the program for semantic correctness, ensuring that the program makes sense according to the language's rules. This can involve type checking, verifying that variables are declared before use, and ensuring that operations are applied to compatible data types [1].

4. **Intermediate Code Generation:** After semantic analysis, the compiler may generate an intermediate representation of the program. This representation is often a low-level, machine-independent code that facilitates optimisation and code generation for various target architectures [1].

5. **Code Optimisation:** This optional phase aims to improve the intermediate code to make the program run faster or use fewer resources. Various optimisation techniques can be applied at this stage [1].

6. **Code Generation:** The final phase translates the optimised intermediate code into the target machine code or assembly language that can be executed by the computer [1]. This phase also involves tasks like register allocation and instruction scheduling.

The compilation process is often divided into a frontend, which primarily deals with the analysis of the source code (lexical, syntax, and semantic analysis), and a backend, which focuses on the synthesis of the target code (intermediate code generation, optimisation, and code generation) [1]. Throughout

these phases, a symbol table is maintained to store information about identifiers (such as variable names, function names) used in the program, including their type, scope, and memory location [1]. The modularity of the compilation process into these distinct phases allows for a structured approach to compiler design, where each phase can be developed and optimised with a degree of independence. Breaking down the complex task of translation into smaller, more manageable phases simplifies the overall development effort.

This thesis concentrates on the compiler frontend—namely lexical analysis and syntax analysis—as they form the theoretical basis for `tinyC`'s formal language processing. Lexical analysis tokenises the raw source into a sequence of symbols, while syntax analysis parses these tokens into a structured Abstract Syntax Tree (AST). Together, they ensure that source programs conform to the grammar of `tinyC`, providing the necessary structure for all subsequent compiler stages.

Phase Name	Input	Output	Key Tasks
Lexical Analysis (Scanning)	Source code	Stream of tokens	Reads source code, groups characters into lexemes, produces tokens, removes whitespace/comments, reports errors.
Syntax Analysis (Parsing)	Stream of tokens	Parse tree or AST	Verifies token sequence against grammar, constructs parse tree/AST, reports syntax errors.
Semantic Analysis	Parse tree or AST	Annotated AST	Checks type and scope rules, annotates AST with semantic information.
Intermediate Code Generation	Annotated AST	Intermediate representation (IR)	Emits machine-independent IR for optimisation and code generation.
Code Optimisation	IR	Optimised IR	Applies transformations to improve performance or resource usage.
Code Generation	Optimised IR	Target machine code	Translates IR to assembly/machine code, performs register allocation and scheduling.

Table 2.1 Phases of a Compiler and their key tasks

2.3 Lexical Analysis

The lexical analyser, often referred to as a scanner or tokenizer, is the first phase in the compilation process. Its primary role is to read the source code, which is essentially a stream of characters, and to group these characters into meaningful sequences known as lexemes [1].

2.3.1 Responsibilities

The lexical analyser is responsible for the following tasks:

1. Produce a token for each identified lexeme, categorising it as a keyword (e.g. `if`, `while`), identifier (e.g. variable names), operator (e.g. `=`, `+`), literal (e.g. numbers, strings), or punctuation (e.g. parentheses, semicolons) [1].
2. Remove whitespace (spaces, tabs, newlines) and comments, as they serve only to separate tokens and are not needed by the parser [1].
3. Detect and report lexical errors, such as invalid characters or character sequences that do not match any token pattern [1].
4. Store preliminary information about tokens—particularly identifiers—in the symbol table for later compiler phases [1].
5. Track the location (line and column numbers) of each token for precise error reporting and downstream stages.
6. Simplify the parser's input by transforming the raw source code into a structured stream of tokens, thereby abstracting away character-level details.

2.3.2 Tokens, Lexemes, and Patterns

To effectively understand the process of lexical analysis, it is crucial to distinguish between the concepts of tokens, lexemes, and patterns [1].

Tokens are abstract categories (e.g. `KEYWORD`, `IDENTIFIER`, `INTEGER`); lexemes are the actual character sequences (e.g. `if`, `count`, `123`); and patterns are the regular expressions that define which lexemes belong to which token class [1].

Token	Example lexemes	Pattern (regex)
`KEYWORD`	if, while, return	`if|while|return|...`
`IDENTIFIER`	count, tmp	`[A-Za-z][A-Za-z_0-9]*`
`INTEGER`	0, 123	`[0-9]+`

Table 2.2 Common token classes in `tinyC`

Patterns are expressed using operators for concatenation, alternation (`|`), and repetition (`*`, `+`, `?`). For instance, `[A-Za-z_][A-Za-z_0-9]*` defines identifiers, while `[0-9]+` defines integer literals. Tools such as Lex or Flex convert these specifications into effective scanners [1].

2.3.3 Specifying Tokens with Regular Expressions

Regular expressions allow concise specification of token patterns [1]. Key points include:

- Identifiers: `[A-Za-z_][A-Za-z_0-9]*` matches names starting with a letter or underscore.
- Integer literals: `[0-9]+` matches one or more digits.
- Keywords: fixed strings such as `if`, `while`, `return`.
- Operators and punctuation: single- or multi-character sequences (e.g. `=`, `++`, `;`).
- Construction operators: concatenation, alternation (`|`), repetition (`*`, `+`, `?`).

2.3.4 Implementing Lexical Analysers using Finite Automata

The theoretical equivalence between regular expressions and finite automata is fundamental to the implementation of lexical analysers [1]. Regular expressions, as discussed, are used to specify the patterns for tokens. These regular expressions can be systematically converted into Non-deterministic Finite Automata (NFAs) [1].

An NFA is a finite automaton where for each state and each input symbol, there can be zero, one, or more next states. NFAs can also have transitions labelled with ϵ (epsilon), which represent transitions that can occur without consuming any input symbol [1]. While NFAs are useful for specifying the lexical rules derived from regular expressions, they are often not the most efficient model for direct implementation. For implementation purposes, NFAs are typically converted into Deterministic Finite Automata (DFAs) using techniques such as the powerset construction [1].

A DFA is a finite automaton that has exactly one transition for each input symbol from each state, and it does not have any ϵ-transitions. This deterministic nature makes DFAs more straightforward and efficient to implement in a scanner.

A finite automaton (whether an NFA or a DFA) consists of a finite set of states, a set of input symbols (the alphabet), a transition function that defines

how the automaton moves from one state to another based on the input symbol, a start state, and a set of accepting states [1]. In the context of lexical analysis, the scanner operates by reading the input stream of characters, one at a time. Starting from the initial state of the DFA, the automaton transitions between states based on the input character. If, after reading a sequence of characters, the DFA ends up in an accepting state, it indicates that a token has been recognised. The type of the token is determined by the specific accepting state that was reached.

The implementation of a DFA often involves the use of a transition table, which is a two-dimensional array where one dimension represents the current state and the other represents the input symbol. The entry in the table at the intersection of a state and an input symbol specifies the next state to which the automaton should transition [1]. This table-driven approach provides an efficient mechanism for performing lexical analysis. The systematic conversion from regular expressions to NFAs and then to DFAs allows for an automated way to build the lexical analysis phase of a compiler, where token recognition is driven by well-defined state transitions based on the characters in the source code.

2.4 Syntax Analysis

Syntax analysis, commonly known as parsing, is the second phase of the compilation process. It operates on the token stream produced by the lexical analyser and fulfils the following objectives [1]:

1. **Syntax Verification:** Ensure the token sequence conforms to the language grammar.
2. **Structural Construction:** Build a hierarchical representation (parse tree or Abstract Syntax Tree).
3. **Error Reporting:** Detect and report syntactic errors in the source code.

The parser examines the relationships and order of tokens to confirm that the source code adheres to the defined syntax. A successful parse yields a structured representation, which is essential for subsequent phases such as semantic analysis and intermediate code generation.

2.4.1 Context-Free Grammars

Context-free grammars (CFGs) formally specify the syntax of programming languages [1]. A CFG consists of:

- **Terminal Symbols:** Basic tokens provided by the lexical analyser.
- **Non-terminal Symbols:** Higher-level constructs (e.g. expressions, statements).
- **Production Rules:** Definitions of how non-terminals expand into terminals and/or other non-terminals.
- **Start Symbol:** The root non-terminal representing a complete program.

Derivation begins at the start symbol and proceeds by applying production rules until the input token sequence is generated. Table 2.3 illustrates parse tree with its corresponding grammar.

2.4.2 Top-Down and Bottom-Up Parsing

Parsing techniques can be broadly classified into two categories: top-down and bottom-up approaches. As shown in Table 2.4, these methods differ significantly in their approach to constructing the parse tree.

2.4.3 Top-Down Parsing Techniques

Top-down parsing begins at the start symbol and works towards the input tokens, predicting productions to derive the input. Common methods include:

Recursive Descent Parsing

Each non-terminal is implemented as a recursive function. Functions attempt to match tokens and call other functions for nested constructs. Backtracking may be required if a prediction fails [1].

LL(1) Parsing

LL(1) parsers scan left-to-right, construct a leftmost derivation, and use one lookahead symbol. They rely on FIRST and FOLLOW sets and parsing tables to avoid backtracking [1].

2.4.4 Bottom-Up Parsing Techniques

Bottom-up parsing starts with tokens and reduces them to non-terminals, reconstructing a rightmost derivation in reverse. Key approach:

LR (Shift-Reduce) Parsing

LR parsers use ACTION and GOTO tables to decide between *shift* and *reduce* operations. Variants include SLR, LALR, and LR(1) [1].

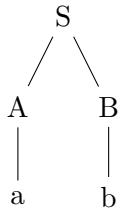
Explanation	**Parse Tree**
A CFG derivation can be visualised as a tree: the start symbol is at the root, and each application of a production rule adds children corresponding to the symbols on the right-hand side. For instance, given: $S \to AB, \quad A \to a, \quad B \to b,$ the parse tree captures exactly that structure, showing how S expands into A and B, and then into terminals a and b.	

Table 2.3 Parse tree for the grammar $S \to AB,\ A \to a,\ B \to b$.

Feature	Top-Down Parsing	Bottom-Up Parsing
Starting Point	Start symbol (root of parse tree)	Input tokens (leaves of parse tree)
Derivation Type	Leftmost derivation	Rightmost derivation in reverse
Handling of Left Recursion	Requires elimination of left recursion	Can handle left-recursive grammars
Common Techniques	Recursive Descent, LL(1) Parsing, Predictive Parsing	LR Parsing (SLR, LALR, LR), Shift-Reduce Parsing, Operator Precedence Parsing
Prediction of Production	Predicts the next production to apply	Postpones the decision until the entire right side is seen
Control	Explicitly constructs the parse tree from the top	Attempts to reduce the input to the start symbol

Table 2.4 Comparison of Top-Down and Bottom-Up Parsing

2.5 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a tree-like data structure representing the essential syntactic structure of a program's source code. It is constructed by the syntax analysis phase of a compiler, immediately after lexical analysis. The AST serves as a crucial intermediate representation for subsequent compiler phases: semantic analysis, intermediate code generation, and code optimisation. It abstracts away inessential concrete syntax details (e.g. parentheses, semicolons) to focus on core syntactic and semantic information [1].

