Context Free Grammar

Course Name: Compiler Design
Course Code: CSE331
Level:3, Term:3
Department of Computer Science and Engineering
Daffodil International University

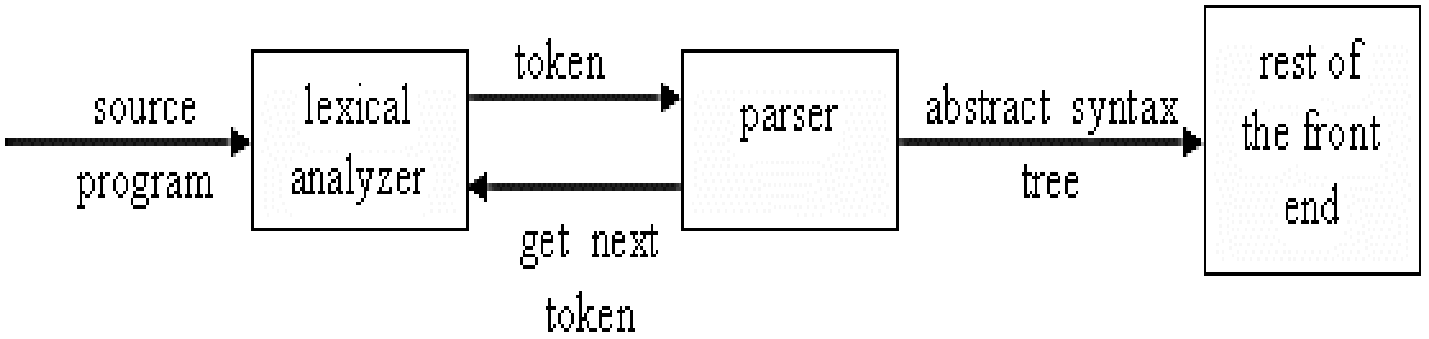
Syntax Analysis

- **Parsing** or **syntactic analysis** is the process of analyzing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar.
- The second phase of compiler and the sometimes also being called as **Hierarchical Analysis**.

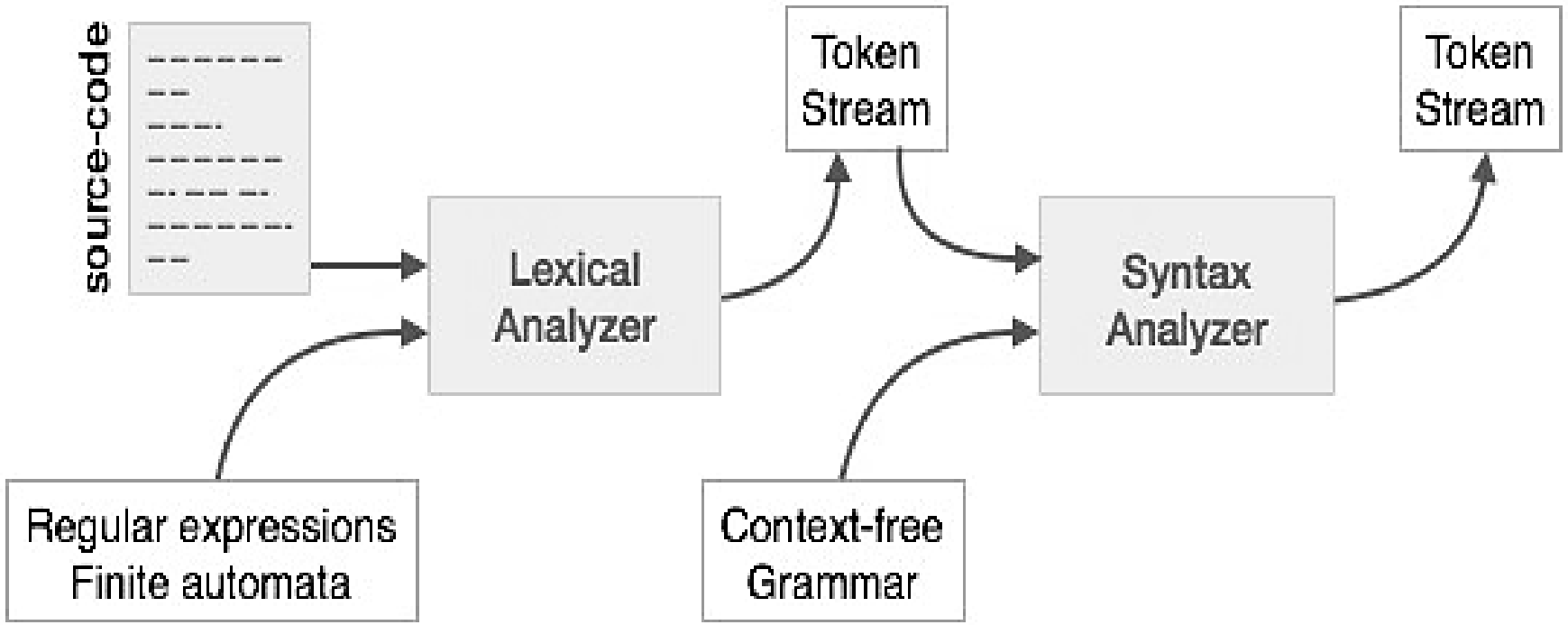
Why Syntax Analysis?

