Lesson 16: Compiler Design and Parsing

Compiler design and parsing are fundamental concepts in the field of computer science, particularly in the development of programming languages and software systems. Compiler design involves the construction of efficient and accurate compilers, which are software programs responsible for translating high-level programming languages into executable machine code. Parsing, on the other hand, focuses on the analysis and interpretation of the structure and syntax of programming languages.

The process of compiler design encompasses various stages, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, optimization, and code generation. Each stage plays a crucial role in transforming human-readable source code into machine-executable instructions. The primary objective of compiler design is to ensure that the resulting compiled program is correct, efficient, and adheres to the intended semantics of the programming language.

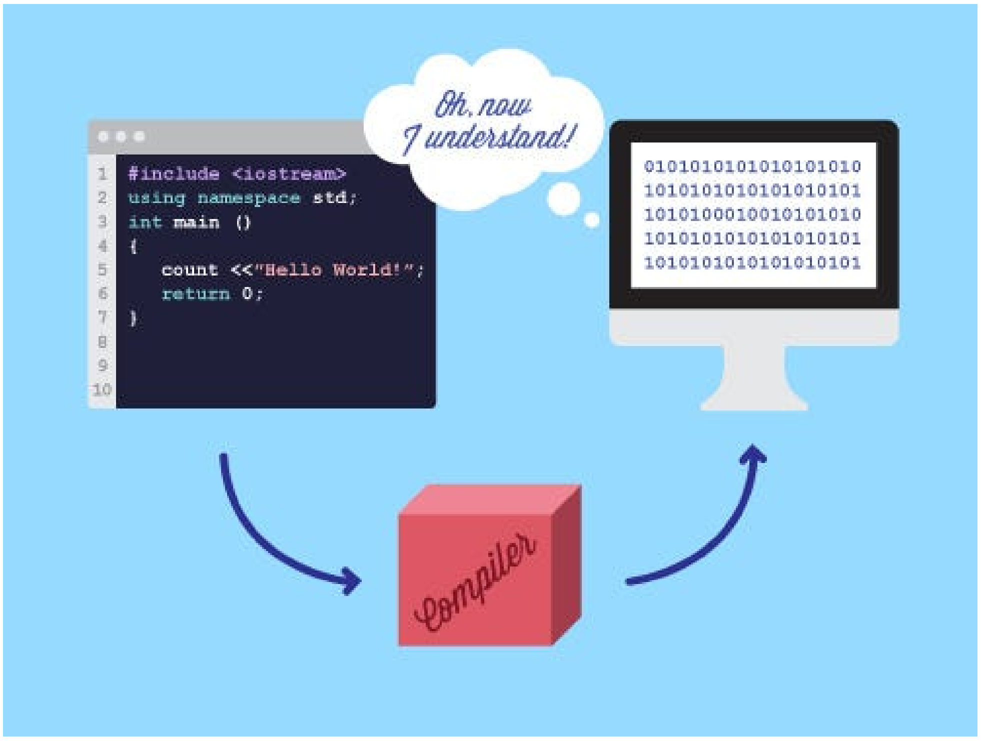
Parsing, a key component of compiler design, involves the analysis of the syntax or structure of a programming language. It verifies whether the given program follows the specified grammar rules and generates a parse tree or abstract syntax tree (AST) that represents the hierarchical structure of the program. Parsing ensures that the program is well-formed and aids in subsequent stages of the compilation process, such as semantic analysis and code generation.

To perform parsing, various parsing techniques are employed, including top-down parsing (such as recursive descent parsing and LL parsing) and bottom-up parsing (such as LR parsing and LALR parsing). These techniques utilize formal grammars, such as context-free grammars, to define the syntactic rules of a programming language and guide the parsing process.

The study of compiler design and parsing is essential for understanding how programming languages are processed and executed by computers. It involves a deep understanding of language theory, formal grammars, parsing algorithms, and code generation techniques. Compiler design and parsing have practical applications in the development of programming languages, optimizing code performance, and ensuring the reliability and efficiency of software systems.

Compiler structure and phases

A compiler is a software tool that translates high-level programming code, written in a programming language, into machine code that can be executed directly by a computer or a target platform. It acts as an intermediary between human-readable source code and the low-level instructions that the computer's processor can understand. The main purpose of a compiler is to automate the process of converting high-level code, which is easier for humans to understand and write, into a format that the computer can execute efficiently. It performs several crucial tasks during this translation process, including lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation.



```
#include <iostream>
using namespace std;
int main ()
{
   count <<"Hello World!";
   return 0;
}
```

```
01010101010101010
10101010101010101
10101000100101010
10101010101010101
10101010101010101
```

Lexical Analysis:
The compiler's lexical analyzer, or lexer, scans the source code to break it down into a sequence of tokens. Tokens represent the smallest meaningful units in the programming language, such as keywords, identifiers, constants, and operators.

