Bachelor's thesis

TINYC COMPILER FRONTEND

Mykhailo Anisimov

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Petr Máj, Ph.D.
May 9, 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.

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.

In Prague on May 9, 2025

Abstract

This thesis presents a universal compiler frontend for the `tinyC` programming language used in the NI-GEN course. The frontend allows students to implement compiler backends in any language of their choice via a standardized JSON interface, while also providing a direct C++ library option. Based on a thorough analysis of parser technologies and AST representations, the implementation features a recursive descent parser with LL(1) grammar, comprehensive source location tracking, and a well-defined JSON schema. With its clean architecture and extensive documentation, this frontend serves as an effective 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 univerzální překladový frontend pro programovací jazyk `tinyC` používaný v kurzu NI-GEN. Frontend umožňuje studentům implementovat zadní části překladače v libovolném jazyce prostřednictvím standardizovaného JSON rozhraní, přičemž nabízí i přímé použití C++ knihovny. Na základě důkladné analýzy technologií parserů a reprezentací AST implementace zahrnuje rekurzivní sestupný parser s LL(1) gramatikou, komplexní sledování pozic ve zdrojovém kódu a jasně definované JSON schéma. Díky své přehledné architektuře a rozsáhlé dokumentaci slouží tento frontend jako efektivní 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 universal 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 robust and 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 comprehensive, 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 robust 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 universal compiler frontend for the `tinyC` language. The key objectives are:

- Developing a robust 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 thorough 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

The construction of robust and efficient compilers relies heavily on a strong theoretical foundation. This chapter aims to establish the importance of these theoretical underpinnings by exploring the key areas that are fundamental to understanding and building the frontend of a compiler. These areas include the principles of formal languages, the distinct phases of the compilation process, the intricacies of lexical analysis, the methodologies of syntax analysis, and the design considerations for abstract syntax trees.

A solid grasp of these theoretical concepts empowers compiler developers to make informed design decisions, effectively troubleshoot challenges, and ultimately create reliable and performant language processing tools. The ability to build a compiler from its foundational principles, as well as to leverage existing compiler construction tools and understand their inherent limitations, is significantly enhanced by a comprehensive theoretical understanding.

2.1 Formal Languages

▶ **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.

2.1.1 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].

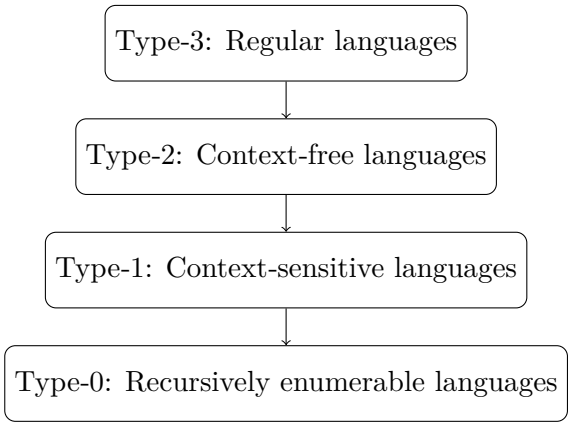


Figure 2.1 Chomsky hierarchy of formal languages

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.

2.1.2 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].

2.1.3 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.

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).

- **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].
- **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].
- **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].
- **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].
- **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].
- **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.

2.2.1 Emphasis on the Frontend Phases

This thesis concentrates on the compiler front end—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:

- 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].
- Remove whitespace (spaces, tabs, newlines) and comments, as they serve only to separate tokens and are not needed by the parser [1].
- Detect and report lexical errors, such as invalid characters or character sequences that do not match any token pattern [1].
- Store preliminary information about tokens—particularly identifiers—in the symbol table for later compiler phases [1].
- Track the location (line and column numbers) of each token for precise error reporting and downstream stages.
- 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 equivalence of regex and finite automata underpins scanner implementation [1]. Main steps:

- Convert each regular expression into an NFA.
- Apply the powerset construction to transform NFAs into DFAs.
- Build a transition table indexed by state and input symbol.
- Scan input by following DFA transitions; reaching an accepting state emits the corresponding token.
- Use the DFA's accepting-state information to determine token type.
- Report errors when no valid transition exists for an input character.

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]:

- **Syntax Verification:** Ensure the token sequence conforms to the language grammar.
- **Structural Construction:** Build a hierarchical representation (parse tree or Abstract Syntax Tree).
- **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 (CFGs): Defining Syntax

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 shows the explanatory text and a general parse tree side by side.

