Lecture 1: Parsers

CESE4085 Modern Computer Architecture Course
Part 2, L6.1
Carlo Galuzzi

TUDelft

LECTURE CONTENTS

- **Introduction to Parsing & Its Role in Compilation**
- **Scanners vs. Parsers**
- **Syntax Representation: Grammars & Parse Trees**
- **Parsing Techniques: Top-Down vs. Bottom-Up**
- **LR Parsing & Parsing Tables**
- **Parsing Challenges in different Architectures**
- **AI & Future Trends in Parsing**
- **Key Takeaways**
- **Conclusion**

LEARNING GOALS:

By the end of this lecture, among other goals, you will be able to:

- **Understand the role of parsing in compilation** and key parsing techniques.
- **Differentiate between top-down and bottom-up parsing** methods.
- **Interpret parsing tables and shift-reduce parsing.**
- **Analyze parsing challenges for different architectures.**
- **Explore AI-driven parsing and future trends.**

EXTRA READING MATERIAL IN BRIGHSPACE

Finite Automata and Their Role in Scanners for Compilation

Table of Contents

1. Introduction

Finite automata serve as fundamental computational models in various fields, especially in compiler design. They provide a mathematical framework for recognizing patterns in text, making them indispensable in lexical analysis—the first stage of compilation. The efficiency and correctness of lexical analysis are critical because it directly affects subsequent compilation stages such as parsing and semantic analysis.

Lexical analysis involves scanning source code and breaking it into tokens, which represent keywords, identifiers, literals, and operators. This process ensures that the syntax and structure of the program adhere to the rules of the programming language. Finite automata, through their deterministic and non-deterministic models, provide a structured approach to recognizing these tokens efficiently. By implementing these automata, lexical analyzers can swiftly process source code and prepare it for syntactic and semantic validation.

One of the key advantages of using finite automata in lexical analysis is their ability to represent **regular languages**, which define patterns for keywords, operators, and identifiers in a programming language. Regular expressions, commonly used in defining lexical rules, can be directly translated into finite automata, ensuring a smooth and systematic approach to token recognition.

For example, a keyword such as `if` in a programming language can be represented using a finite automaton that transitions from a start state to an accepting state when the characters `i` and `f` are encountered sequentially. This deterministic process ensures that lexical analyzers operate efficiently, making compilers faster and more reliable.

Understanding finite automata is crucial for compiler engineers as it lays the groundwork for designing robust lexical analyzers. This document provides an introductory analysis of finite automata, their types, and how they contribute to the efficiency of lexical analysis in compilers. We also discuss practical implementations and optimization techniques that make finite automata more efficient in real-world applications.

Parsers in the Compilation Process for RISC-V Architectures

Table of Contents

1. Introduction

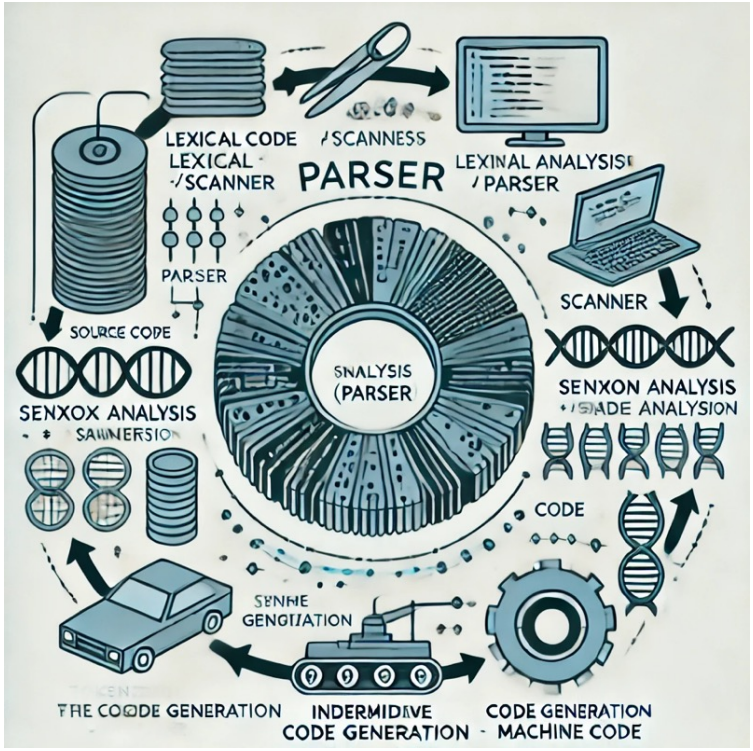
Parsing is a fundamental step in the compilation process. It takes the tokenized input from the scanner and structures it into a hierarchical form—usually a **parse tree** or an **abstract syntax tree (AST)**—which represents the syntactic structure of the source program. This process ensures that the program adheres to the grammatical rules of the programming language and prepares it for semantic analysis and code generation.