Syntax Analysis:
The syntax analyzer, or parser, ensures that the tokens produced by the lexer conform to the grammar rules of the programming language. It constructs a hierarchical structure called a parse tree or abstract syntax tree (AST) that represents the syntactic structure of the program.

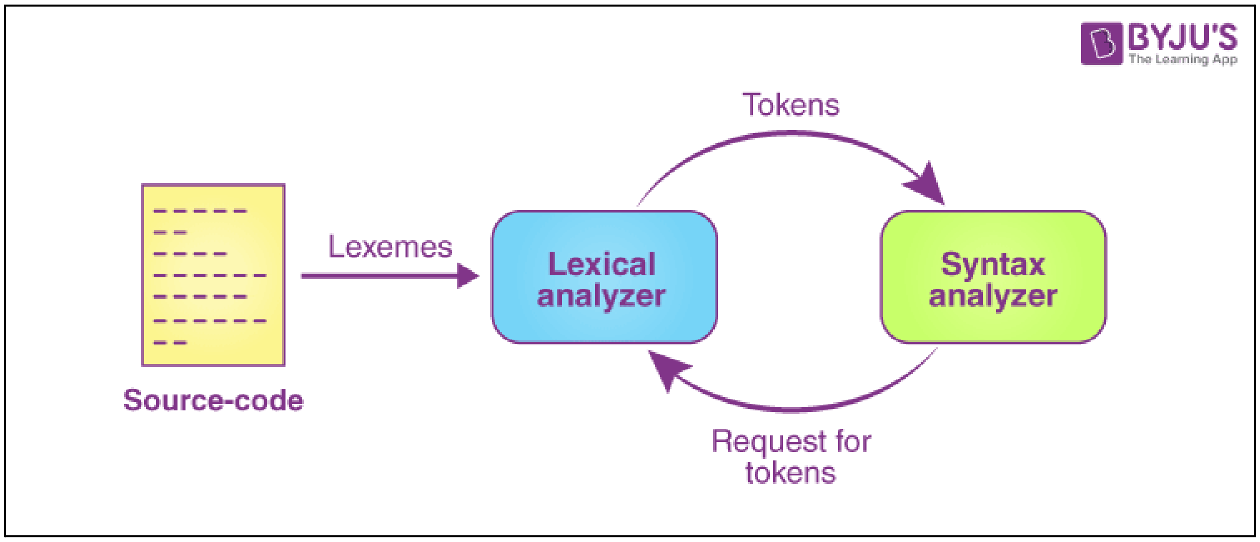
Semantic Analysis:

The semantic analyzer checks the program's meaning and correctness by examining the parse tree or AST. It enforces the language's semantics, such as type compatibility, variable declarations, scoping rules, and other language-specific constraints.

Code Optimization:
The compiler's optimizer analyzes the intermediate representation of the program, obtained from the previous phases, and applies various optimization techniques. These techniques aim to improve the efficiency and performance of the compiled code by reducing execution time, minimizing memory usage, and eliminating redundant or inefficient code.

Code Generation:
In the final phase, the compiler translates the optimized intermediate representation into machine code specific to the target platform or architecture. The code generator emits instructions that the computer's processor can execute directly, such as assembly language instructions or binary machine code.

Once the compilation process is complete, the resulting executable code can be executed by the computer, resulting in the desired program behavior. The compiled program is typically more efficient and faster than an interpreted or unoptimized version.



Compilers are essential tools in software development, enabling programmers to write code in high-level languages and compile them into executable programs for various platforms. They play a vital role in the creation of applications, system software, and firmware across a wide range of domains and industries.

Lexical analysis and parsing

Lexical analysis and parsing are two fundamental phases in the process of compiling a programming language. They are responsible for analyzing and understanding the structure and syntax of the source code.

Lexical Analysis:

Lexical analysis, also known as scanning, is the first phase of the compilation process. It involves breaking down the source code into a stream of tokens, which are the smallest meaningful units of the programming language. Tokens can include keywords, identifiers, constants, operators, and punctuation marks. The lexical analyzer, also called a lexer or scanner, reads the source code character by character, identifies the tokens, and categorizes them according to their lexical classes. This phase removes any unnecessary whitespace or comments and produces a sequence of tokens as output.