2.4.2 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:

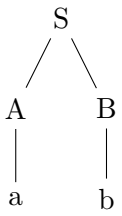
Explanation	General Parse Tree
A CFG derivation can be visualised as a tree: the start symbol is at the root, and each application of a production rule adds children corresponding to the symbols on the right-hand side. For instance, given: $S \to AB, \quad A \to a, \quad B \to b,$ the parse tree captures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

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

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].

2.4.3 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].

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].

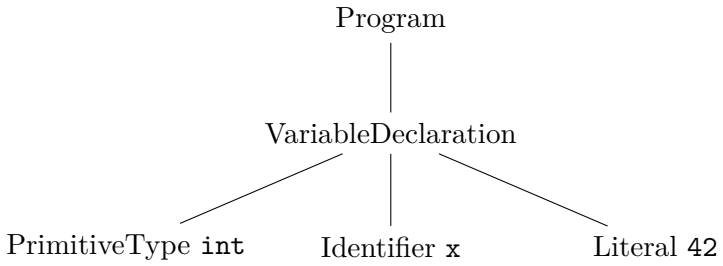


Figure 2.2 AST of the program `int x = 42;`

2.5.1 Key Components

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)`, the root node represents the addition operation (`+`), with two children: one for the literal `2`, and one for the subtraction operation (`-`), which in turn has children `z` and `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].

2.5.2 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].

2.5.3 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 several advantages:

- **Directness**: The correspondence between the grammar and code is clear, making it easier for students to understand the relation between formal grammar specifications and their implementation [4].
- **Control over error handling**: Developers have fine-grained control over error messages, allowing for more student-friendly diagnostics [5].
- **Flexibility**: Modifications to the grammar can be directly translated to code changes without intermediary tools [6].

However, handwritten parsers also have limitations:

- **Labor-intensive**: They require considerable manual coding, especially for complex languages [6].
- **Left recursion**: They cannot handle left recursion directly, necessitating grammar transformations [1].
- **Maintenance burden**: Changes to the grammar require coordinated changes across potentially many functions [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:

- **Productivity**: They generate parser code from grammar specifications, reducing implementation time [9].
- **Formal basis**: They implement well-studied parsing algorithms with proven properties [1].
- **Grammar checking**: Many parser generators validate the grammar and report potential issues [8].

However, they also present challenges in educational contexts:

- **Learning curve**: Students must learn both the grammar specification language and the integration patterns with the host language [10].
- **Abstraction gap**: The generated code may obscure the connection between grammar and parsing logic [6].
- **Error reporting**: Default error messages are often cryptic, requiring additional work to make them student-friendly [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.

Advantages of parser combinators include:

- **Composability**: Simple parsers can be combined in modular ways to handle complex structures [13].

- **Type safety**: In statically typed languages, type errors in parser construction are caught at compile time [14].
- **Readable code**: The parsing logic often resembles the grammar directly [15].

Limitations include:

- **Performance**: Naive implementations may have poor performance due to backtracking [13].
- **Functional programming prerequisite**: Students need familiarity with functional programming concepts [10].
- **Limited error recovery**: Advanced error recovery techniques can be difficult to implement [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 advantages:

- **No ambiguity**: The ordered choice operator eliminates parsing ambiguities by definition [16].
- **Integrated lexing and parsing**: Many PEG implementations combine lexical and syntactic analysis [17].
- **Expressive power**: They can recognize some non-context-free languages [18].

However, they also have drawbacks:

- **Subtle behavior**: The ordered choice can lead to unexpected parsing behavior [11].
- **Memory usage**: Packrat parsing, while efficient in time, can have high memory requirements [19].
- **Less theory**: PEGs have less theoretical foundation compared to LR or LL parsing [18].

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

3.1.5 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.

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#.

Benefits of this approach include:

- **Intuitive modeling**: The class hierarchy naturally maps to the syntactic categories of the language [21].
- **Extensibility**: New node types can be added by extending the hierarchy [20].
- **Type safety**: Strong typing ensures operations are applied to appropriate nodes [22].

Challenges with this approach include:

- **Rigidity**: Adding new operations across the entire AST typically requires modifying every node class [20].
- **Boilerplate code**: Substantial repetitive code is often required for node constructors, accessors, etc. [21].
- **Visitor implementation complexity**: Implementing the visitor pattern correctly requires careful attention to detail [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.