For RISC-V architectures, parsing is particularly important in generating efficient low-level code. The way constructs are parsed and transformed impacts the generated assembly instructions, influencing performance and optimization opportunities within an LLVM-based compiler toolchain. The efficiency of parsing can determine how well high-level constructs translate into optimized machine code, ensuring minimal execution overhead and effective utilization of the RISC-V instruction set.

Parsing also plays a role in software security and reliability. A well-designed parser ensures that syntactic errors, ambiguities, and unintended behavior are detected early, preventing possible vulnerabilities. Many modern compilers integrate defensive parsing techniques to prevent attacks such as injection-based exploits in interpreters and compilers.

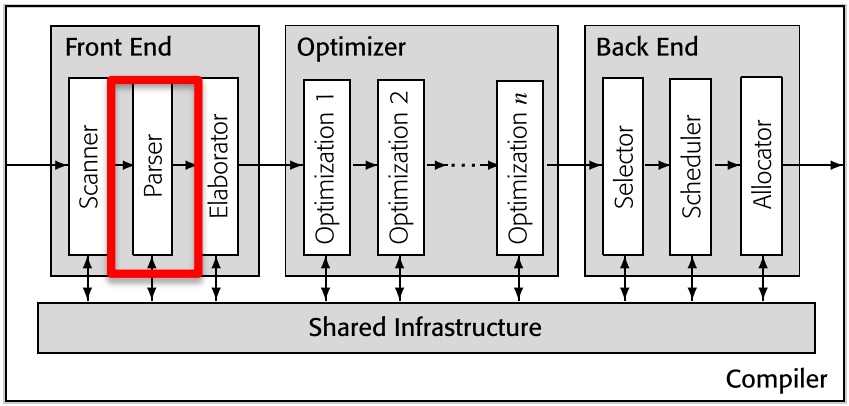
Furthermore, parsers facilitate multiple stages of the compilation process, including macro expansion, type checking, and intermediate representation (IR) generation. The ability to recognize and structure program constructs influences the effectiveness of compiler optimizations such as **constant folding**, **dead code elimination**, and **loop unrolling**—all of which are crucial for improving execution efficiency on RISC-V processors.

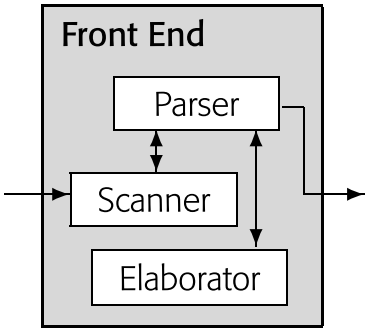
PARSERS


LEXICAL CODE
LEXICAL
PARSER
PARSER
SOURCE CODE
SCANNER
SEMANTIC ANALYSIS
CODE
CODE GENERATION
INTERMEDIATE
CODE GENERATION
CODE GENERATION
MACHINE CODE

WHAT IS PARSING? (1)

The **parser (or syntax analyzer)** is the second major step in compilation. It takes the stream of tokens produced by the scanner and organizes them into a structured representation, typically a **parse tree** or **abstract syntax tree (AST)**. The parser ensures that the sequence of tokens follows the grammatical rules of the programming language, distinguishing valid constructs from syntax errors.





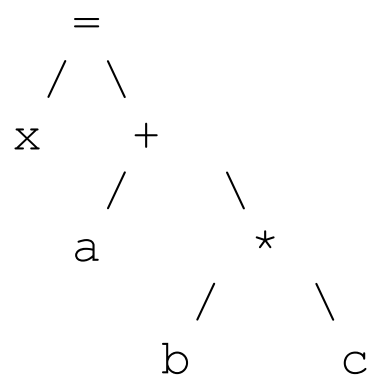
Let's zoom in on what parsing actually does and why it is necessary. Understanding these fundamentals clarifies the difference between a mere sequence of tokens and the well-structured, hierarchical representation required for further compilation steps.

WHAT IS PARSING? (2)

Example: Parsing a simple expression in C

```
x = a + b * c;
```

Parsing converts this into a structured format, like this:



- The **tree structure** shows how operators and operands are grouped together.
- It ensures that `b * c` is calculated **before** adding a.

THE ROLE OF PARSING IN COMPILATION

Where Does Parsing Fit in a Compiler?