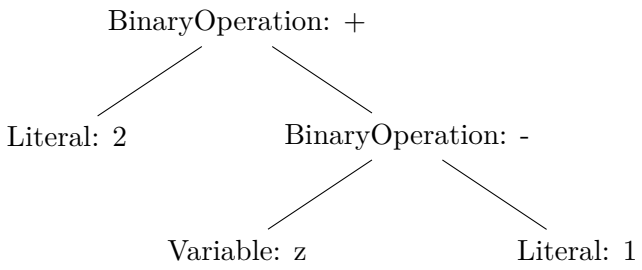


Figure 2.2 AST of the expression `2 + (z - 1)`

In an AST, each node represents a construct occurring in the source code [1]. These constructs can include operators (e.g. addition, subtraction), variables, control flow statements (e.g. if-then-else, loops), function calls, and literals. The children of a node represent its components or operands. For instance, in an AST for the expression `2 + (z - 1)`, as shown in Figure 2.2, the root node represents a binary addition operation, with two children: one for the literal value `2`, and one for a binary subtraction operation, which in turn has children representing the variable `z` and the literal value `1`. Each node stores information about its construct type and related data (e.g. variable names, literal values), encoding hierarchical relationships and operator precedence [1].

2.5.1 Abstraction from Concrete Syntax

The AST abstracts away inessential concrete syntax details present in a parse tree [1]. It omits grouping parentheses, statement-terminating semicolons, and parsing-only non-terminals, retaining only the constructs needed for semantic understanding. By eliminating syntactic sugar and focusing on semantic content, the AST provides a cleaner, more manageable structure for semantic analysis and code generation [1].

2.5.2 Further Compiler Stages

As an intermediate representation, the AST bridges initial code analysis and later compiler phases. It underpins semantic analysis—checking types, scopes,

and other constraints—by providing a traversable structure. The AST also forms the basis for generating machine-independent intermediate code and enables code optimisation techniques through its well-defined hierarchy. A carefully designed AST thus directly impacts the efficiency and correctness of generated target code [1].

2.6 Conclusion

This chapter has explored the fundamental theoretical concepts that underpin the frontend of a compiler. We have examined the principles of formal languages, which provide the mathematical framework for defining the syntax of programming languages. We have also discussed the compilation process as a sequence of phases, with a particular focus on lexical analysis, the process of breaking down the source code into tokens, and syntax analysis, which involves parsing these tokens according to the language's grammar. Finally, we delved into the design and importance of Abstract Syntax Trees, which serve as a crucial intermediate representation of the program's structure for subsequent compiler stages. These theoretical concepts are interconnected and collectively form the essential foundation for the design and implementation of efficient and reliable compiler frontends. The principles discussed here will be directly applicable and further elaborated upon in the subsequent chapters, where we will explore their application in the specific context of the `tinyC` compiler frontend.

Chapter 3

Analysis of Existing Solutions

The development of a compiler frontend for educational purposes requires careful consideration of existing technologies, approaches, and best practices. This chapter examines various parser technologies, AST representations, serialization formats, and educational compiler frontends, with the aim of informing the design decisions for the `tinyC` frontend implementation.

3.1 Parser Technologies

Numerous approaches to parsing have been developed over the decades, each with its own advantages and trade-offs. This section analyzes the most relevant parser technologies from the perspective of educational compiler development.

3.1.1 Handwritten Recursive Descent Parsers

Recursive descent parsers represent one of the most straightforward approaches to parsing, implementing the grammar rules directly as a set of mutually recursive functions [1]. Each function corresponds to a non-terminal in the grammar and is responsible for recognizing the corresponding syntactic construct.

For educational purposes, handwritten recursive descent parsers offer significant pedagogical value. Their directness creates a clear correspondence between grammar and code, making it easier for students to grasp the relationship between formal grammar specifications and implementation [4]. They provide developers with fine-grained control over error messages, enabling more student-friendly diagnostic information [5]. Additionally, their flexibility allows modifications to the grammar to be directly translated to code changes without requiring intermediary tools [6].

Despite these advantages, handwritten parsers present certain challenges. They tend to be labor-intensive, requiring considerable manual coding, particularly for complex languages [6]. A significant technical limitation is their

inability to handle left recursion directly, which necessitates grammar transformations [1]. From a maintenance perspective, changes to the grammar often require coordinated modifications across numerous functions, increasing the risk of inconsistencies [7].

3.1.2 Parser Generators

Parser generators automate the creation of parsers from grammar specifications. Tools like ANTLR [8], Bison, and Yacc have long been used in compiler construction courses.

These tools offer significant advantages for compiler development. They enhance productivity by generating parser code directly from grammar specifications, substantially reducing implementation time [9]. Parser generators stand on a solid formal basis, implementing well-studied parsing algorithms with proven mathematical properties and performance characteristics [1]. Additionally, many parser generators include grammar validation capabilities that check for potential issues such as ambiguities or conflicts, providing early feedback to developers before implementation begins [8].

Despite these benefits, parser generators present certain challenges in educational contexts. Students face a steeper learning curve as they must master both the grammar specification language and the integration patterns with the host language, adding cognitive load to the learning process [10]. An abstraction gap often emerges where the generated code obscures the connection between grammar and parsing logic, potentially hindering students' understanding of the underlying principles [6]. Furthermore, the default error messages produced by many parser generators tend to be cryptic and implementation-oriented rather than user-friendly, requiring significant additional work to make them accessible and helpful to students [11].

3.1.3 Parser Combinators

Parser combinators represent a functional approach to parsing, where basic parsers are combined using higher-order functions to create more complex parsers [12]. Libraries like Parsec for Haskell and parser-combinators for Scala exemplify this approach.

The composability of parser combinators constitutes one of their primary strengths, allowing simple parsers to be combined in modular ways to handle increasingly complex structures [13]. In statically typed languages, parser combinators benefit from type safety, ensuring that type errors in parser construction are identified at compile time rather than manifesting as runtime errors [14]. This approach also tends to produce readable code where the parsing logic closely resembles the grammar itself, enhancing maintainability and facilitating understanding of the relationship between grammar and implementation [15].

Despite these merits, parser combinators come with certain drawbacks. Performance issues may arise in naive implementations due to excessive backtracking, potentially leading to exponential time complexity in worst-case scenarios [13]. Educational applications face a particular challenge in that parser combinators typically require familiarity with functional programming concepts, creating a potential barrier for students without this background [10]. Furthermore, advanced error recovery techniques—critical for providing helpful feedback to users—can be particularly difficult to implement in combinator-based parsers, often requiring specialized extensions to the basic combinator framework [11].

3.1.4 PEG Parsers

Parsing Expression Grammars (PEGs) provide an alternative formalism to Context-Free Grammars, with ordered choice replacing the ambiguous choice of CFGs [16]. PEG parsers often use packrat parsing techniques to achieve linear time complexity.

PEG parsers offer several distinctive advantages in parser implementation. Their ordered choice operator fundamentally eliminates parsing ambiguities by definition, as the first successful match is always chosen, creating deterministic behavior even with complex grammar constructs [16]. Many PEG implementations combine lexical and syntactic analysis into a unified process, removing the traditional separation between scanner and parser phases and potentially simplifying the overall implementation [17]. Beyond the capabilities of traditional context-free grammar tools, PEGs possess expressive power that enables them to recognize some non-context-free languages, broadening the range of syntactic structures they can parse [18].

These advantages, however, are counterbalanced by notable challenges. The ordered choice operator, while eliminating ambiguity, can lead to subtle and sometimes unexpected parsing behavior when grammar rules interact in complex ways [11]. Although packrat parsing achieves efficient linear time complexity, this comes at the cost of potentially high memory requirements due to the memoization of intermediate results [19]. From a theoretical perspective, PEGs have a less established mathematical foundation compared to LR or LL parsing techniques, which have decades of theoretical analysis and optimization research behind them [18].

3.1.5 Comparative Analysis

Table 3.1 presents a comparison of these parser technologies across dimensions particularly relevant to educational compiler development.

For the `tinyC` frontend, considering its educational purpose and the need for high-quality error reporting, a recursive descent parser with predictive parsing techniques offers an appealing balance of clarity, control, and performance.

Table 3.1 Comparison of Parser Technologies for Educational Use

Criterion	Recursive Descent	Parser Generators	Parser Combinators	PEG Parsers
Implementation effort	High	Low	Medium	Low
Error reporting quality	Excellent	Poor to Medium	Medium	Medium
Educational clarity	High	Medium	High (if functional programming is known)	Medium
Grammar flexibility	Medium	Medium to High	High	High
Performance	Good	Excellent	Variable	Good to Excellent
Tool dependencies	None	Required	Library only	Library or Generator

3.2 AST Representations

The Abstract Syntax Tree (AST) serves as the central data structure in a compiler, bridging the parsing and semantic analysis phases. Various approaches to AST design and implementation exist, each with implications for educational compiler development.

3.2.1 Object-Oriented Hierarchies

Traditional object-oriented AST designs use class hierarchies, with a base node class and derived classes for specific language constructs [20]. This approach is common in compilers implemented in languages like C++, Java, and C#.

Object-oriented AST hierarchies provide intuitive modeling where the class structure naturally maps to the syntactic categories of the language, creating clear parallels between the grammar and implementation [21]. Their extensibility allows new node types to be added by extending the hierarchy without disturbing existing code, facilitating incremental development [20]. Furthermore, their strong typing ensures operations are applied to appropriate nodes, catching many errors at compile time rather than runtime [22].

Despite these strengths, object-oriented hierarchies exhibit certain limitations. Their inherent rigidity means that adding new operations across the

entire AST typically requires modifying every node class, violating the open-closed principle [20]. Implementation often demands substantial repetitive code for node constructors, accessors, and other boilerplate, increasing maintenance burden [21]. Additionally, correctly implementing the visitor pattern to address the operation extension problem requires careful attention to detail and introduces its own complexity [20].

3.2.2 Algebraic Data Types

Languages that support algebraic data types (ADTs) offer an alternative approach to AST representation [23]. This approach is common in functional languages like ML, Haskell, and increasingly in modern multi-paradigm languages.

ADTs enable notably concise AST definitions, often allowing the entire tree structure to be described in a fraction of the code required by object-oriented approaches [23]. Their support for exhaustive pattern matching facilitates AST processing, with compilers typically warning about unhandled cases, reducing the risk of overlooking edge cases [22]. The immutable nature of typical ADT implementations simplifies reasoning about AST transformations, eliminating concerns about unexpected state changes during processing [24].

The ADT approach, however, faces significant practical challenges. Many mainstream languages used in education lack native ADT support, limiting the applicability of this approach in diverse educational settings [23]. ADTs also suffer from the expression problem in reverse: while adding operations is straightforward, adding new node types requires modifying all functions that process the AST [25]. From a performance perspective, naive ADT implementations may incur higher memory overhead compared to optimized object-oriented designs due to boxing and indirection [21].