Advantages of ADTs for AST representation include:

- **Conciseness**: ADTs allow for compact definitions of the entire AST structure [23].
- **Pattern matching**: Exhaustive pattern matching facilitates comprehensive AST processing [22].
- **Immutability**: ADTs are typically immutable, simplifying reasoning about AST transformations [24].

Limitations include:

- **Limited language support**: Many mainstream languages used in education do not support ADTs natively [23].
- **Expression problem**: Adding new node types requires modifying all functions that process the AST [25].
- **Memory overhead**: Naive implementations may incur higher memory overhead compared to optimized OO designs [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.

Different implementations of the Visitor pattern offer varying trade-offs:

- **Classic Visitor**: Defines a visitor interface with visit methods for each node type, allowing for new operations without modifying the node classes [20].
- **Acyclic Visitor**: A variant that reduces coupling between the visitor and visitable classes [26].
- **Reflective Visitor**: Uses reflection to dynamically dispatch based on runtime types, reducing the need for explicit accept methods [27].

For educational compilers, the classic Visitor pattern offers a good balance of type safety and extensibility, while teaching an important design pattern [20].

3.2.4 Memory Management Strategies

Memory management for AST nodes is a critical consideration, affecting both performance and correctness. Various approaches exist, from manual memory management to garbage collection and smart pointers.

For the `tinyC` frontend implementation, the decision to use `std::unique_ptr` for AST nodes offers several advantages:

- **Automatic resource management**: Nodes are automatically deleted when their owning pointer goes out of scope, preventing memory leaks [28].
- **Ownership semantics**: `unique_ptr` clearly expresses that the owner has exclusive responsibility for the node's lifetime [29].
- **No reference counting overhead**: Unlike `shared_ptr`, `unique_ptr` has no runtime overhead beyond raw pointers [28].
- **Move semantics**: Efficient transfer of ownership through move operations [29].

This approach is particularly suitable for a tree structure like an AST, where each node typically has a single parent responsible for its lifetime [28]. It also demonstrates modern C++ practices to students while avoiding the complexities of manual memory management.

3.2.5 Location Tracking Approaches

Maintaining source location information is essential for meaningful error messages and debugging support. Several approaches to location tracking exist:

- **Embedded location data**: Including location information directly in each AST node [21].
- **Location maps**: Maintaining a separate mapping from nodes to locations [30].
- **Span-based tracking**: Tracking start and end positions to represent ranges in the source code [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.

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 [31]. Notable examples include:

- **ESTree**: A JSON schema for JavaScript ASTs used by tools like Babel and ESLint [32].
- **TypeScript AST**: Microsoft's TypeScript compiler provides JSON serialization of its AST [33].
- **Clang AST**: Clang's AST can be dumped in a JSON format, though it is quite verbose [34].

Advantages of JSON serialization include:

- **Language independence**: JSON parsers exist for virtually all programming languages [31].

- **Human readability**: The text format is relatively easy to understand and debug [31].
- **Schema support**: JSON Schema can be used to validate AST structures [35].

Limitations include:

- **Verbosity**: JSON representations can be significantly larger than binary formats [36].
- **Limited data types**: JSON's type system is limited, requiring encoding of specialized types [31].
- **No standardized references**: Representing cross-references in the AST requires custom conventions [36].

3.3.2 XML Representations

XML was once the dominant format for serializing complex data structures, including ASTs [37]. While less common today, XML-based AST representations offer some unique features:

- **Schema validation**: XML Schema and DTD provide strong validation capabilities [37].
- **Transformation tools**: XSLT allows for sophisticated transformations of the AST [38].
- **XPath querying**: XPath provides a powerful way to query specific parts of the AST [39].

However, XML has significant drawbacks for modern AST serialization:

- **Extreme verbosity**: XML markup significantly increases the size of the representation [37].
- **Parsing overhead**: XML parsing is generally more expensive than JSON parsing [40].
- **Complexity**: XML technologies have a steeper learning curve compared to JSON [40].

3.3.3 Binary Serialization

For performance-critical applications, binary serialization formats offer significant space and time efficiency advantages [41]. Relevant formats include:

- **Protocol Buffers**: Google's language-neutral, platform-neutral extensible mechanism for serializing structured data [42].
- **MessagePack**: A binary form of JSON with additional data types [43].
- **CBOR**: Concise Binary Object Representation, designed for small code size and small message size [44].

Binary formats offer:

- **Compactness**: Significantly smaller representation compared to text formats [41].
- **Parsing efficiency**: Binary formats can be parsed more efficiently than text-based formats [41].
- **Rich type system**: Many binary formats support a wider range of data types than JSON [42].