A compiler consists of multiple stages, with parsing as a crucial step:

1. **Lexical Analysis (Scanning):** Breaks raw source code into **tokens** (e.g., `if`, `+`, `x`).
2. **Syntax Analysis (Parsing):** Ensures tokens follow **grammar rules** and forms a structured representation.
3. **Semantic Analysis:** Checks for **logical errors** (e.g., using an undeclared variable).
4. **Intermediate Representation (IR) Generation:** Converts the parsed structure into an **intermediate form**.
5. **Optimization & Code Generation:** Produces **efficient machine code** for execution.

SCANNERS AND PARSERS (1)

What is the Difference Between a Scanner and a Parser?

- **Scanner (Lexical Analyzer):** Breaks raw source code into **meaningful tokens**.
- **Parser (Syntax Analyzer):** Uses those tokens to **check syntax and build a structured representation**.

Example Tokenization of `x = a + b * c;`
Scanner Output:

```
IDENTIFIER(x) ASSIGN IDENTIFIER(a) PLUS IDENTIFIER(b)
MULT IDENTIFIER(c) SEMICOLON
```

The **parser** then organizes these tokens into a structured format (like an **Abstract Syntax Tree** or **Parse Tree**).

SCANNERS AND PARSERS (2)

Why is this Important?

- **Scanners remove unnecessary details** (like whitespace) and **speed up parsing**.
- **Parsers enforce syntax rules**, ensuring code correctness before further compilation.

EXPRESSING SYNTAX – GRAMMARS (1)

What is a Grammar?

A **grammar** is a formal set of rules that defines the structure of valid sentences in a programming language. Grammars are used by parsers to determine whether a sequence of tokens forms a syntactically correct program.

Context-Free Grammar (CFG)

A **Context-Free Grammar (CFG)** is a commonly used formalism to describe the syntax of programming languages. A CFG consists of:

1. **A set of non-terminal symbols**: High-level components (e.g., `E`, `T`, `F` in arithmetic expressions).
2. **A set of terminal symbols**: Actual language elements (e.g., `+`, `*`, `id`, `num`).
3. **A set of production rules** that define how non-terminals can be replaced.
4. **A start symbol** from which parsing begins.

EXPRESSING SYNTAX – GRAMMARS (2)

Understanding the Components of the CFG

- $\mathbb{E}$ (Expression): Represents an entire arithmetic expression.
- $\mathbb{T}$ (Term): Represents multiplicative operations (higher precedence than addition and subtraction).
- $\mathbb{F}$ (Factor): Represents atomic elements like numbers or variables.
- Parentheses （） ensure that expressions inside them are evaluated first.

EXPRESSING SYNTAX – GRAMMARS (3)

Example CFG for Arithmetic Expressions:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | id | num
```

- Ensures **operators and operands are used correctly**.
- Allows the **parser to construct valid expressions** from tokens.

Take: `E → E + T | E - T | T`

What does this mean?

- `E` represents an **expression**.
- An expression can be:
  - Another **expression** followed by + and a **term** (`T`).
  - Another **expression** followed by - and a **term** (`T`).
  - Just a **term** (`T`) on its own.

EXPRESSING SYNTAX – GRAMMARS (4)

Take: `T → T * F | T / F | F`

What does this mean?

- T represents a **term**, which consists of **multiplication** or **division**.
- A term can be:
  - Another **term** followed by * and a **factor** (F).
  - Another **term** followed by / and a **factor** (F).
  - Just a **factor** (F) on its own.

EXPRESSING SYNTAX – GRAMMARS (5)

Take: `F → (E) | id | num`

What does this mean?

- `F` represents a **factor**.
- A factor can be:
  - A **parenthesized expression** (E), ensuring operations inside parentheses happen first.
  - An **identifier** (id), which means a variable like x or y.
  - A **number** (num), like 5 or 10.

BREAKING DOWN THE GRAMMAR RULES (1)

How the Grammar Works

Given an expression like:

```
a + b * c
```

- The grammar ensures that `b * c` is evaluated before `a +` because `T` has a higher precedence than `E`.
- The parser follows the rules to create a valid parse tree structure.

So:

Step 1: The expression starts as E.

Step 2: The parser follows the rule `E → E + T`, recognizing `a +` as part of `E`.

Step 3: `b * c` follows `T → T * F`, ensuring **multiplication happens before addition**.

BREAKING DOWN THE GRAMMAR RULES (2)

Why Does This Matter?

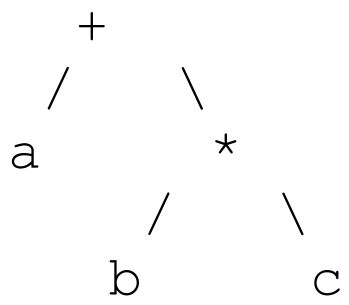
- This grammar **helps the parser understand valid arithmetic expressions**.
- It ensures that **multiplication and division happen before addition and subtraction**.
- It allows parsing **nested expressions with parentheses** (e.g., `(a + b) * c`).

UNDERSTANDING PARSE TREES

What is a Parse Tree?

- A **parse tree** visually represents how grammar rules apply to an expression.
- It helps ensure **correct operator precedence and grouping**.

Example Parse Tree for: `a + b * c`



- This tree structure ensures correct **operator precedence and associativity**: The **multiplication happens first**, then addition.
- The tree **guides further compilation steps**.

HANDLING OPERATOR PRECEDENCE (1)

Why is Operator Precedence Important?

- It ensures that operations are performed **in the correct order**.
- In mathematics and programming, multiplication (*) has a **higher precedence** than addition (+).
- Without precedence rules, ambiguous expressions like a + b * c could be misinterpreted.

Grammar for Operator Precedence

```
E  → E + T | E - T | T
T  → T * F | T / F | F
F  → (E) | id | num
```

- Since `T` expands before `E`, multiplication (*) is **evaluated before** addition (+).

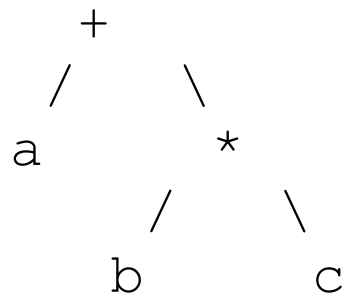
HANDLING OPERATOR PRECEDENCE (2)

Example with Precedence Rules

Expression:

```
a + b * c
```

Correct parsing:



- The parser **ensures correct execution order**.
- This prevents unintended behaviors and incorrect results.

WHAT IS LEFT RECURSION? (1)

Understanding Recursion in Grammar Rules

- **Recursion** happens when a rule refers to itself.
- **Left recursion** occurs when the rule refers to itself at the **beginning**.
- This creates **problems** for top-down parsers because they may enter an **infinite loop**.

Example of Left Recursion:

$$E \rightarrow E + T \mid T$$

What's Wrong With It?

- If the parser starts with E, it **calls itself first** (because of E → E + T).
- This means the parser **keeps calling itself forever** and gets stuck.
- This is called **infinite recursion**, and it **must be avoided**.

WHAT IS LEFT RECURSION? (2)

How to Remove Left Recursion? Rewrite the rule so that it does not immediately refer to itself.

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid \varepsilon$$

How does this fix the problem?

1. **Breaks recursion into two parts**:
   - E **always starts with a T (a term)** instead of calling itself.
   - E' (E-prime) **handles repetition** (like adding more terms).
2. **What does E' → + T E' | ε mean?**
   - It allows + T to be added **multiple times**.
   - ε means **"nothing"**, allowing the recursion to stop when needed.

This **eliminates** infinite recursion and allows parsing to continue.

WHAT IS LEFT RECURSION? (3)

Example: Parsing `a + b + c`

1. Start with E: `E → T E'`
2. Recognize `a` as T: `E → a E'`
3. `E'` expands into `+ T E'`: `E' → + b E'`
4. `E'` expands again: `E' → + c E'`
5. Finally, `E' → ε` (nothing), so parsing **completes successfully**.

Why is this important?

- This change **prevents infinite recursion**.
- It **still allows repeated addition** (a + b + c).
- The **parser can now process expressions correctly**.

WHAT IS BOTTOM-UP PARSING?

How Does Bottom-Up Parsing Work?

- **Starts from the smallest components (tokens) and builds up to larger structures.**
- Opposite of top-down parsing, which starts from the grammar rules.
- More powerful and can handle **more complex languages**.

Steps in Bottom-Up Parsing:

1. **Read the input tokens.**
2. **Combine them into phrases based on grammar rules.**
3. **Reduce those phrases until a complete structure is formed.**

Why is Bottom-Up Parsing Useful?

- Works well with **left-recursive grammars**.
- Used in powerful **parser generators** like *yacc* and *bison*.

WHAT IS SHIFT-REDUCE PARSING? (1)

Understanding the Shift-Reduce Approach

- A type of **bottom-up parsing** that builds the parse tree **from leaves (tokens) to root**.
- Uses a **stack** to keep track of symbols as they are processed.
- Follows two main operations:
  - **Shift:** Read a token from the input and **push it onto the stack**.
  - **Reduce:** Apply a **grammar rule** to replace items on the stack with a non-terminal.

How Does It Work?

- The parser continuously **shifts** tokens onto the stack.
- When a valid pattern is detected, it **reduces** the stack content by applying a grammar rule.
- This process continues until the entire input is parsed successfully.

WHAT IS SHIFT-REDUCE PARSING? (2)

Example: Parsing a + b Using Shift-Reduce

	Stack	Input	Action
1		a + b	Shift a
2	a	+ b	Shift +
3	a +	b	Shift b
4	a + b		Reduce E → a + b
5	E		Accept (Parsing complete)

- The parser **reduces** expressions until the whole statement is processed.
- The **goal** is to reduce everything into the **start symbol (E)**, meaning the input is valid.