3.2.3 Visitor Pattern Applications

The Visitor pattern provides a way to separate algorithms from the objects they operate on, addressing some limitations of rigid class hierarchies [20]. It is commonly used for AST traversal and transformation.

The classic Visitor pattern defines a visitor interface with visit methods for each node type, enabling new operations to be added without modifying the node classes themselves. This approach maintains strong type checking while improving extensibility for operations [20]. The Acyclic Visitor variant reduces coupling between visitor and visitable classes, allowing for more selective implementation of visit methods for only relevant node types [26]. For situations requiring more flexibility, the Reflective Visitor uses runtime type information to dynamically dispatch based on actual types, reducing the need for explicit accept methods in each node class [27].

For educational compilers, the classic Visitor pattern offers an excellent balance of type safety and extensibility while introducing students to an important design pattern widely used in software engineering [20].

3.2.4 Location Tracking Approaches

Maintaining source location information is essential for meaningful error messages and debugging support. Several approaches to location tracking exist, each with different trade-offs.

The embedded location data approach includes location information directly in each AST node, ensuring location data is always available when processing nodes [21]. Alternative implementations use location maps that maintain a separate mapping from nodes to locations, reducing node size but requiring additional lookups [28]. More sophisticated span-based tracking systems track both start and end positions to represent ranges in the source code, providing more precise information for error highlighting [8].

The embedded approach, where each AST node contains its source location, offers the most straightforward implementation and is well-suited for educational purposes [21]. This approach ensures that location information is always available when processing a node, simplifying error reporting and debugging for students implementing compiler backends.

3.3 AST Serialization Formats

For a compiler frontend to be useful across different backend implementations and languages, a well-defined serialization format is essential. This section analyzes various approaches to AST serialization, with a focus on interoperability and readability.

3.3.1 JSON-based Serialization

JSON has become a popular format for AST serialization due to its ubiquitous support across programming languages and human readability [29]. Notable examples include ESTree for JavaScript ASTs used by tools like Babel and ESLint [30], Microsoft's TypeScript compiler with its JSON serialization capabilities [31], and Clang's AST which can be dumped in JSON format, though with considerable verbosity [32].

The language independence of JSON creates significant advantages for cross-language compiler development, with parsers available for virtually all programming languages, facilitating backend implementations in students' preferred languages [29]. Its text-based format maintains human readability, making the AST structure relatively easy to understand and debug—an important consideration for educational tools [29]. For validation purposes, JSON Schema provides a standardized way to define and validate AST structures, ensuring conformance to expected patterns [33].

Despite these benefits, JSON serialization faces certain limitations. Its representation tends toward verbosity, with JSON outputs significantly larger than equivalent binary formats, potentially affecting performance for very large ASTs [34]. The limited native data type system in JSON requires encoding of specialized types like symbol tables or type information through conventions [29]. Additionally, JSON lacks standardized reference mechanisms, requiring custom conventions to represent cross-references within the AST structure [34].

3.3.2 XML Representations

XML was once the dominant format for serializing complex data structures, including ASTs [35]. While less common today, XML-based AST representations continue to offer unique capabilities for specialized applications.

The rich schema validation features of XML Schema and DTD provide strong validation capabilities beyond what JSON Schema typically offers, enabling precise structural enforcement [35]. For transformation purposes, XSLT provides a specialized language for sophisticated transformations of the AST, enabling complex operations without custom code [36]. The XPath query language integrated with XML offers a powerful mechanism to select and manipulate specific parts of the AST without traversing the entire structure [37].

These advantages, however, are overshadowed by significant drawbacks in modern compiler implementations. XML's extreme verbosity, with extensive markup overhead, dramatically increases the size of the AST representation compared to alternatives [35]. The parsing overhead for XML is generally higher than for JSON, affecting performance particularly in web or resource-constrained environments [38]. From an educational perspective, XML technologies present a steeper learning curve compared to JSON, potentially distracting from the core compiler concepts being taught [38].

3.3.3 Binary Serialization

For performance-critical applications, binary serialization formats offer significant space and time efficiency advantages [39]. Relevant formats include Protocol Buffers from Google [40], MessagePack offering a binary form of JSON with extended data types [41], and CBOR (Concise Binary Object Representation) designed specifically for small code and message sizes [42].

Binary formats achieve remarkable compactness with significantly smaller representations compared to text formats, important for large ASTs or resource-constrained environments [39]. Their parsing efficiency substantially outperforms text-based formats, reducing processing overhead during deserialization [39]. Many binary formats support richer type systems than JSON, including direct representation of advanced data structures, binary data, and language-specific types [40].

Despite these performance advantages, binary formats present substantial drawbacks for educational compiler implementations. Their fundamental human unreadability makes direct inspection and debugging difficult, requiring specialized tools to examine the serialized data [39]. These tool dependencies create additional requirements for students, who need special software to inspect and manipulate the ASTs [40]. Schema evolution—handling changes to the data format as the language evolves—often becomes more complex with binary formats, requiring careful versioning strategies [40].

3.3.4 Language Server Protocol

The Language Server Protocol (LSP) has established conventions for representing code structures for IDE integration [43]. While not primarily an AST serialization format, LSP's approaches inform modern compiler design and increasingly influence how ASTs are represented for tooling integration.

LSP employs position-based representation using zero-based line and character offsets to represent source positions, establishing a standard that many development tools now expect [43]. Its support for incremental updates allows efficient partial modifications to code structures without reprocessing entire files, important for responsive IDE experiences [43]. The protocol also defines

standardized diagnostic formats for reporting errors and warnings, creating consistency across language implementations [43].

These conventions are increasingly relevant for educational compiler design, as modern development environments increasingly expect LSP-compatible integration, and students may wish to build IDE support around their compiler implementations [43].

Table 3.2 Comparison of AST Serialization Formats

Characteristic	JSON	XML	Binary Formats
Human readability	High	Medium	None
Size efficiency	Low	Very Low	High
Parsing speed	Medium	Slow	Fast
Language independence	Excellent	Excellent	Good
Schema validation	Good	Excellent	Good
Tool support	Ubiquitous	Extensive	Framework-specific
Educational suitability	Excellent	Good	Limited

3.3.5 Schema Design Considerations

When designing an AST serialization schema, several key considerations emerge that shape the effectiveness of the representation. Node identification mechanisms determine how to uniquely identify and reference nodes within the AST, critical for cross-references and symbol resolution [4]. Location representation strategies determine how source positions are encoded in a language-agnostic way that supports precise error reporting [8]. Type system encoding approaches define how the language's type system is represented in the serialized format, necessary for type checking and semantic analysis [21]. Annotation mechanisms determine how additional metadata such as type information or compiler hints can be attached to nodes [21]. Finally, versioning strategies define how schema evolution is handled as the language and compiler evolve, ensuring backward compatibility [40].

For the `tinyC` frontend, JSON offers an appropriate balance of human readability, language independence, and schema validation capabilities. A well-designed JSON schema with clear node identification and comprehensive location tracking will support both educational objectives and practical backend integration.

3.4 Educational Compiler Frontends

Several compiler projects have been designed specifically for educational purposes. Analyzing their approaches provides valuable insights for the `tinyC` frontend design.

3.4.1 MiniJava Implementations

MiniJava is a subset of Java designed for teaching compiler construction [44]. Several implementations exist, offering different educational approaches. The Tiger Book implementation provides a reference implementation using ML, emphasizing functional programming techniques and type-directed compilation [44]. The jmm project offers a Java-based implementation that highlights object-oriented design principles and how they apply to compiler architecture [45]. For incremental learning, the MiniJava-compiler-construction project is specifically designed to be extended by students in stages, allowing them to build up their understanding progressively [45].

These varied implementations collectively demonstrate the importance of several architectural principles in educational compilers. Clear separation of concerns through well-defined interfaces between compiler phases allows students to focus on one concept at a time without being overwhelmed by the entire system [44]. An incremental learning path structured to allow students to build the compiler in logical stages facilitates understanding complex topics through progressive mastery [45]. The use of consistent design patterns throughout the codebase reinforces software engineering concepts alongside compiler theory, enhancing the educational value [44].

3.4.2 LLVM-based Educational Projects

LLVM has become a popular backend for educational compiler projects due to its modular design and extensive optimization capabilities [46]. The Kaleidoscope tutorial offered by the LLVM project itself provides a step-by-step guide to building a simple language frontend that targets the LLVM infrastructure [46]. Academic language implementations like the Classroom Object-Oriented Language (COOL) compiler have been adapted to use LLVM as their code generation target, providing students with powerful optimization capabilities [47]. More specialized projects such as the CMSC430 compiler for a subset of Racket demonstrate how functional language features can be implemented using LLVM [48].

These projects effectively illustrate key principles in modern compiler design. Their backend independence demonstrates the value of cleanly separating the frontend from code generation concerns, allowing students to focus on language semantics without worrying about target architecture details [46]. The

use of well-defined intermediate representations as interfaces between compiler phases creates clear boundaries for student implementations, making the projects more manageable [46]. Their instructional design typically follows a pattern of progressive complexity, starting with simple language constructs and gradually adding features, helping students build confidence through incremental success [47].

Table 3.3 Comparison of Educational Compiler Projects

Feature	MiniJava	LLVM Kaleidoscope	COOL
Implementation language	ML/Java	C++	C++/Python
Source language paradigm	OOP	Functional	OOP
Error reporting quality	High	Basic	High
Documentation	Excellent	Excellent	Good
Backend flexibility	Limited	Excellent	Good
Incremental learning path	Strong	Strong	Medium
Type system complexity	Medium	Low	High

3.4.3 Error Reporting Quality

The quality of error reporting significantly impacts the educational value of a compiler frontend [49]. Analysis of educational compilers reveals several approaches to effective error communication. Advanced implementations incorporate error recovery mechanisms that continue parsing after encountering errors, allowing them to report multiple issues in a single pass rather than stopping at the first problem [1]. The most helpful systems provide contextual error messages that offer suggestions based on the specific parsing context, helping students understand not just what went wrong but how to fix it [49]. Visual approaches using source highlighting to indicate error locations directly in the code provide immediate spatial context for understanding the problem's location [50].

The most educationally effective compiler implementations share several error reporting characteristics. They excel at precise error localization, pinpointing the exact location of errors down to the specific token or character, eliminating guesswork for students [50]. Their explanatory messages go beyond

simply stating that an error occurred to clearly explain what went wrong in language accessible to students still learning the concepts [49]. Many advanced educational compilers also include suggested corrections that hint at how to fix the error, serving both as immediate help and as a learning tool [49]. Table 3.3 includes error reporting quality as one of the key comparison dimensions across educational compiler projects.

3.4.4 Documentation Standards

Effective documentation is crucial for educational compiler projects [51]. Analysis of successful projects reveals several documentation patterns that enhance their teaching value. Architectural overviews provide high-level explanations of the compiler's structure and phases, helping students understand the big picture before diving into details [51]. Detailed API documentation clearly specifies interfaces between components, critical for students building on existing code or implementing new modules [51]. Tutorial-style guides offer step-by-step explanations of how to use and extend the compiler, supporting active learning through guided practice [45]. Implementation notes explaining key algorithms and design decisions provide insight into the reasoning behind the code, helping students develop their own design thinking [44].