Their limitations include:

- **Human unreadability**: Binary formats are not directly human-readable, complicating debugging [41].
- **Tool dependencies**: Special tools are required to inspect and manipulate the data [42].
- **Schema evolution**: Handling changes to the data schema can be complex [42].

3.3.4 Language Server Protocol

The Language Server Protocol (LSP) has established conventions for representing code structures for IDE integration [45]. While not primarily an AST serialization format, LSP's approaches inform modern compiler design:

- **Position-based representation**: LSP uses zero-based line and character offsets to represent source positions [45].
- **Incremental updates**: The protocol supports efficient incremental updates to code structures [45].
- **Standardized diagnostics**: LSP defines a common format for reporting errors and warnings [45].

These conventions are increasingly relevant for educational compiler design, as modern development environments increasingly expect LSP-compatible integration [45].

3.3.5 Schema Design Considerations

When designing an AST serialization schema, several key considerations emerge:

- **Node identification**: How to uniquely identify and reference nodes within the AST [4].
- **Location representation**: How to represent source locations in a language-agnostic way [8].
- **Type system encoding**: How to represent the language's type system in the serialized format [21].
- **Annotations**: How to attach additional metadata such as type information or compiler hints [21].
- **Versioning**: How to handle schema evolution as the language and compiler evolve [42].

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 [46]. Several implementations exist, offering different educational approaches:

- **The Tiger Book implementation**: A reference implementation using ML, focusing on functional programming techniques [46].
- **jmm**: A Java-based implementation emphasizing object-oriented design [47].
- **MiniJava-compiler-construction**: An implementation designed to be extended by students incrementally [47].

These implementations highlight the importance of:

- **Clear separation of concerns**: Well-defined interfaces between compiler phases [46].
- **Incremental learning path**: A structure that allows students to build the compiler in stages [47].
- **Consistent design patterns**: Using consistent patterns throughout the codebase to reinforce concepts [46].

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 [48]:

- **Kaleidoscope tutorial**: LLVM's tutorial for building a simple language frontend [48].
- **COOL compiler**: The Classroom Object-Oriented Language compiler, adapted to use LLVM [49].
- **CMSC430 compiler**: A compiler for a subset of Racket targeting LLVM [50].

These projects demonstrate:

- **Backend independence**: Cleanly separating the frontend from code generation concerns [48].
- **IRs as boundaries**: Using well-defined intermediate representations as interfaces between compiler phases [48].
- **Progressive complexity**: Starting with simple constructs and gradually adding language features [49].

3.4.3 Error Reporting Quality

The quality of error reporting significantly impacts the educational value of a compiler frontend [51]. Analysis of educational compilers reveals several approaches to error reporting:

- **Error recovery in parsing**: Techniques for continuing parsing after errors to report multiple issues [1].
- **Contextual error messages**: Providing context-specific suggestions based on the parsing context [51].
- **Source highlighting**: Visual indication of error locations in the source code [52].

The most effective educational compilers provide:

- **Precise error localization**: Pinpointing the exact location of the error [52].
- **Explanatory messages**: Clear explanations of what went wrong [51].
- **Suggested corrections**: Hints about how to fix the error [51].

3.4.4 Documentation Standards

Effective documentation is crucial for educational compiler projects [53]. Analysis of successful projects reveals several documentation patterns:

- **Architectural overviews**: High-level explanations of the compiler's structure and phases [53].
- **API documentation**: Clear documentation of interfaces between components [53].
- **Tutorial-style guides**: Step-by-step explanations of how to use and extend the compiler [47].
- **Implementation notes**: Explanations of key algorithms and design decisions [46].

The most effective documentation addresses multiple audiences:

- **Students using the compiler**: Clear guides on how to write programs in the target language [53].
- **Students extending the compiler**: Tutorials on adding features or optimizations [47].
- **Instructors**: Materials to help integrate the compiler into coursework [53].

3.4.5 Integration Flexibility

Educational compiler frontends must be designed for flexible integration with various backend components [54]. Successful approaches include:

- **Clean API boundaries**: Well-defined interfaces that hide implementation details [54].
- **Multiple output formats**: Supporting different IR formats for backend compatibility [48].
- **Extensibility hooks**: Designated points where students can add functionality [47].

For `tinyC`, the dual approach of providing both a C++ library and a standalone executable with JSON output maximizes integration flexibility, allowing students to choose the approach that best fits their implementation language and preferences.