For example, given the source code line:

```
int sum = 10 + 5;
```

The lexical analyzer would generate the tokens:

```
<INT, "int">, <ID, "sum">, <EQ, "=">, <NUM, "10">, <PLUS, "+">, <NUM,
"5">, <SEMICOLON, ";">
```

Parsing:

Parsing is the second phase of the compilation process, following lexical analysis. Its primary objective is to analyze the syntactic structure of the source code and construct a hierarchical representation of its grammar. This hierarchical representation is often represented as a parse tree or an abstract syntax tree (AST).

During parsing, a parser reads the stream of tokens generated by the lexer and checks whether they adhere to the grammar rules of the programming language. The parser applies a set of production rules defined by the language's context-free grammar to identify the structure and relationships between tokens. It verifies the correctness of the code's syntax by ensuring that the tokens can be combined according to the language's grammar rules.

The resulting parse tree or AST represents the structure of the code in a hierarchical manner. Nodes in the tree represent language constructs, such as expressions, statements, and declarations, while edges represent the relationships between these constructs. The parse tree or AST serves as a foundation for subsequent phases of the compilation process, such as semantic analysis, optimization, and code generation.

For example, given the tokens from the lexical analysis phase:

```
<INT, "int">, <ID, "sum">, <EQ, "=">, <NUM, "10">, <PLUS, "+">, <NUM,
"5">, <SEMICOLON, ";">
```

The parser constructs a parse tree or AST that represents the assignment statement **`sum = 10 + 5`**.

Lexical analysis and parsing are crucial steps in the compilation process, as they lay the foundation for understanding the structure and syntax of the source code. They provide the necessary information for subsequent phases to perform semantic analysis, optimization, and ultimately generate the executable machine code. These phases work together to ensure that the code is correct, adheres to the language's syntax rules, and can be executed accurately.

Context-free grammars and parsing algorithms

Context-free grammars (CFGs) and parsing algorithms are essential components of parsing, which is a crucial phase in the compilation process. Context-free grammars provide a formal notation to describe the syntax and structure of programming languages, while parsing algorithms analyze the input based on these grammars to construct parse trees or abstract syntax trees.

Context-Free Grammars:

A context-free grammar is a formal notation consisting of a set of production rules that describe the syntax of a programming language or any other formal language. It consists of terminals, non-terminals, production rules, and a start symbol.

- Terminals: These are the basic symbols or tokens in the language, such as keywords, identifiers, operators, and constants. Terminals cannot be further broken down.

- Non-terminals: These are symbols that represent sets of strings or language constructs. Non-terminals can be expanded into other non-terminals or terminals.

- Production Rules: These rules define how non-terminals can be expanded or replaced by other non-terminals or terminals. They describe the syntactic structure of the language.

- Start Symbol: The start symbol represents the overall language construct being parsed. It is the root of the parse tree or abstract syntax tree.

Parsing Algorithms:

Parsing algorithms are algorithms used to analyze the structure of the input based on the context-free grammar. They take the stream of tokens generated by the lexical analysis phase and determine whether it can be derived from the given grammar. Here are a few commonly used parsing algorithms:

1. Recursive Descent Parsing: Recursive descent parsing is a top-down parsing technique where each non-terminal in the grammar is associated with a parsing function. The parser starts from the start symbol and recursively applies the production rules based on the next token until it matches the entire input. It constructs the parse tree in a depth-first manner.

2. LL Parsing: LL (left-to-right, leftmost derivation) parsing is a family of top-down parsing algorithms that parse the input from left to right and construct a leftmost derivation. LL parsing uses a lookahead symbol to decide which production rule to apply based on a parsing table constructed from the grammar. LL(k) parsing, where k is the number of lookahead symbols, is a widely used variant of LL parsing.

3. LR Parsing: LR (left-to-right, rightmost derivation) parsing is a family of bottom-up parsing algorithms. LR parsers read the input from left to right, construct a rightmost derivation, and build the parse tree in a bottom-up manner. LR parsing uses a stack and a parsing table to determine the appropriate reduction or shift actions. LR(1) and LALR(1) parsing are popular variants of LR parsing.

4. Earley Parsing: Earley parsing is a general parsing algorithm that can handle any context-free grammar, including ambiguous grammars. It uses dynamic programming and builds a set of states to predict, scan, and complete items based on the grammar rules. Earley parsing is known for its ability to handle various language constructs, including those with ambiguity.

These parsing algorithms, along with variations and optimizations, enable the analysis of input based on the context-free grammar. They construct parse trees or abstract syntax trees that represent the structure of the program and serve as a foundation for subsequent phases in the compilation process, such as semantic analysis, optimization, and code generation.