Why is Shift-Reduce Parsing Important?

- It is **efficient** and works well for many programming languages.
- Used in **bottom-up parsers**, including **LR parsers**.
- Forms the basis for **automated parser generators** like yacc & bison.

WHAT IS LR PARSING? (1)

Breaking Down LR Parsing

- **LR (Left-to-right, Rightmost derivation)** parsing is a **bottom-up parsing** technique.
- It **reads the input from left to right** and **constructs the parse tree bottom-up**.
- LR parsing handles **more complex grammars** than top-down parsing.

How Does LR Parsing Work?

- **Uses a parsing table** that tells the parser when to **shift, reduce, or accept**.
- **Handles left recursion naturally**.
- **No backtracking required**, making it efficient for real-world compilers.

WHAT IS LR PARSING? (2)

Key Advantages of LR Parsing:

- **Can handle most programming language grammars**.
- **Supports left-recursive rules** without modification.
- **More powerful** than LL(1) parsing (which is used in top-down parsing).
- **Can be implemented efficiently using parsing tables.**

Variants of LR Parsers:

- **LR(0):** Basic version with no lookahead.
- **SLR (Simple LR):** Uses follow sets to resolve conflicts.
- **LALR (Look-Ahead LR):** Memory-efficient version, used in tools like bison.
- **LR(1):** Most powerful but requires larger parsing tables.

WHAT IS LR PARSING? (3)

Why is LR Parsing Important?

- It is used in real-world compilers like **GCC and Clang**.
- It allows **automatic parser generation** from grammars.
- Enables compilers to **process complex language structures efficiently**.

UNDERSTANDING AN LR PARSING TABLE

What is an LR Parsing Table?

- An **LR Parsing Table** is used by **bottom-up parsers** to decide how to process tokens.
- It tells the parser **when to shift (read input), reduce (apply a rule), or accept (finish parsing).**

A Simple Grammar for LR Parsing:

$$S \rightarrow A$$
$$A \rightarrow 0\ A \mid 1$$

- This grammar recognizes **binary numbers that start with 0s and end in 1** (e.g., 0001, 01, 1).
- S is the **starting symbol**.
- A is a **non-terminal** that can expand into 0 A or 1.

LR PARSING TABLE FOR THE SIMPLE GRAMMAR

LR Parsing Table for S → A, A → 0 A | 1

State	0	1	$	Action
0	S2	S3		
1			Accept	
2	S2	S3	R1	
3			R2	

- **Shift (S2, S3):**
  - When 0 is read, **shift** and move to state 2.
  - When 1 is read, **shift** and move to state 3.
- **Reduce (R1, R2):**
  - **R1 (Reduce Rule 1)**: Replace A → 0 A when a valid sequence is found.
  - **R2 (Reduce Rule 2)**: Replace A → 1 (base case of the recursion).
- **Accept:**
  - If the input matches S → A, parsing **succeeds**.

STEP-BY-STEP PARSING EXAMPLE (001 WITH LR TABLE)

Input: `001` **(Valid Binary Sequence)**

Step	Stack	Input	Action
1		001	Shift 0 (State 2)
2	0	01	Shift 0 (State 2)
3	0 0	1	Shift 1 (State 3)
4	0 0 1		Reduce A → 1
5	0 0 A		Reduce A → 0 A
6	0 A		Reduce A → 0 A
7	A		Accept (Parsing Complete)

Key Takeaways:

- Shifting moves tokens onto the stack.
- Reducing replaces sequences with non-terminals.
- The parser follows the table to correctly recognize valid inputs.

PARSING IN LLVM AND RISC-V COMPILATION

How Parsing Works in LLVM

- LLVM uses a **modular parsing system** to process code efficiently.
- The parser converts the **high-level code into an Abstract Syntax Tree (AST)**.
- The AST is then transformed into **LLVM Intermediate Representation (IR)**.
- IR undergoes **optimizations** before generating RISC-V assembly.

Example: LLVM IR for `int add(int a, int b)`