The most effective educational compiler documentation addresses multiple distinct audiences with appropriate content for each. For students using the compiler, clear guides on how to write programs in the target language facilitate immediate engagement with the system [51]. Students extending the compiler benefit from tutorials on adding features or optimizations, providing structured pathways for deeper exploration [45]. Instructors require materials that help integrate the compiler into coursework, including sample assignments and evaluation criteria [51].

3.4.5 Integration Flexibility

Educational compiler frontends must be designed for flexible integration with various backend components to maximize their utility across different teaching contexts [52]. Successful approaches to achieving this flexibility share several architectural characteristics. Clean API boundaries with well-defined interfaces hide implementation details while providing reliable connection points for student-developed components [52]. Support for multiple output formats enables compatibility with different backend frameworks and implementation languages, expanding the range of assignments possible [46]. Thoughtfully placed extensibility hooks provide designated points where students can add functionality without needing to understand the entire system, lowering the barrier to entry for initial assignments [45].

For the `tinyC` frontend, the dual approach of providing both a C++ library and a standalone executable with JSON output maximizes integration flexibil-

ity. This design allows students to choose the approach that best fits their implementation language and preferences, accommodating diverse learning styles and technical backgrounds while maintaining a consistent representation of the language semantics.

3.5 Summary of Design Implications

The analysis of existing solutions leads to several design implications for the `tinyC` frontend. Based on the comparative analysis presented in provided tables, as well as the review of educational compiler projects in Table 3.3, specific approaches emerge as most appropriate for this educational context.

A handwritten recursive descent parser with LL(1) predictive parsing offers the best balance of clarity, control, and educational value, providing students with a straightforward implementation that clearly demonstrates parsing principles. For AST representation, an object-oriented hierarchy with the visitor pattern provides a clear structure while teaching an important design pattern, offering a good compromise between type safety and extensibility.

JSON with a well-defined schema emerges as the optimal serialization format, offering the best combination of human readability and language independence while providing sufficient validation capabilities. To maximize educational value, source location tracking and contextual error messages are essential, helping students locate and understand issues in their code. Clear architectural documentation, including API specifications, should accompany the implementation to facilitate both use and extension. Finally, the dual library/executable approach maximizes flexibility for various backend implementations, accommodating diverse student preferences and technical backgrounds.

These insights have directly informed the design requirements and architecture of the `tinyC` frontend, as detailed in the following chapter.

Chapter 4

Design Requirements & Architecture

This chapter outlines the design requirements and architecture of the `tinyC` compiler frontend. First, it defines the functional and non-functional requirements that guide the design decisions. Then, it analyses the `tinyC` grammar to understand its characteristics and challenges. Finally, it describes the system architecture, including the AST design, API specifications, and test suite architecture.

4.1 Functional Requirements

The primary purpose of the `tinyC` compiler frontend is to provide students with a reliable tool for parsing `tinyC` source code, allowing them to focus on compiler middle-end and back-end development in the language of their choice. The functional requirements define the essential capabilities the system must provide to fulfill this purpose.

4.1.1 Core Parsing Functionality

At its heart, the frontend must perform accurate lexical analysis by tokenizing `tinyC` source code according to the language specification. This process involves correctly identifying all token types—keywords, identifiers, literals, operators, and punctuation—while properly managing whitespace and comments. The frontend then conducts syntax analysis, parsing the token stream according to the `tinyC` grammar rules to verify syntactic correctness.

Upon successful parsing, the system constructs a well-structured abstract syntax tree (AST) that accurately represents the parsed program's hierarchical structure. This AST serves as the primary interface between the frontend and any student-implemented backend. To facilitate error reporting and de-

bugging throughout the compilation process, each node in the AST includes precise source location information, including filename, line number, and column number.

4.1.2 Error Handling

Effective error handling constitutes a critical aspect of the frontend's functionality. The system detects and reports lexical errors (such as invalid characters or malformed tokens), and syntactic errors (such as grammar violations). More importantly than merely detecting these errors, the frontend provides clear, informative error messages that include relevant source location information. These messages guide students toward understanding and resolving issues in their code, enhancing the educational value of the tool.

4.1.3 Interface Options

The frontend offers two complementary interface options to accommodate different student preferences and project requirements. First, a C++ library interface provides direct access to AST classes and parsing functionality. This option is ideal for students who prefer working in C++ and wish to tightly integrate the frontend with their backend implementation.

Second, a command-line interface provides a standalone executable that accepts `tinyC` source files and outputs the AST in JSON format. This interface option enables students to work in any programming language of their choice, as they can parse the standardized JSON output using libraries available in virtually any language.

The JSON output follows a well-defined schema that includes all necessary information from the AST, including node types, hierarchical relationships, and source locations. This schema ensures consistency and interoperability regardless of which backend implementation approach a student chooses.

4.2 Non-functional Requirements

While functional requirements define what the system must do, non-functional requirements define the qualities that determine how the system should behave. These requirements are crucial for ensuring the frontend's suitability for its educational context and for providing a positive experience for students using it in their coursework.

4.2.1 Performance

The frontend must perform efficiently enough to handle typical `tinyC` programs without introducing frustrating delays in the development workflow.

Parsing time should scale proportionally with input size, maintaining reasonable performance even for larger programs.

Memory usage should remain modest, avoiding unnecessary duplication of data and implementing appropriate data structures for the AST nodes. Since the frontend will be used primarily in an educational setting, absolute performance optimization is less critical than clarity and correctness, but the implementation should still follow reasonable efficiency principles.

4.2.2 Usability

For the frontend to serve its educational purpose effectively, it must be straightforward to integrate into student projects. This requires minimal external dependencies, clear documentation, and intuitive interfaces. The API, JSON format, and usage examples need documentation that students can reference without extensive prior knowledge of compiler construction. Installation and setup should be simple, with clear instructions for different operating systems.

Platform independence is essential, as students in the course may use various development environments across Windows, macOS, and Linux. The frontend should function consistently across these platforms, avoiding system-specific features that might create barriers for some students. For the command-line interface, the output format should be consistent regardless of the operating system, ensuring that student backends can rely on a standardized input format.

4.2.3 Maintainability

The implementation must be clean, well-structured, and follow modern C++ best practices to facilitate future maintenance and extension. A modular design separates concerns like lexical analysis, syntax analysis, and AST construction, making each component easier to understand and modify independently. The JSON output format and AST structure are designed with extensibility in mind, allowing for potential future extensions to the `tinyC` language without requiring a complete redesign.

The codebase requires unit tests for both the lexer and parser components to verify their correctness independently. These unit tests should cover a wide range of input cases, including edge cases and error conditions, ensuring that each component behaves as expected in isolation before being integrated. This approach facilitates a more maintainable codebase where changes to one component do not inadvertently affect others.

The codebase should be maintained in a version control system with a clear commit history, enabling instructors to track changes and understand the evolution of the implementation. This approach also supports potential contributions from teaching assistants or even students in future course iterations.

4.2.4 Educational Value

Given its pedagogical context, the frontend must serve not only as a functional tool but also as a learning resource. The code should be readable and instructive, serving as a good example for students learning about compiler construction. Implementation choices should balance theoretical correctness with practical considerations, demonstrating sound software engineering principles alongside compiler theory.

Beyond unit tests, the system requires a language test suite that verifies the frontend's behavior against a collection of `tinyC` programs. This test suite serves multiple purposes: it validates the correctness of the provided implementation, gives students a tool to verify their own implementations, and demonstrates the expected behavior of the language across various scenarios. The test suite should include specially formatted `tinyC` source files with metadata specifying expected outcomes, enabling automated validation of parser behavior.

The test suite should support incremental development, allowing students to focus on specific language features one at a time. It should provide clear feedback about why tests fail, helping students identify and fix issues in their implementations. Additionally, the test suite should cover both successful parsing cases and error conditions, ensuring that parsers correctly handle invalid input.

4.3 `tinyC` Grammar Analysis

The `tinyC` language is a simplified subset of C designed for educational purposes. Understanding its grammar characteristics and implementation challenges is essential for designing an effective parser that serves the course's pedagogical goals.

The `tinyC` language encompasses a carefully selected subset of C language features, providing enough complexity to be educational while remaining manageable for a course project. It supports basic types including integer, double, character, and void types, along with variables, arrays, and pointers. Function declarations and definitions follow C-like syntax, and the language includes standard control structures—if-else conditionals, while and do-while loops, for loops, and switch-case statements. Expression syntax maintains C-like operator precedence, and the language supports struct definitions for creating custom data types.

From a formal language perspective, the `tinyC` grammar can be generally classified as LL(1), making it suitable for implementation using predictive recursive descent parsing techniques. This classification is particularly advantageous in an educational context, as recursive descent parsers closely mirror the grammar's structure, making the relationship between theory and implementation more transparent to students. To verify the LL(1) properties of the

grammar and to assist with implementation, I used a parsing table generator tool developed by Ing. Tomáš Pecka (available at `pages.fit.cvut.cz/peckato1/parsingtbl`). This tool proved invaluable for identifying and resolving grammar conflicts, as well as for generating templates for recursive descent parsing functions.

Several aspects of the `tinyC` grammar presented challenges for implementing a pure LL(1) parser. Expression parsing required careful factoring to eliminate left recursion while preserving operator precedence. Type declarations, especially pointer types and function pointers, needed special handling to account for their complex syntax. Statements with optional components like the for-loop's initialization, condition, and update expressions required nullable productions and appropriate lookahead testing. Additionally, the original grammar contained instances of left recursion and common prefixes that needed systematic elimination through standard grammar transformations to ensure LL(1) compatibility.

The final grammar used for implementation is provided in Appendix A. It represents a factored form of the `tinyC` grammar that is suitable for LL(1) parsing while remaining faithful to the language specification. While the transformations introduce some additional non-terminals and productions compared to the original grammar, they preserve the language's semantics and ensure efficient, predictable parsing behavior.

4.4 System Architecture

The `tinyC` compiler frontend is designed with a modular architecture that separates concerns and facilitates both maintenance and extension. Figure 4.1 illustrates the high-level architecture of the system.

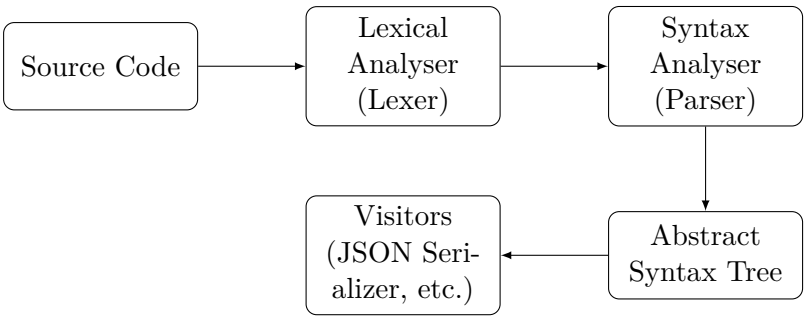


Figure 4.1 High-level architecture of the `tinyC` compiler frontend

The architecture consists of the following main components:

1. **Lexical Analyser (Lexer)**: The lexer reads the source code character by character and groups them into tokens according to the lexical rules of `tinyC`. It handles whitespace, comments, and reports lexical errors.