3.5 Summary of Design Implications

The analysis of existing solutions leads to several design implications for the `tinyC` frontend:

- **Parser approach**: A handwritten recursive descent parser with LL(1) predictive parsing offers the best balance of clarity, control, and educational value.
- **AST representation**: An object-oriented hierarchy with the visitor pattern provides a clear structure while teaching an important design pattern.
- **Memory management**: Using `std::unique_ptr` for AST nodes demonstrates modern C++ practices while ensuring efficient and safe memory management.
- **Serialization format**: JSON with a well-defined schema offers the best combination of human readability and language independence.
- **Error reporting**: Comprehensive source location tracking and contextual error messages are essential for educational value.
- **Documentation**: Clear architectural overviews, API documentation, and tutorials should accompany the implementation.
- **Integration flexibility**: The dual library/executable approach maximizes flexibility for various backend implementations.

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 what the frontend must accomplish:

4.1.1 Core Parsing Functionality

1. **Lexical Analysis**: The frontend must tokenise `tinyC` source code according to the language specification, correctly identifying all token types (keywords, identifiers, literals, operators, etc.).
2. **Syntax Analysis**: The frontend must parse the token stream according to the `tinyC` grammar rules, verifying syntactic correctness.
3. **AST Construction**: Upon successful parsing, the frontend must construct a well-structured abstract syntax tree that accurately represents the parsed program.

4. **Source Location Tracking**: Each node in the AST must include precise source location information (filename, line, column) to facilitate error reporting and debugging.

4.1.2 Error Handling

1. **Error Detection**: The frontend must detect and report lexical, syntactic, and basic semantic errors.
2. **Helpful Error Messages**: Error messages must be clear, informative, and include relevant source location information.

4.1.3 Interface Options

1. **Library Interface**: The frontend must be usable as a C++ library, providing direct access to AST classes and parsing functionality.
2. **Command-Line Interface**: The frontend must provide a standalone executable that accepts `tinyC` source files and outputs the AST in JSON format.
3. **JSON Output**: The JSON output must follow a well-defined schema that includes all necessary information from the AST, including source locations.

4.2 Non-functional Requirements

The non-functional requirements define quality attributes that determine how the system should behave:

4.2.1 Performance

1. **Parsing Efficiency**: The parser should handle typical `tinyC` programs efficiently, with parsing time proportional to input size.
2. **Memory Usage**: Memory consumption should be reasonable, avoiding unnecessary duplication of data.

4.2.2 Usability

1. **Easy Integration**: The frontend should be straightforward to integrate into student projects, with minimal dependencies.
2. **Comprehensive Documentation**: The API, JSON format, and usage examples must be thoroughly documented.

3. **Platform Independence**: The frontend should work on major platforms used by students (Windows, macOS, Linux).

4.2.3 Maintainability

1. **Code Clarity**: The implementation should be clean, well-structured, and follow C++ best practices.
2. **Modular Design**: The system should be modular to allow for future extensions and modifications.
3. **Version Control**: The codebase should be maintained in a version control system with clear commit history.

4.2.4 Educational Value

1. **Readable Implementation**: The code should be readable and instructive, serving as a good example for students.
2. **Testability**: The system should include comprehensive tests that can also serve as usage examples.

4.3 `tinyC` Grammar Analysis

The `tinyC` language is a simplified subset of C designed for educational purposes. Understanding its grammar is essential for designing an effective parser.

4.3.1 Grammar Characteristics

`tinyC` includes the following key language constructs:

- Basic types (int, double, char, void)
- Variables and arrays
- Pointers
- Function declarations and definitions
- Control structures (if-else, while, do-while, for, switch)
- Expressions with C-like operator precedence
- Struct definitions

The grammar can be generally classified as LL(1), making it suitable for predictive recursive descent parsing with some transformations. The parsing table generator tool developed by Ing. Tomáš Pecka (available at `pages.fit.cvut.cz/peckato1/parsingtbl`) was used to verify the LL(1) properties of the grammar and to generate templates for recursive descent parsing functions. This tool proved invaluable for identifying and resolving grammar conflicts.