```
define i32 @add(i32 %a, i32 %b) {
entry:
    %sum = add i32 %a, %b
    ret i32 %sum
}
```

- The parser ensures that operations are **structured correctly**.
- The generated LLVM IR is **hardware-independent**, making it easy to optimize for RISC-V.

PARSING IN MODERN COMPILERS

How Do Modern Compilers Use Parsing?

- Most modern compilers use **LR or LALR parsers** because they are efficient.
- Parsing is done in **multiple stages**, often using **Abstract Syntax Trees (ASTs)**.
- **Optimization techniques** improve performance by analyzing the parsed structure.

Why is Parsing Critical in Compiler Design?

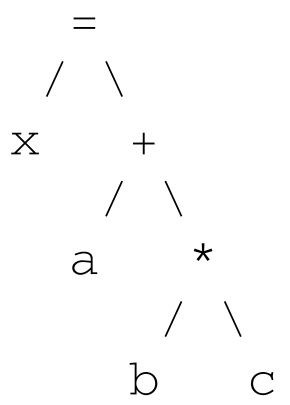
- It converts **high-level source code into structured data**.
- It ensures **correct syntax and program structure** before code generation.
- **Efficient parsing** speeds up compilation and improves error detection.

UNDERSTANDING ABSTRACT SYNTAX TREES (ASTs)

What is an Abstract Syntax Tree (AST)?

- An **AST is a tree representation** of the program's structure.
- Unlike a **parse tree**, it removes unnecessary syntax details.

Example: AST for `x = a + b * c`



Why Use an AST?

- Simplifies further compilation steps.
- Makes optimizations easier.
- Reduces complexity compared to a full parse tree.

PARSING AND OPTIMIZATION IN COMPILATION

Why is Parsing Important for Optimization?

- Parsing **structures code** in a way that enables optimizations.
- **Abstract Syntax Trees (ASTs) and Intermediate Representations (IRs)** are used to apply transformations.
- A well-structured AST allows compilers to detect **redundancies, inline functions, and eliminate dead code**.

Example: Optimization in LLVM IR

Before Optimization:

```
%temp = add i32 %a, 0
ret i32 %temp
```

After Optimization:

```
ret i32 %a
```

- The parser ensures that the **structure is correct**, and the optimizer removes **redundant computations**.

HOW PARSING AFFECTS PERFORMANCE

Parsing Efficiency and Compiler Speed

- **Fast parsing techniques** reduce compilation time.
- **Ambiguous grammars and large syntax trees** can slow down the process.
- **Incremental parsing** (re-parsing only modified parts of the code) is used in modern compilers.

Example: Trade-off Between Accuracy and Speed

Parsing Technique	Accuracy	Speed
LL(1) Predictive Parsing	Low	Fast
LR(1) Parsing	High	Slower
LALR Parsing	Medium	Balanced

Key Takeaways:

- The **choice of parsing strategy** impacts compiler performance.
- **Optimized parsing algorithms** help manage **large codebases efficiently**.

PARSING AND MEMORY MANAGEMENT (1)

Why is Memory Management Important in Parsing?

- Parsing **stores and processes large amounts of structured data**.
- Inefficient memory use can lead to **slow performance and high resource consumption**.
- **Compilers use memory-efficient data structures** like **hash tables, trees, and symbol tables**.

Example: Memory-Efficient Parsing in LLVM

- LLVM uses **pointer-based ASTs** to avoid redundant memory allocation.
- **Garbage collection and reference counting** help manage memory efficiently.

PARSING AND MEMORY MANAGEMENT (2)

Optimization Techniques for Memory-Efficient Parsing

1. **Reusing Parse Trees** instead of regenerating them.
2. **Using AST compression** to store only essential information.
3. **Employing caching mechanisms** to speed up re-parsing.

PARSING FOR DIFFERENT PROGRAMMING LANGUAGES (1)

How Do Parsing Techniques Differ Across Languages?

- Some languages have **simpler grammars**, making parsing easier.
- Others (e.g., C++, Rust) require **complex parsing due to syntax ambiguity**.

Comparison of Parsing Complexity

Language	Parsing Complexity	Parser Type Used
Python	Low	LL(1)
Java	Medium	LALR
C++	High	GLR (Generalized LR)

Example: Parsing Differences in Python vs. C++

- **Python** uses **indentation** to define blocks → simple LL parsing.
- **C++** has **template specialization and overloading**, requiring **more advanced LR parsing techniques**.

PARSING FOR DIFFERENT PROGRAMMING LANGUAGES (2)

Key Takeaways:

- Parsing strategies **must be chosen based on the language's complexity**.
- **More expressive languages** require **stronger parsing methods**.

ADAPTIVE PARSING TECHNIQUES

What is Adaptive Parsing?

- Adaptive parsing allows a compiler to **adjust its parsing strategy** based on the complexity of the input.
- It improves **performance** and **error handling** by dynamically selecting the best parsing method.

Types of Adaptive Parsing

1. **Lazy Parsing** – Only parse necessary parts of the code.
2. **Incremental Parsing** – Re-parses only modified sections of the source code.
3. **Speculative Parsing** – Tries multiple parsing strategies and selects the best one.

Example: Adaptive Parsing in IDEs - Modern **Integrated Development Environments (IDEs)** use adaptive parsing to provide **real-time code analysis and suggestions** without fully compiling the program.

PREDICTIVE PARSING IN MODERN COMPILERS (1)

What is Predictive Parsing?

- A **top-down parsing technique** that makes decisions based on a **lookahead token**.
- Used in **LL(1) parsers**, which determine the correct rule by looking at only **one symbol ahead**.

How It Works:

1. Reads a token from input.
2. Uses a parsing table to determine the next step.
3. Expands the correct grammar rule without backtracking.

PREDICTIVE PARSING IN MODERN COMPILERS (2)

Example: LL(1) Parsing for $E \rightarrow E + T \mid T$

- If the **lookahead token** is +, the parser knows to expand $E \rightarrow E + T$.
- If it is another token, it applies $E \rightarrow T$.

Advantages of Predictive Parsing:

✅ Fast and efficient (no backtracking).
✅ Works well for simple, well-structured grammars.
❌ Cannot handle left-recursive or ambiguous grammars without modifications.

HANDLING CONTEXT-SENSITIVE GRAMMAR (1)

What is Context-Sensitive Grammar?

- A **context-sensitive grammar (CSG)** allows rules to change based on the surrounding context.
- Unlike **context-free grammars (CFGs)**, CSGs can express more **complex language rules**.

HANDLING CONTEXT-SENSITIVE GRAMMAR (2)

Example: Context-Sensitive Rule in C