2. **Syntax Analyser (Parser)**: The parser takes the token stream produced by the lexer and verifies that it conforms to the `tinyC` grammar. It constructs an abstract syntax tree and reports syntax errors.

3. **Abstract Syntax Tree (AST)**: The AST represents the hierarchical structure of the parsed program. Each node in the tree corresponds to a language construct in the source code.

4. **Visitors**: The visitor pattern is implemented to traverse the AST for various operations. The most important visitor is the JSON serializer, which converts the AST to a standardized JSON representation. Other potential visitors could include a pretty-printer, static analyzer, or code generator.

The core implementation is structured as a library (lib`tinyC`) that can be used directly in C++ projects. A separate command-line interface (`tinyC`-compiler) is built on top of this library, providing the standalone executable functionality.

4.5 AST Design

The AST design follows object-oriented principles to create a clear, maintainable, and extensible representation of `tinyC` programs. The design prioritizes simplicity, allowing students to easily work with the AST representation regardless of their implementation language.

4.5.1 Node Structure

The AST consists of various node types that represent different language constructs in `tinyC`. Rather than implementing a monolithic node class with numerous type-specific fields, the design employs a class hierarchy with a common base class (`ASTNode`) and specialized derived classes for different language elements. This approach follows the Single Responsibility Principle, with each class representing exactly one language construct.

Key node categories include declaration nodes for variables, functions, and structs; type nodes representing primitive types, named types, and pointer types; expression nodes for literals, identifiers, and operations; and statement nodes for blocks, conditionals, loops, and other control structures. Each node

contains specific fields relevant to its language construct and provides access methods that expose these fields while maintaining appropriate encapsulation.

The base `ASTNode` class establishes common functionality across all nodes, including source location tracking for error reporting. Each derived class extends this base with construct-specific fields and behavior. This inheritance hierarchy allows for both static type safety through C++'s type system and runtime type identification through virtual functions and the visitor pattern.

4.5.2 Memory Management

The AST uses `std::unique_ptr` for managing node ownership, a choice that naturally aligns with the hierarchical structure of abstract syntax trees. In an AST, each node has exactly one parent (except the root), creating a clear ownership hierarchy that unique pointers enforce at compile time. This approach offers several advantages over alternatives like `std::shared_ptr`.

Unique pointers eliminate the reference counting overhead of shared pointers—an unnecessary cost for tree structures with clear parent-child relationships. They also prevent circular reference problems that could arise with shared pointers if the AST implementation needed cross-references between nodes. When using unique pointers, such relationships must be implemented as non-owning raw pointers, making ownership direction explicit and preventing accidental memory leaks.

From a design perspective, unique pointers make ownership transfer explicit through `std::move`, clearly indicating when a node becomes part of the parent structure. This explicitness improves code readability—particularly valuable in an educational context—while ensuring deterministic destruction that follows the tree hierarchy when nodes are deleted. This approach balances memory safety with performance considerations while maintaining a clear, understandable implementation.

4.5.3 Visitor Pattern Implementation

The visitor pattern provides a mechanism for performing operations on the AST without modifying the node classes, adhering to the Open-Closed Principle. This design pattern is particularly valuable in compiler construction, where different passes (e.g., semantic analysis, optimization, code generation) need to traverse the same AST structure while performing different operations.

The base `NodeVisitor` interface declares virtual `visit` methods for each concrete node type, as shown in Listing 4.1:

Each AST node implements an `accept` method that invokes the appropriate `visit` method on the visitor, as demonstrated in Listing 4.2:

This design allows for adding new operations on the AST without modifying the node classes, adhering to the Open-Closed Principle as discussed in the visitor pattern implementation.

```
class NodeVisitor {
public:
    virtual ~NodeVisitor() = default;

    // Visit methods for declaration nodes
    virtual void visit(const VariableNode& node) = 0;
    virtual void visit(const FunctionDeclarationNode& node) = 0;
    // ... other declarations

    // Visit methods for type nodes
    virtual void visit(const PrimitiveTypeNode& node) = 0;
    // ... other types

    // Visit methods for expression nodes
    virtual void visit(const LiteralNode& node) = 0;
    // ... other expressions

    // Visit methods for statement nodes
    virtual void visit(const BlockStatementNode& node) = 0;
    // ... other statements
};
```

Code listing 4.1 NodeVisitor interface declaration

```
void BlockStatementNode::accept(NodeVisitor& visitor) const {
    visitor.visit(*this);
}
```

Code listing 4.2 BlockStatementNode's accept method

4.5.4 Source Location Tracking

Each AST node includes source location information to facilitate error reporting and debugging, as shown in Listing 4.3:

The `SourceLocation` struct contains the filename, line number (1-based), and column number (1-based). This information originates in the lexer, which tracks position as it tokenizes the source code. The parser then passes this location information to created AST nodes, ensuring that every node can be traced back to its origin in the source code.

This location tracking is particularly valuable for error reporting during later compilation phases. When a semantic error is detected (e.g., type mismatch or undeclared variable), the compiler can provide precise information about where the error occurred, greatly assisting students in debugging their `tinyC` programs.

```
class ASTNode {
public:
    explicit ASTNode(lexer::SourceLocation location);
    virtual ~ASTNode() = default;

    [[nodiscard]] lexer::SourceLocation getLocation() const;

    virtual void accept(NodeVisitor& visitor) const = 0;

private:
    const lexer::SourceLocation location;
};
```

Code listing 4.3 ASTNode base class declaration

4.6 API Design

The `tinyC` compiler frontend provides two main interfaces: a C++ library API for direct integration into student projects and a command-line interface for language-agnostic usage. This dual approach maximizes flexibility, accommodating students with varying programming language preferences and experience levels.

4.6.1 Library API

The library API allows direct integration of the `tinyC` frontend into C++ projects. This approach is ideal for students who prefer working in C++ and want tight integration between the frontend and their backend implementation. The API is designed with simplicity and clarity in mind, providing intuitive access to the frontend's core functionality. The main entry point for lexical analysis is the `Lexer` class, shown in Listing 4.4:

```
namespace tinyC::lexer {
    class Lexer {
    public:
        explicit Lexer(std::string source,
                       std::string filename = "<input>");
        TokenPtr nextToken();
        std::vector<TokenPtr> tokenize();
    };
}
```

Code listing 4.4 Lexer class interface

The `Lexer` constructor accepts the source code as a string, along with an optional filename for error reporting. The `nextToken` method returns the next token in the input stream, while the `tokenize` method processes the entire input and returns a vector of all tokens. This design allows for both incremental token-by-token processing and batch processing of the entire input.

For syntax analysis, the `Parser` class provides the entry point, as shown in Listing 4.5:

```
namespace tinyC::parser {
    class Parser {
    public:
        explicit Parser(lexer::Lexer& lexer);
        ast::ASTNodePtr parseProgram();
    };
}
```

Code listing 4.5 Parser class interface

The `Parser` constructor takes a reference to a `Lexer` instance, establishing the token source. The `parseProgram` method initiates parsing of the entire program, returning the root node of the resulting AST. This approach decouples the lexer and parser, allowing for separate testing and potential reuse of the lexer with different parsers.

For AST traversal and processing, the library provides visitor classes that implement the `NodeVisitor` interface, as shown in Listing 4.6:

```
namespace tinyC::ast {
    class DumpVisitor : public NodeVisitor {
    public:
        explicit DumpVisitor(std::ostream& os);
        // Visit methods for all node types
    };

    class JSONVisitor : public NodeVisitor {
    public:
        explicit JSONVisitor(bool prettyPrint = true);
        std::string getJSON() const;
        // Visit methods for all node types
    };
}
```

Code listing 4.6 Visitor classes for AST traversal

The `DumpVisitor` outputs a human-readable representation of the AST to the specified output stream, useful for debugging and understanding the parsed

structure. The `JSONVisitor` converts the AST to its JSON representation, with control over formatting through the `prettyPrint` option. The resulting JSON can be retrieved as a string using the `getJSON` method.

This API design prioritizes flexibility and ease of use. Students can work directly with the AST classes, implement their own visitors for custom operations, and control the parsing process step by step. The clear separation of concerns between lexical analysis, syntax analysis, and AST processing provides a solid foundation for understanding compiler structure.

4.6.2 Command-Line Interface

The command-line interface provides a standalone executable for parsing `tinyC` source files and outputting the AST as JSON. The main functionalities include:

1. **Lexical Analysis Mode**: The command `tinyC-compiler --lex file.tc` tokenizes the input file and outputs the tokens. This mode is useful for debugging lexical issues and understanding how the source code is tokenized.

2. **Parsing Mode**: The command `tinyC-compiler --parse file.tc` parses the input file and outputs the AST as JSON. This is the primary mode for students implementing their own backends, as it provides the complete AST structure in a standardized format.

3. **Interactive Mode**: Running `tinyC-compiler` without arguments launches an interactive shell where students can enter `tinyC` code and immediately see the resulting tokens or AST. This mode is particularly useful for learning and experimentation.

The command-line interface is designed to be simple and intuitive, with clear error messages and help text, making it accessible to students with varying levels of experience.

4.6.3 JSON Output Format

The JSON output format provides a standardized representation of the AST that can be consumed by any programming language. This format is the key to enabling language-agnostic backend implementation, as it decouples the frontend from any specific programming language or framework.

The JSON representation follows a well-defined schema with these key characteristics:

1. **Node Type Identification**: Each node includes a `nodeType` field that identifies its type, allowing backends to process nodes according to their specific language construct.

2. **Hierarchical Structure**: The JSON structure mirrors the hierarchical structure of the AST, with parent nodes containing their children as nested objects or arrays.

3. **Source Locations**: Each node includes location information with filename, line, and column, enabling precise error reporting in later compilation phases.

4. **Type-Specific Fields**: Each node type includes fields specific to that language construct, providing all necessary information for backend processing.

An example JSON output for a simple variable declaration illustrating these characteristics is shown in Listing 4.7.

This example demonstrates how the JSON format captures the hierarchical structure of the AST, with the program containing a variable declaration that includes a type and an initializer expression. Each node includes its type and location information, providing a complete representation of the parsed program.

```
{
  "nodeType": "Program",
  "declarations": [
    {
      "nodeType": "VariableDeclaration",
      "identifier": "x",
      "type": {
        "nodeType": "PrimitiveType",
        "kind": "int",
        "location": {
          "filename": "example.tc",
          "line": 1,
          "column": 1
        }
      },
      "initializer": {
        "nodeType": "Literal",
        "kind": "integer",
        "value": "42",
        "location": {
          "filename": "example.tc",
          "line": 1,
          "column": 7
        }
      },
      "location": {
        "filename": "example.tc",
        "line": 1,
        "column": 5
      }
    }
  ],
  "location": {
    "filename": "example.tc",
    "line": 1,
    "column": 1
  }
}
```

Code listing 4.7 Example JSON output for a variable declaration: `int x = 42;`

4.7 Test Suite Architecture

The test suite for the `tinyC` compiler frontend is a comprehensive framework designed to validate the correctness of both the provided implementation and student-created alternatives. It offers systematic verification of lexical analysis, parsing functionality, and error handling capabilities.