4.3.2 Grammar Challenges

Several aspects of the `tinyC` grammar presented challenges for LL(1) parsing:

1. **Expression Parsing**: The grammar for expressions with multiple precedence levels required careful factoring to eliminate left recursion while preserving operator precedence.
2. **Type Declarations**: The grammar for type declarations, especially pointer types and function pointers, required special attention to handle the various forms correctly.
3. **Statement Parsing**: The grammar for statements, particularly those with optional components like the for-loop's initialization, condition, and update expressions, needed careful design to ensure correct parsing.
4. **Left Recursion**: Several productions in the original grammar contained left recursion, which needed to be eliminated for LL(1) parsing.
5. **Common Prefixes**: Some productions had common prefixes, requiring left factoring to make the grammar suitable for predictive parsing.

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.

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.

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.

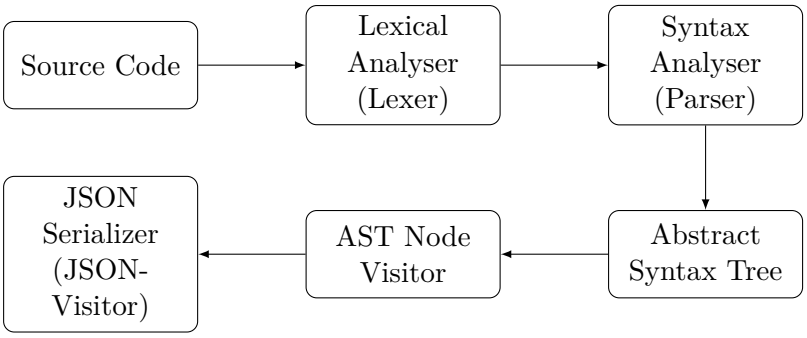


Figure 4.1 High-level architecture of the `tinyC` compiler frontend

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. **AST Node Visitor**: The visitor pattern is used to traverse the AST for serialization.
5. **JSON Serializer**: This component converts the AST to a JSON representation following a well-defined schema available in Appendix B.

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.

4.5.1 Node Structure

The AST consists of various node types that represent different language constructs in `tinyC`. Each node contains specific fields relevant to its language construct and provides methods to access these fields. The design prioritizes simplicity for JSON parsing, allowing students to easily work with the AST representation regardless of their implementation language.

Key node categories include:

- **Declaration Nodes**: Represent variable, function, and struct declarations
- **Type Nodes**: Represent primitive types, named types, and pointer types

- **Expression Nodes**: Represent literals, identifiers, binary and unary operations, etc.
- **Statement Nodes**: Represent blocks, conditionals, loops, etc.

4.5.2 Memory Management

The AST uses `std::unique_ptr` for managing node ownership, ensuring that:

1. Memory is automatically managed, preventing leaks
2. Ownership semantics are clear: each node can have only one owner
3. There is no overhead compared to raw pointers

This approach aligns with modern C++ best practices and provides a good balance between safety and performance.

4.5.3 Visitor Pattern Implementation

The visitor pattern is implemented to allow operations on the AST without modifying the node classes. 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.

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
- The line number (1-based)
- The column number (1-based)

This information is preserved throughout the compilation process and included in the JSON output, as we'll see in Section 5.3.

4.6 API Design

The `tinyC` compiler frontend provides two main interfaces: a C++ library API and a command-line interface.

```
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

4.6.1 Library API

The library API allows direct integration of the `tinyC` frontend into C++ projects. The main entry points are:

1. **Lexer Class**: For tokenizing source code, as shown in Listing 4.4:
2. **Parser Class**: For parsing tokens into an AST, as shown in Listing 4.5:
3. **Visitor Classes**: For traversing and operating on the AST, as shown in Listing 4.6:

This API design allows for flexible use of the frontend, enabling students to:

- Directly work with the AST classes in their backend implementations

```
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

```
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

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

Code listing 4.5 Parser class interface

- Implement their own visitors for custom AST operations
- Control the parsing process step by step

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:

```
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

1. **Lexical Analysis Mode**, as shown in Listing 4.7: This mode tokenizes

```
tinyC-compiler --lex file.tc
```

Code listing 4.7 Lexical analysis mode command

   the input file and outputs the tokens.

2. **Parsing Mode**, as shown in Listing 4.8: This mode parses the input file

```
tinyC-compiler --parse file.tc
```

Code listing 4.8 Parsing mode command

   and outputs the AST as JSON.

3. **Interactive Mode**, as shown in Listing 4.9: This mode provides an interactive shell for entering `tinyC` code and seeing the resulting tokens or AST.

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. The format follows a well-defined schema with the following key characteristics:

```
tinyC-compiler
```

Code listing 4.9 Interactive mode command

1. **Node Type Identification**: Each node includes a `nodeType` field that identifies its type.
2. **Hierarchical Structure**: The JSON structure mirrors the hierarchical structure of the AST.
3. **Source Locations**: Each node includes location information with filename, line, and column.
4. **Type-Specific Fields**: Each node type includes fields specific to that language construct.