```
int x;
float x;  // Error: 'x' was already declared as an int
```

- The second declaration is invalid because **the context (previous declaration) changes the meaning of the second x**.

Why Context Sensitivity Matters in Parsing?

- Used for **type checking, scope resolution, and language semantics**.
- Requires **extra processing beyond basic parsing**, often handled in the **semantic analysis phase**.

PARSING AND CODE GENERATION

How Does Parsing Affect Code Generation?

- The parser **structures the input** so it can be converted into an **Intermediate Representation (IR)**.
- This IR is later **translated into machine code**.

Example: Parsing to LLVM IR

Source Code (C):

```
int add(int a, int b) {
  return a + b;
}
```

Generated LLVM IR:

```
define i32 @add(i32 %a, i32 %b) {
entry:
    %sum = add i32 %a, %b
    ret i32 %sum
}
```

The parser ensures that each operation is **correctly structured** before code generation.

1) PARSING FOR HIGH-PERFORMANCE COMPUTING

Challenges in Parsing for HPC

- **Large codebases:** HPC applications involve **millions of lines of code**.
- **Parallel execution:** Code must be optimized for multi-threading.
- **Memory efficiency:** Parsers must minimize overhead.

Parsing Techniques Used in HPC Compilers

1. **Parallel Parsing:** Splitting parsing work across multiple cores.
2. **Incremental Compilation:** Re-parsing only modified sections of the code.
3. **Optimized IR Generation:** Ensuring the **intermediate representation** is suited for HPC architectures like **RISC-V**.

Example: Parsing for Vectorized Operations

- Modern compilers use parsing techniques to **detect loops** and **transform them into vectorized instructions** for better performance.

2) PARSING FOR EMBEDDED SYSTEMS

Challenges of Parsing in Embedded Systems

- **Limited computing resources:** Embedded devices have minimal processing power.
- **Low memory availability:** Parsers must be optimized to use **very little RAM**.
- **Real-time constraints:** Parsing must be **fast and predictable**.

Parsing Strategies for Embedded Systems

- **Lightweight parsers** (e.g., using LL(1) or LALR parsers).
- **Token stream compression** to reduce memory footprint.
- **Incremental parsing** to process only necessary code sections.

Example: Parsing in an Embedded Compiler

- Many embedded compilers use **specialized parsers** to handle real-time constraints efficiently, ensuring fast code execution on hardware like **RISC-V microcontrollers**.

3) PARSING CHALLENGES IN RISC-V

Why is Parsing Important for RISC-V?

- RISC-V is an **open-source processor architecture**, meaning **many different compilers** are developed for it.
- Parsing must ensure **compatibility with multiple instruction sets and optimizations**.

Challenges in Parsing for RISC-V

1. **Custom Instruction Sets:** Different vendors extend RISC-V with unique instructions.
2. **Optimized Code Generation:** Parsing must produce **efficient machine code** for low-power devices.
3. **Handling Inline Assembly:** RISC-V allows inline assembly, which must be parsed correctly.

For example, LLVM's **RISC-V backend** parses assembly-like constructs within C/C++ code to ensure correct execution.

4) PARSING FOR MULTI-CORE ARCHITECTURES

Why is Parsing Different for Multi-Core Systems?

•Multi-core processors **execute multiple instructions in parallel**.

•Parsing must **analyze and optimize** code to **take advantage of parallel execution**.

Techniques for Multi-Core Parsing

1. **Parallel Parsing:** Breaking code into sections and parsing them in parallel.
2. **Speculative Parsing:** Predicting code structure ahead of time to reduce parsing delays.
3. **Thread-Aware Parsing:** Ensuring correct **synchronization** and **memory consistency**.

For example, compilers for **multi-threaded applications** use parsing strategies to **identify parallelizable sections of code** and optimize execution.

FUTURE DIRECTIONS IN PARSING ALGORITHMS

How is Parsing Evolving?

- Traditional parsing methods work well, but newer challenges require innovative solutions.
- Advances in **adaptive parsing, parallel parsing, and AI-driven techniques** are shaping the future.

Emerging Trends

1. **Parallel and Distributed Parsing:** Handling large-scale codebases efficiently.
2. **Incremental and Just-In-Time Parsing:** Faster compilation by only re-parsing modified sections.
3. **Self-Learning Parsers:** AI-based techniques that learn from past parsing experiences.

AI AND MACHINE LEARNING IN PARSING

How Can AI Improve Parsing?

- AI-driven parsers can **predict and optimize parsing decisions**.
- Machine learning models help detect **syntax patterns and errors** in a more intelligent way.

Key Applications of AI in Parsing

1. **Error Prediction & Correction:** AI can suggest **fixes for syntax errors** before compilation.
2. **Grammar Learning:** AI can **analyze and generate grammars** for new programming languages.
3. **Automated Optimization:** AI-driven techniques **predict the most efficient parsing approach** based on past data.

Example: AI-Assisted Parsing in IDEs

- Modern IDEs (e.g., VS Code, JetBrains) use AI-assisted parsing to provide **real-time code suggestions** and **error detection**.

THE FUTURE OF COMPILER PARSING

What Will Parsing Look Like in the Next Decade?

- **Traditional parsing techniques will continue to evolve.**
- The rise of **AI-assisted parsing** and **self-optimizing compilers** will shape the next generation of parsing methods.

Future Innovations in Parsing

1. **Neural Parsers:** AI-based models that learn how to parse from vast datasets.
2. **Hybrid Parsing Techniques:** Combining **rule-based and AI-based parsing**.
3. **Interactive and Incremental Parsing:** Parsing that adjusts dynamically during program execution.

Impact on High-Performance and Embedded Systems

- **Faster parsing speeds** for real-time systems.
- **Smarter parsing** for complex instruction sets (e.g., **RISC-V extensions**).

SUMMARY OF PARSING TECHNIQUES

Key Parsing Strategies Discussed

- **Top-Down Parsing:** LL(1), recursive descent, predictive parsing.
- **Bottom-Up Parsing:** Shift-reduce, LR(1), LALR(1).
- **Specialized Parsing Techniques:** Context-sensitive parsing, AI-driven parsing, adaptive parsing.

When to Use Different Parsing Methods?

Parsing Type	Use Case
LL(1)	Simple, predictable grammars
LR(1)	More complex grammars, high accuracy
LALR(1)	Balanced between efficiency and power
AI-Based	Self-improving and error-correcting systems

KEY TAKEAWAYS ON PARSING

What Have We Learned?

- **Parsing is a critical stage in compilation**, transforming source code into structured representations.
- **Different parsing methods** (top-down, bottom-up, AI-driven) have **unique advantages**.
- **Efficient parsing** leads to **faster compilation and better optimization** in RISC-V and other architectures.

Why Parsing Matters for Engineers

- Knowing how parsers work **helps optimize compiler design**.
- Understanding parsing **improves debugging, error handling, and performance tuning**.

See you at the next Lecture!

Don’t forget to consult the Q&A blog on Brightspace!