4.7.1 Test File Structure and Format

The test files follow a metadata-driven format that allows precise specification of expected outcomes. Each test file contains:

```
// tinyC TEST
// INFO: description of the test's purpose
// RUN: test type specification (parser, exec)
// EXPECT: expected outcome (SUCCESS, PARSER_ERROR, or LEXER_ERROR)
// RESULT: expected JSON output (for SUCCESS tests only)

// Actual TinyC code follows...
```

Code listing 4.8 Example test file structure showing the required metadata header

This approach separates test expectations from the code being tested, making it easier to understand the purpose of each test and to verify correct functionality. The addition of the `RUN` directive enables the test framework to support different testing modes, particularly distinguishing between parser validation tests and execution tests that can be used once a backend implementation is available.

4.7.2 Test Categories

The test suite encompasses several categories of tests. The basic language feature tests cover empty programs, variable declarations and initializations, arrays and pointers, basic expressions and operations, and function declarations and definitions. Advanced language feature tests include control structures (if-else, loops, switch-case), struct declarations and usage, function pointers, type casting, and complex expressions with nested operations.

For error handling, the test suite verifies proper detection and reporting of lexical errors (such as invalid characters and unterminated strings/comments) and syntax errors (including mismatched parentheses and missing semicolons). Additionally, the test suite addresses edge cases by testing deeply nested expressions, complex combinations of language features, and boundary conditions for different constructs.

These comprehensive test categories ensure that the parser correctly handles the full spectrum of `tinyC` language features while providing appropriate error messages for invalid inputs.

4.7.3 Test Runner Capabilities

The `test_runner.py` script executes and validates tests through several key features. It supports targeted testing through command-line options, allowing users to run individual tests by number (`--test` flag), a range of tests (`--range` flag), or all available tests by default.

For successful parses, the runner validates the JSON output against a formal schema specification, performs structural comparison between expected and actual AST structures, and uses semantic equivalence comparison rather than exact string matching. The runner handles source location information, which can vary but must follow the correct format.

When testing error conditions, the runner verifies that the appropriate error type is reported (lexical or syntax), checks for informative error messages, and validates that the parser exits with a non-zero code for errors. The test runner provides test-by-test reports showing pass/fail status, error diagnostics for failed tests, comparison of expected vs. actual outputs with previews, and summary statistics at the end.

4.7.4 Schema Validation

The test suite includes a formal JSON schema that defines the expected structure of the AST. This schema validates node types and required fields, ensures proper nesting of AST components, and verifies that source location information is present and properly formatted. As shown in Listing 4.9, the schema defines the root Program node structure with its required properties and references to other node definitions. The schema serves both as documentation of the JSON format and as a validation tool, ensuring that AST outputs conform to the expected structure. This is particularly valuable for students implementing their own parsers, as it provides immediate feedback about structural correctness.

4.7.5 Test Generation Utilities

The `test_generator.py` utility complements the test runner by automatically generating test files from example `tinyC` code. It creates expected JSON outputs for validation, maintains a consistent test nomenclature and organization, and supports different test categories with appropriate descriptions. The test generator provides a systematic way to create new tests as the implementation evolves, ensuring consistent test coverage across language features.

```
{
    "type": "object",
    "properties": {
        "nodeType": {
            "enum": ["Program"]
        },
        "declarations": {
            "type": "array",
            "items": {
                "oneOf": [
                    { "$ref": "#/definitions/VariableDeclaration" },
                    { "$ref": "#/definitions/FunctionDeclaration" },
                    { "$ref": "#/definitions/StructDeclaration" },
                    { "$ref": "#/definitions/FunctionPointerDeclaration" }
                ]
            }
        },
        "location": { "$ref": "#/definitions/SourceLocation" }
    },
    "required": ["nodeType", "declarations", "location"]
}
```

Code listing 4.9 Excerpt from the JSON schema showing the root Program node structure

4.7.6 Educational Value

The test suite is designed with educational objectives in mind. Tests progressively introduce language features, helping students understand the language incrementally. The detailed error reporting guides students toward correct implementations, while the test framework itself demonstrates good testing practices. Students can extend the test suite with their own tests as they implement additional features, providing a practical way to learn about test-driven development and compiler construction.

4.8 Summary

This chapter has outlined the design requirements and architecture of the `tinyC` compiler frontend. The functional and non-functional requirements define what the system must accomplish and how it should behave. The `tinyC` grammar analysis identified key challenges and informed the parsing approach. The system architecture, AST design, API specifications, and test suite architecture collectively provide a comprehensive blueprint for an educational compiler frontend that meets the needs of NI-GEN course students.

The design emphasizes clarity, modularity, and educational value, enabling students to focus on compiler middle-end and back-end development while providing a solid foundation for understanding compiler frontends.

Chapter 5
Evaluation

This chapter presents a concise evaluation of the `tinyC` compiler frontend developed in this thesis, focusing on its practical advantages and limitations within the educational context of the NI-GEN course.

5.1 Comparison with Existing Implementation

The current `tinyC` parser implementation provided to students in the NI-GEN course has several significant shortcomings that this project aims to address:

- **Reliability Issues**: The existing implementation contains numerous bugs that require substantial debugging effort from students before they can even begin their backend work.
- **Inflexible Design**: Students are effectively forced to implement their backends in C++ regardless of their proficiency level, as the existing parser cannot easily output to a language-agnostic format.
- **Limited Documentation**: The existing implementation lacks documentation, making it difficult for students to understand the inner workings of the parser.

The `tinyC` compiler frontend developed in this thesis addresses these issues through its implementation, clear object-oriented design, test suite, and language-agnostic JSON output. The dual interface approach—providing both a C++ library and a standalone executable—gives students the flexibility to work in their preferred programming language without sacrificing quality or functionality.

5.2 Functional Assessment

The frontend successfully fulfills the key requirements outlined in Chapter 4:

- The lexical analyzer correctly identifies all token types in the `tinyC` language, handling whitespace, comments, and reporting precise location information for errors.
- The syntax analyzer implements the LL(1) grammar and accurately constructs an AST that represents the program structure with clear error messages.
- The AST implementation follows object-oriented principles with a clean hierarchy of node types, and consistently tracks source location information.
- The JSON output follows a well-defined schema that includes all necessary information from the AST nodes and their source locations.

5.3 Limitations

Despite its improvements over the existing implementation, the frontend has certain limitations:

- **Limited Error Recovery**: The parser stops after encountering the first syntax error rather than attempting to continue and identify multiple errors in a single pass.
- **No Student Feedback**: Unfortunately, due to timing constraints—this thesis being completed in the summer semester while the NI-GEN course runs in the winter semester—it has not been possible to gather direct feedback from students on the frontend's effectiveness in a real course setting.
- **Semantic Analysis Boundaries**: By design, the frontend does not perform semantic analysis tasks such as type checking or scope resolution, leaving these aspects for student implementation.

5.4 Summary

The evaluation indicates that the `tinyC` compiler frontend represents a significant improvement over the existing implementation provided to students. Its clean architecture, documentation, and dual interface approach make it well-suited for educational purposes, while its implementation and testing ensure reliability.

While direct student feedback is not yet available, the technical improvements and design considerations suggest that the frontend will effectively serve its primary goal of allowing students to focus on backend compiler development, providing a solid foundation for their coursework in the NI-GEN course.

Chapter 6
Conclusion

This thesis has presented the design and implementation of a compiler frontend for the `tinyC` programming language, specifically created for students of the NI-GEN course. The project aimed to provide students with a reliable parsing tool that would allow them to focus on the middle-end and back-end aspects of compiler construction, which are the main learning objectives of the course.

6.1 Summary of Achievements

The key achievements of this thesis include:

- Development of a lexical analyzer capable of tokenizing `tinyC` source code with location tracking
- Implementation of a predictive recursive descent parser that handles the entire `tinyC` grammar and generates meaningful error messages
- Design of a clean, object-oriented abstract syntax tree structure with a visitor pattern implementation
- Creation of both a direct C++ library interface and a standardized JSON output format for language-agnostic consumption
- Implementation of a testing framework with metadata-driven test files, schema validation, and detailed error reporting

6.2 Revisiting the Original Objectives

The thesis set out to accomplish four main objectives, which have been successfully addressed:

1. **Analyze the landscape of language parsers and language-agnostic AST representations**: Chapter 3 provided an analysis of existing parser technologies, AST designs, and serialization formats, forming the theoretical foundation for the implementation decisions.
2. **Design and document AST representation for `tinyC` and its JSON format**: Chapter 4 detailed the AST hierarchy for `tinyC`, documenting both the internal class structure and the corresponding JSON schema.
3. **Design, document, implement and test the `tinyC` parser**: Chapter 4 presented the design and implementation of the entire frontend, supported by an extensive test suite.
4. **Discuss further development of the project**: The limitations and potential extensions have been discussed in Chapter 5 (Evaluation) and below in Future Work.

6.3 Limitations

While the implemented frontend successfully meets its primary objectives, some limitations remain:

- The error recovery mechanisms are relatively basic, stopping after the first syntax error rather than attempting to recover and continue parsing
- The frontend focuses exclusively on lexical and syntactic analysis, deliberately leaving semantic analysis for student implementation

These limitations are intentional boundaries that define where the frontend's responsibilities end and where student work on the compiler middle-end begins.

6.4 Future Work

Several potential enhancements could be considered for future development:

- **Enhanced error recovery**: Implementing more sophisticated error recovery techniques would allow the parser to continue after encountering errors, providing more detailed feedback.
- **IDE integration**: Developing plugins for popular IDEs would provide syntax highlighting, code completion, and inline error reporting for `tinyC`.
- **Web-based interface**: Creating a web application that allows students to experiment with `tinyC` parsing would make the tool more accessible.
- **Language server protocol implementation**: Supporting LSP would enable integration with a wide range of editors and IDEs.

6.5 Educational Value

The clean, object-oriented design and well-documented API further enhance the educational value of the frontend, providing students with clear examples and patterns they can apply in their own work. The visitor pattern implementation, in particular, demonstrates an elegant solution to the problem of separating concerns in compiler design.

The testing suite represents an additional contribution beyond the core frontend implementation. It provides students with a systematic way to verify their own implementations through metadata-driven test files and schema validation. The test framework demonstrates good software engineering practices, including test-driven development, automated validation, and detailed error reporting. This extra work significantly enhances the educational value of the project by giving students both a reference implementation and a mechanism to validate their own work.