Example JSON output for a simple variable declaration is shown in Listing 4.10:

The complete JSON schema is provided in Appendix B, which defines the structure for all node types supported by the frontend.

```
{
  "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 4.10 Example JSON output for a variable declaration

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:

1. **Standardized Header Metadata:**
   - `// tinyC TEST` marker indicating a test file
   - `// 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)
2. Actual TinyC code following the metadata section

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 provides sophisticated test execution and validation:

Flexible Test Selection

The runner supports targeted testing through:

- Running individual tests by number (`--test` flag)
- Running a range of tests (`--range` flag)
- Running all available tests by default

Robust Output Validation

For successful parses, the runner performs:

- JSON schema validation against a formal specification
- Structural comparison between expected and actual AST structures
- Smart comparison that focuses on semantic equivalence rather than exact string matching
- Proper handling of source location information (which can vary but must follow the correct format)

Error Detection Validation

For error tests, the runner:

- Verifies that the appropriate error type is reported (lexical or syntax)
- Checks for informative error messages
- Validates that the parser exits with a non-zero code for errors

Detailed Reporting

The test runner provides:

- Clear test-by-test reports showing pass/fail status
- Detailed error diagnostics for failed tests
- Comparison of expected vs. actual outputs with previews
- Aggregated 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:

- Validates node types and required fields
- Ensures proper nesting of AST components
- Verifies that source location information is present and properly formatted

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
- Creating expected JSON outputs for validation
- Maintaining a consistent test nomenclature and organization
- Supporting 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.

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

- Detailed error reporting guides students toward correct implementations
- The test framework itself demonstrates good testing practices
- Students can extend the test suite with their own tests as they implement additional features

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 comprehensive 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 robust implementation, clear object-oriented design, comprehensive 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, comprehensive documentation, and dual interface approach make it well-suited for educational purposes, while its robust implementation and thorough 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 universal 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 robust lexical analyzer capable of accurately tokenizing TinyC source code with precise 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 comprehensive visitor pattern implementation
- Creation of both a direct C++ library interface and a standardized JSON output format for language-agnostic consumption
- Development of an extensive test suite that verifies the correctness of all components

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 a comprehensive 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**: Chapters 4 and 5 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 both the Evaluation chapter 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 comprehensive 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 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.

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.

While there are opportunities for enhancement, the current implementation successfully fulfills its primary objective: providing a reliable, well-designed, and thoroughly 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*⟩ ::= ⟨*PROGRAM_ITEM*⟩ ⟨*PROGRAM*⟩ | ε

⟨*PROGRAM_ITEM*⟩ ::= ⟨*NON_VOID_TYPE*⟩ `identifier` ⟨*NON_VOID_DECL_TAIL*⟩
| `void` ⟨*VOID_DECL_TAIL*⟩ | ⟨*STRUCT_DECL*⟩ | ⟨*FUNPTR_DECL*⟩

⟨*NON_VOID_DECL_TAIL*⟩ ::= ⟨*VARIABLE_TAIL*⟩ | ⟨*FUNCTION_DECLARATION_TAIL*⟩