6.6 Closing Remarks

The `tinyC` compiler frontend developed in this thesis provides a solid foundation for students in the NI-GEN course to focus on the middle-end and back-end aspects of compiler construction. By offering both a direct C++ library interface and a language-agnostic JSON output format, it accommodates a variety of implementation approaches while maintaining a consistent representation of the `tinyC` language.

While there are opportunities for enhancement, the current implementation successfully fulfills its primary objective: providing a reliable, well-designed, and tested compiler frontend that allows students to focus on the core learning objectives of the NI-GEN course. The true measure of success will be seen when the frontend is deployed in future offerings of the course, potentially transforming the learning experience by enabling deeper exploration of optimization techniques and code generation strategies.

Appendix A

Grammar

⟨PROGRAM⟩	::=	⟨PROG_ITEM⟩ ⟨PROGRAM⟩ | ε
⟨PROG_ITEM⟩	::=	⟨NON_VOID_TYPE⟩ `identifier` ⟨NON_VOID_TAIL⟩ |
		`void` ⟨VOID_DECL_TAIL⟩ |
		⟨STRUCT_DECL⟩ |
		⟨FUNPTR_DECL⟩
⟨NON_VOID_TAIL⟩	::=	⟨VAR_TAIL⟩ | ⟨FUNC_DECL_TAIL⟩
⟨VOID_DECL_TAIL⟩	::=	`identifier` ⟨FUNC_DECL_TAIL⟩ |
		⟨STAR_PLUS⟩ `identifier` ⟨FUNC_OR_VAR_TAIL⟩
⟨FUNC_OR_VAR_TAIL⟩	::=	⟨VAR_TAIL⟩ | ⟨FUNC_DECL_TAIL⟩
⟨VAR_TAIL⟩	::=	⟨OPT_ARRAY_SIZE⟩ ⟨OPT_INIT⟩
		⟨MORE_GLOBAL_VARS⟩ `;`
⟨FUNC_DECL_TAIL⟩	::=	`(` ⟨OPT_FUN_ARGS⟩ `)` ⟨FUNC_TAIL⟩
⟨FUNC_TAIL⟩	::=	⟨BLOCK_STMT⟩ | `;`
⟨MORE_GLOBAL_VARS⟩	::=	`,` `identifier` ⟨OPT_ARRAY_SIZE⟩ ⟨OPT_INIT⟩
		⟨MORE_GLOBAL_VARS⟩ |
		ε
⟨OPT_FUN_ARGS⟩	::=	⟨FUN_ARG⟩ ⟨FUN_ARG_TAIL⟩ | ε
⟨FUN_ARG_TAIL⟩	::=	`,` ⟨FUN_ARG⟩ ⟨FUN_ARG_TAIL⟩ | ε
⟨FUN_ARG⟩	::=	⟨TYPE⟩ `identifier`
⟨STMT⟩	::=	⟨BLOCK_STMT⟩ | ⟨IF_STMT⟩ | ⟨SWITCH_STMT⟩ |
		⟨WHILE_STMT⟩ | ⟨DO_WHILE_STMT⟩ |
		⟨FOR_STMT⟩ |
		⟨BREAK_STMT⟩ | ⟨CONTINUE_STMT⟩ |
		⟨RETURN_STMT⟩ |
		⟨EXPR_STMT⟩
⟨BLOCK_STMT⟩	::=	`{` ⟨STMT_STAR⟩ `}`
⟨STMT_STAR⟩	::=	⟨STMT⟩ ⟨STMT_STAR⟩ | ε
⟨IF_STMT⟩	::=	`if` `(` ⟨EXPR⟩ `)` ⟨STMT⟩ ⟨ELSE_PART⟩
⟨ELSE_PART⟩	::=	`else` ⟨STMT⟩ | ε
⟨SWITCH_STMT⟩	::=	`switch` `(` ⟨EXPR⟩ `)` `{` ⟨CASE_DEFLT_STAR⟩ `}`
⟨CASE_DEFLT_STAR⟩	::=	⟨CASE_STMT⟩ ⟨CASE_DEFLT_STAR⟩ |
		ε |
		⟨DEFAULT_CASE⟩ ⟨CASE_STAR⟩
⟨CASE_STAR⟩	::=	⟨CASE_STMT⟩ ⟨CASE_STAR⟩ | ε
⟨CASE_STMT⟩	::=	`case` `integer_literal` `:` ⟨CASE_BODY⟩

⟨CASE_BODY⟩ ::= ⟨STMT_STAR⟩
⟨DEFAULT_CASE⟩ ::= `default :` ⟨CASE_BODY⟩
⟨WHILE_STMT⟩ ::= `while (` ⟨EXPR⟩ `)` ⟨STMT⟩
⟨DO_WHILE_STMT⟩ ::= `do` ⟨STMT⟩ `while (` ⟨EXPR⟩ `) ;`
⟨FOR_STMT⟩ ::= `for (` ⟨OPT_DECL⟩ `;` ⟨OPT_EXPR⟩ `;`
⟨OPT_EXPR⟩ `)` ⟨STMT⟩
⟨OPT_DECL⟩ ::= ⟨EXPR_OR_VAR_DECL⟩ | ε
⟨OPT_EXPR⟩ ::= ⟨EXPR⟩ | ε
⟨BREAK_STMT⟩ ::= `break ;`
⟨CONTINUE_STMT⟩ ::= `continue ;`
⟨RETURN_STMT⟩ ::= `return` ⟨OPT_EXPR⟩ `;`
⟨EXPR_STMT⟩ ::= ⟨EXPR_OR_VAR_DECL⟩ `;`
⟨EXPR_OR_VAR_DECL⟩ ::= ⟨VAR_DECLS⟩ | ⟨EXPRS⟩
⟨VAR_DECLS⟩ ::= ⟨VAR_DECL⟩ ⟨VAR_DECLS_TAIL⟩
⟨VAR_DECLS_TAIL⟩ ::= `,` ⟨VAR_DECL⟩ ⟨VAR_DECLS_TAIL⟩ | ε
⟨VAR_DECL⟩ ::= ⟨TYPE⟩ `identifier` ⟨OPT_ARRAY_SIZE⟩ ⟨OPT_INIT⟩
⟨OPT_ARRAY_SIZE⟩ ::= `[` ⟨E9⟩ `]` | ε
⟨OPT_INIT⟩ ::= `=` ⟨EXPR⟩ | ε
⟨EXPRS⟩ ::= ⟨EXPR⟩ ⟨EXPRS_TAIL⟩
⟨EXPRS_TAIL⟩ ::= `,` ⟨EXPR⟩ ⟨EXPRS_TAIL⟩ | ε
⟨TYPE⟩ ::= ⟨BASE_TYPE⟩ ⟨STAR_SEQ⟩ |
⟨TYPENAME⟩ ⟨STAR_SEQ⟩ |
`void` ⟨STAR_PLUS⟩
⟨NON_VOID_TYPE⟩ ::= ⟨BASE_TYPE⟩ ⟨STAR_SEQ⟩ |
⟨TYPENAME⟩ ⟨STAR_SEQ⟩
⟨BASE_TYPE⟩ ::= `int` | `double` | `char`
⟨TYPE_FUN_RET⟩ ::= ⟨FUN_RET_TYPES⟩ ⟨STAR_SEQ⟩
⟨FUN_RET_TYPES⟩ ::= `void` | ⟨BASE_TYPE⟩ | ⟨TYPENAME⟩
⟨STAR_PLUS⟩ ::= `*` ⟨STAR_SEQ⟩
⟨STAR_SEQ⟩ ::= `*` ⟨STAR_SEQ⟩ | ε
⟨STRUCT_DECL⟩ ::= `struct identifier` ⟨OPT_STRUCT_BODY⟩ `;`
⟨OPT_STRUCT_BODY⟩ ::= `{` ⟨STRUCT_FIELDS⟩ `}` | ε
⟨STRUCT_FIELDS⟩ ::= ⟨STRUCT_FIELD⟩ ⟨STRUCT_FIELDS⟩ | ε
⟨STRUCT_FIELD⟩ ::= ⟨TYPE⟩ `identifier ;`
⟨FUNPTR_DECL⟩ ::= `typedef` ⟨TYPE_FUN_RET⟩ `( * identifier )`
`(` ⟨OPT_FUNPTR_ARGS⟩ `) ;`
⟨OPT_FUNPTR_ARGS⟩ ::= ⟨FUNPTR_ARGS⟩ | ε
⟨FUNPTR_ARGS⟩ ::= ⟨TYPE⟩ ⟨FUNPTR_ARGS_TAIL⟩
⟨FUNPTR_ARGS_TAIL⟩ ::= `,` ⟨TYPE⟩ ⟨FUNPTR_ARGS_TAIL⟩ | ε
⟨EXPR⟩ ::= ⟨E9⟩ ⟨EXPR_TAIL⟩
⟨EXPR_TAIL⟩ ::= `=` ⟨EXPR⟩ | ε
⟨E9⟩ ::= ⟨E8⟩ ⟨E9_Prime⟩
⟨E9_Prime⟩ ::= `||` ⟨E8⟩ ⟨E9_Prime⟩ | ε
⟨E8⟩ ::= ⟨E7⟩ ⟨E8_Prime⟩
⟨E8_Prime⟩ ::= `&&` ⟨E7⟩ ⟨E8_Prime⟩ | ε
⟨E7⟩ ::= ⟨E6⟩ ⟨E7_Prime⟩
⟨E7_Prime⟩ ::= `|` ⟨E6⟩ ⟨E7_Prime⟩ | ε
⟨E6⟩ ::= ⟨E5⟩ ⟨E6_Prime⟩
⟨E6_Prime⟩ ::= `&` ⟨E5⟩ ⟨E6_Prime⟩ | ε
⟨E5⟩ ::= ⟨E4⟩ ⟨E5_Prime⟩
⟨E5_Prime⟩ ::= `==` ⟨E4⟩ ⟨E5_Prime⟩ |
`!=` ⟨E4⟩ ⟨E5_Prime⟩ | ε
⟨E4⟩ ::= ⟨E3⟩ ⟨E4_Prime⟩
⟨E4_Prime⟩ ::= `<` ⟨E3⟩ ⟨E4_Prime⟩ |