⟨*VOID_DECL_TAIL*⟩ ::= `identifier` ⟨*FUNCTION_DECLARATION_TAIL*⟩
| ⟨*STAR_PLUS*⟩ `identifier` ⟨*FUNC_OR_VAR_TAIL*⟩

⟨*FUNC_OR_VAR_TAIL*⟩ ::= ⟨*VARIABLE_TAIL*⟩ | ⟨*FUNCTION_DECLARATION_TAIL*⟩

⟨*VARIABLE_TAIL*⟩ ::= ⟨*OPT_ARRAY_SIZE*⟩ ⟨*OPT_INIT*⟩ ⟨*MORE_GLOBAL_VARS*⟩
`;`

⟨*FUNCTION_DECLARATION_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`

⟨*STATEMENT*⟩ ::= ⟨*BLOCK_STMT*⟩ | ⟨*IF_STMT*⟩ | ⟨*SWITCH_STMT*⟩ |
⟨*WHILE_STMT*⟩ | ⟨*DO_WHILE_STMT*⟩ | ⟨*FOR_STMT*⟩ | ⟨*BREAK_STMT*⟩
| ⟨*CONTINUE_STMT*⟩ | ⟨*RETURN_STMT*⟩ | ⟨*EXPR_STMT*⟩

⟨*BLOCK_STMT*⟩ ::= `{` ⟨*STATEMENT_STAR*⟩ `}`

⟨*STATEMENT_STAR*⟩ ::= ⟨*STATEMENT*⟩ ⟨*STATEMENT_STAR*⟩ | ε

⟨*IF_STMT*⟩ ::= `if (` ⟨*EXPR*⟩ `)` ⟨*STATEMENT*⟩ ⟨*ELSE_PART*⟩

⟨*ELSE_PART*⟩ ::= `else` ⟨*STATEMENT*⟩ | ε

⟨*SWITCH_STMT*⟩ ::= `switch (` ⟨*EXPR*⟩ `) {` ⟨*CASE_DEFLT_STMT_STAR*⟩ `}`

⟨*CASE_DEFLT_STMT_STAR*⟩ ::= ⟨*CASE_STMT*⟩ ⟨*CASE_DEFLT_STMT_STAR*⟩ | ε | ⟨*DEFAULT_CASE*⟩ ⟨*CASE_STMT_STAR*⟩

⟨*CASE_STMT_STAR*⟩ ::= ⟨*CASE_STMT*⟩ ⟨*CASE_STMT_STAR*⟩ | ε

⟨*CASE_STMT*⟩ ::= `case integer_literal :` ⟨*CASE_BODY*⟩

⟨*CASE_BODY*⟩ ::= ⟨*STATEMENT_STAR*⟩

⟨*DEFAULT_CASE*⟩ ::= `default :` ⟨*CASE_BODY*⟩

⟨*WHILE_STMT*⟩ ::= `while (` ⟨*EXPR*⟩ `)` ⟨*STATEMENT*⟩

⟨*DO_WHILE_STMT*⟩ ::= `do` ⟨*STATEMENT*⟩ `while (` ⟨*EXPR*⟩ `) ;`

⟨*FOR_STMT*⟩ ::= `for (` ⟨*OPT_EXPR_OR_VAR_DECL*⟩ `;` ⟨*OPT_EXPR*⟩ `;` ⟨*OPT_EXPR*⟩ `)` ⟨*STATEMENT*⟩

⟨*OPT_EXPR_OR_VAR_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_CALL_INDEX_MEMBER_POST*⟩

⟨*E_CALL_INDEX_MEMBER_POST*⟩ ::= ⟨*F*⟩ ⟨*E_CALL_IDX_MEM_POST_Prime*⟩

⟨*E_CALL_IDX_MEM_POST_Prime*⟩ ::= ⟨*E_CALL*⟩ ⟨*E_CALL_IDX_MEM_POST_Prime*⟩
| ⟨*E_INDEX*⟩ ⟨*E_CALL_IDX_MEM_POST_Prime*⟩ | ⟨*E_MEMBER*⟩
⟨*E_CALL_IDX_MEM_POST_Prime*⟩ | ⟨*E_POST*⟩ ⟨*E_CALL_IDX_MEM_POST_Prime*⟩
| 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_POST*⟩ ::= `++` | `--`

⟨*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]. [N.d.]. [visited on 2025-04-18]. Available from: `https://runestone.academy/ns/books/published/thinkcspy/GeneralIntro/FormalandNaturalLanguages.html`.
3. DRAGAN, Feodor F. *Context-Free Grammars and Pushdown Automata* [online]. [N.d.]. [visited on 2025-04-18]. Available from: `https://www.cs.kent.edu/~dragan/ThComp/lect07-2.pdf`. Accessed 2025-04-18.
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/`.

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. MEYERS, Scott. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014.

29. STROUSTRUP, Bjarne. *The C++ Programming Language*. Addison-Wesley, 2013.

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

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

32. BABEL TEAM. ESTree Specification. *GitHub repository*. 2015. Available also from: `https://github.com/estree/estree`.

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

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

35. 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/`.

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

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

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

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

40. 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.

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

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

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

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

45. MICROSOFT. Language Server Protocol Specification. 2016. Available also from: https://microsoft.github.io/language-server-protocol/specifications/specification-current/.

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

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

48. 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.

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

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

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

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

53. 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.

54. 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.txt` ..... stručný popis obsahu média
  - `exe` ..... adresář se spustitelnou formou implementace
  - `src`
    - `impl` ..... zdrojové kódy implementace
    - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `text` ..... text práce
    - `thesis.pdf` ..... text práce ve formátu PDF