		`<=` ⟨E3⟩ ⟨E4_Prime⟩ |
		`>` ⟨E3⟩ ⟨E4_Prime⟩ |
		`>=` ⟨E3⟩ ⟨E4_Prime⟩ | ε
⟨E3⟩	::=	⟨E2⟩ ⟨E3_Prime⟩
⟨E3_Prime⟩	::=	`<<` ⟨E2⟩ ⟨E3_Prime⟩ |
		`>>` ⟨E2⟩ ⟨E3_Prime⟩ | ε
⟨E2⟩	::=	⟨E1⟩ ⟨E2_Prime⟩
⟨E2_Prime⟩	::=	`+` ⟨E1⟩ ⟨E2_Prime⟩ |
		`-` ⟨E1⟩ ⟨E2_Prime⟩ | ε
⟨E1⟩	::=	⟨E_UNARY_PRE⟩ ⟨E1_Prime⟩
⟨E1_Prime⟩	::=	`*` ⟨E_UNARY_PRE⟩ ⟨E1_Prime⟩ |
		`/` ⟨E_UNARY_PRE⟩ ⟨E1_Prime⟩ |
		`%` ⟨E_UNARY_PRE⟩ ⟨E1_Prime⟩ | ε
⟨E_UNARY_PRE⟩	::=	`+` ⟨E_UNARY_PRE⟩ |
		`-` ⟨E_UNARY_PRE⟩ |
		`!` ⟨E_UNARY_PRE⟩ |
		`~` ⟨E_UNARY_PRE⟩ |
		`++` ⟨E_UNARY_PRE⟩ |
		`--` ⟨E_UNARY_PRE⟩ |
		`*` ⟨E_UNARY_PRE⟩ |
		`&` ⟨E_UNARY_PRE⟩ |
		⟨E_POST⟩
⟨E_POST⟩	::=	⟨F⟩ ⟨E_POST_TAIL⟩
⟨E_POST_TAIL⟩	::=	⟨E_CALL⟩ ⟨E_POST_TAIL⟩ |
		⟨E_INDEX⟩ ⟨E_POST_TAIL⟩ |
		⟨E_MEMBER⟩ ⟨E_POST_TAIL⟩ |
		⟨E_POSTFIX⟩ ⟨E_POST_TAIL⟩ |
		ε
⟨E_CALL⟩	::=	`(` ⟨OPT_EXPR_LIST⟩ `)`
⟨OPT_EXPR_LIST⟩	::=	⟨EXPR⟩ ⟨EXPR_TAIL_LIST⟩ | ε
⟨EXPR_TAIL_LIST⟩	::=	`,` ⟨EXPR⟩ ⟨EXPR_TAIL_LIST⟩ | ε
⟨E_INDEX⟩	::=	`[` ⟨EXPR⟩ `]`
⟨E_MEMBER⟩	::=	`.` `identifier` |
		`->` `identifier`
⟨E_POSTFIX⟩	::=	`++` | `--`
⟨F⟩	::=	`integer_literal` |
		`double_literal` |
		`char_literal` |
		`string_literal` |
		`identifier` |
		`(` ⟨EXPR⟩ `)` |
		⟨E_CAST⟩
⟨E_CAST⟩	::=	`cast` `<` ⟨TYPE⟩ `>` `(` ⟨EXPR⟩ `)`

Bibliography

1. AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, Techniques, & Tools*. 2nd ed. Pearson/Addison Wesley, 2007. ISBN 9780321486813.
2. INTERACTIVE, Runestone. *Formal and Natural Languages* [online]. 2024. [visited on 2025-04-18]. Available from: `https://runestone.academy/ns/books/published/thinkcspy/GeneralIntro/FormalandNaturalLanguages.html`. [Online].
3. DRAGAN, Feodor F. *Context-Free Grammars and Pushdown Automata* [online]. 2023. [visited on 2025-04-18]. Available from: `https://www.cs.kent.edu/~dragan/ThComp/lect07-2.pdf`. [Online].
4. PARR, Terence. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010.
5. HOLUB, Allen I. *Compiler Design in C*. Prentice Hall, 1990.
6. GRUNE, Dick; VAN REEUWIJK, Kees; BAL, Henri E.; JACOBS, Ceriel J.H.; LANGENDOEN, Koen. *Modern Compiler Design*. Springer, 2012.
7. CRENSHAW, Jack W. Let's Build a Compiler. *Byte*. 1988.
8. PARR, Terence. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
9. LEVINE, John. *Flex & Bison*. O'Reilly Media, 2009.
10. SESTOFT, Peter. *Programming Language Concepts*. Springer, 2017.
11. TRATT, Laurence. Parsing: The solved problem that isn't. *Software Development Times*. 2010.
12. HUTTON, Graham. Higher-order functions for parsing. *Journal of Functional Programming*. 1992, vol. 2, no. 3, pp. 323–343.

13. LEIJEN, Daan; MEIJER, Erik. Parsec: Direct style monadic parser combinators for the real world. *Technical Report UU-CS-2001-35, Department of Computer Science, Universiteit Utrecht*. 2001.

14. MARLOW, Simon. Haskell 2010 language report. 2011. Available also from: `https://www.haskell.org/onlinereport/haskell2010/`. [Online].

15. MOORS, Adriaan; PIESSENS, Frank; ODERSKY, Martin. Parser combinators in Scala. *Technical Report CW491, Department of Computer Science, KU Leuven*. 2008.

16. FORD, Bryan. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*. 2004, vol. 39, no. 1, pp. 111–122.

17. GRIMM, Robert. Better extensibility through modular syntax. *ACM SIGPLAN Notices*. 2006, vol. 41, no. 6, pp. 38–51.

18. MEDEIROS, Sérgio; MASCARENHAS, Fabio. PEG parsing: Combining ease of implementation with efficient parsing. *Science of Computer Programming*. 2014, vol. 96, pp. 195–210.

19. FORD, Bryan. Packrat parsing: a practical linear-time algorithm with backtracking. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Scheme and Functional Programming*. 2002, pp. 36–47.

20. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

21. APPEL, Andrew W. *Modern Compiler Implementation in Java*. Cambridge University Press, 2004.

22. ODERSKY, Martin; ALTHERR, Philippe; CREMET, Vincent; EMIR, Burak; MANETH, Sebastian; MICHELOUD, Stéphane; MIHAYLOV, Nikolay; SCHINZ, Michel; STENMAN, Erik; ZENGER, Matthias. An overview of the Scala programming language. *Technical Report LAMP-REPORT-2004-006, EPFL*. 2004.

23. KRISHNAMURTHI, Shriram. *Programming Languages: Application and Interpretation*. Self-published, 2007.

24. OKASAKI, Chris. Purely functional data structures. *Journal of Functional Programming*. 1999, vol. 9, no. 1, pp. 1–8.

25. WADLER, Philip. The expression problem. *Java-genericity mailing list*. 1998.

26. MARTIN, Robert C.; RIEHLE, Dirk; BUSCHMANN, Frank. Acyclic visitor. *Pattern Languages of Program Design*. 2000, vol. 3, pp. 93–104.

27. PALSBERG, Jens; JAY, C. Barry. The Essence of the Visitor Pattern. In: *COMPSAC'98. Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference*. 1998, pp. 9–15.

28. LESK, Michael E.; SCHMIDT, Eric. Lex: A lexical analyzer generator. *Computing Science Technical Report*. 1975, vol. 39.

29. CROCKFORD, Douglas. The application/json Media Type for JavaScript Object Notation (JSON). *Internet Engineering Task Force, RFC*. 2006, vol. 4627.

30. BABEL TEAM. ESTree Specification. *GitHub repository*. 2015. Available also from: `https://github.com/estree/estree`. [Online].

31. MICROSOFT. *TypeScript Compiler API*. 2018. Available also from: `https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API`. [Online].

32. LATTNER, Chris. LLVM and Clang: Next generation compiler technology. *The BSD Conference*. 2008, pp. 1–2.

33. JSON SCHEMA AUTHORS. JSON Schema: A Media Type for Describing JSON Documents. *Internet Engineering Task Force, Internet-Draft*. 2019. Available also from: `https://json-schema.org/`. [Online].

34. BRAY, Tim. The JavaScript Object Notation (JSON) Data Interchange Format. *Internet Engineering Task Force, RFC*. 2017, vol. 8259.

35. BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C. M. Extensible Markup Language (XML). *World Wide Web Journal*. 1997, vol. 2, no. 4, pp. 27–66.

36. KAY, Michael. *XSLT Programmer's Reference*. Wrox Press, 2000.

37. CLARK, James; DEROSE, Steve. XML Path Language (XPath). *W3C Recommendation*. 1999, vol. 16.

38. NURSEITOV, Nurzhan; PAULSON, Michael; REYNOLDS, Randall; IZURIETA, Clemente. Comparison of JSON and XML data interchange formats: a case study. In: *CAINE*. 2009, vol. 9, pp. 157–162.

39. WARREN, Henry S. *Hacker's Delight*. Addison-Wesley, 2006.

40. VARDA, Kenton. Protocol buffers: Google's data interchange format. *Google Open Source Blog*. 2008.

41. FURUHASHI, Sadayuki. *MessagePack: It's like JSON but fast and small*. 2013. Available also from: `https://msgpack.org`. [Online].

42. BORMANN, Carsten; HOFFMAN, Paul. Concise Binary Object Representation (CBOR). *Internet Engineering Task Force, RFC*. 2013, vol. 7049.

43. MICROSOFT. Language Server Protocol Specification. 2016. Available also from: `https://microsoft.github.io/language-server-protocol/specifications/specification-current/`. [Online].

44. APPEL, Andrew W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

45. PATEL, Kiran; WRIGSTAD, Tobias. Comparing Educational Compiler Frameworks. *Journal of Computer Science Education*. 2021, vol. 31, no. 3, pp. 283–305.

46. LATTNER, Chris; ADVE, Vikram. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization*. 2004, pp. 75–86.

47. AIKEN, Alex. *COOL: The Classroom Object-Oriented Language*. 2003. Available also from: `https://theory.stanford.edu/~aiken/software/cool/cool.html`. [Online].

48. HSU, David. *CMSC430: Design and Implementation of Programming Languages*. 2020. Available also from: `https://www.cs.umd.edu/class/fall2020/cmsc430/`. [Online].

49. TRAVER, V. Javier. Compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*. 2010, vol. 2010.

50. HORWITZ, Susan. Student misunderstandings of programming language concepts: identification and treatment in a compiler course. *Frontiers in Education Conference*. 2007.

51. FORWARD, Andrew; LETHBRIDGE, Timothy C. The relevance of software documentation, tools and technologies: a survey. In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. 2002, pp. 26–33.

52. JONES, Joel. Minimal-symbol-table-free implementations of programming languages. *Software: Practice and Experience*. 1997, vol. 27, no. 8, pp. 853–880.

Contents of the attachments

```
/
├── README.md ..... project description and setup instructions
├── project ..... implementation source code
│   ├── include ..... header files
│   ├── src ..... source files
│   │   ├── lexer ..... lexical analyzer implementation
│   │   ├── parser ..... parser implementation
│   │   ├── ast ..... abstract syntax tree implementation
│   │   └── main.cpp ..... executable entry point
│   ├── tests ..... test files
│   │   ├── lexer ..... lexer unit tests
│   │   └── parser ..... parser unit tests
│   ├── test_suite ..... testing suite
│   │   ├── tinyc-ast-schema.json ..... JSON schema
│   │   ├── tests ..... TinyC test files
│   │   ├── test_runner.py ..... main test runner script
│   │   └── test_generator.py ..... test case generator
│   ├── docs ..... documentation
│   ├── CMakeLists.txt ..... CMake build configuration
│   └── Doxyfile ..... Doxygen configuration
└── paper ..... thesis text
    └── thesis.pdf ..... thesis text in PDF format
```