- We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules.
- But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions.
- Regular expressions cannot check balancing tokens, such as parenthesis.
- Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

Syntax Analysis



Detail Diagram



- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.
- This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

CFG

- CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple (N, T, P, S)
where,

- N is a set of non-terminal symbols.
- T is a set of terminals where $N \cap T = NULL$.
- P is a set of rules, P: $N \rightarrow (N \cup T)^*$, i.e., the left-hand side of the production rule P does have any right context or left context.
- S is the start symbol.

CFG: Context Free Grammar

- A context-free grammar has four components: G = (V, Σ, P, S)

✓ A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

✓ A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

✓ A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.

✓ One of the non-terminals is designated as the **start symbol** (S); from where the production begins.

Example of CFG:

- G = (V, Σ, P, S) Where:
- V = { Q, Z, N }
- Σ = { 0, 1 }
- P = { Q → Z | Q → N | Q → Ɛ | Z → 0Q0 |N → 1Q1 }
- S = { Q }

➢This grammar describes palindrome language,

such as: 1001, 11100111, 00100, 1010101, 11111, etc.

The Role of the Lexical Analyzer

- **Roles:**
  - Primary role: Scan a source program (a string) and break it up into small, meaningful units, called tokens.
  - Example: position := initial + rate * 60;
  - Transform into meaningful units: identifiers, constants, operators, and punctuation.
- **Other roles:**
  - Removal of comments
  - Case conversion
  - Removal of white spaces
  - Interpretation of compiler directives or pragmas.
  - Communication with symbol table: Store information regarding an identifier in the symbol table. Not advisable in cases where scopes can be nested.
  - Preparation of output listing: Keep track of source program, line numbers, and correspondences between error messages and line numbers.
- **Why Lexical Analyzer is separate from parser?**
  - Simpler design of both LA and parser.
  - More efficient & portable compiler.

Tokens

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example of tokens:
Type token (id, number, real, . . .)
Punctuation tokens (IF, void, return, . . .)
Alphabetic tokens (keywords)
Keywords; Examples-for, while, if etc.
Identifier; Examples-Variable name, function name etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc.

Example of Non-Tokens:
Comments, preprocessor directive, macros, blanks, tabs, newline etc.

Lexeme: The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.
eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Tokens

- **Examples of Tokens**

Operators	**= + – > ({ := == <>**
Keywords	if while for int double
Numeric literals	43 6.035 -3.6e10 0x13F3A
Character literals	‘a’ ‘~’ ‘\’’
String literals	“3.142” “aBcDe” “\”

- **Examples of non-tokens**

White space	**space(‘ ’) tab(‘\t’) newline(‘\n’)**
Comments	/*this is not a token*/

- Type of tokens in C++:
  - Constants:
    - char constants: ‘a’
    - string constants: “i=%d”
    - int constants: 50
    - float point constants
  - Identifiers: i, j, counter, ……
  - Reserved words: main, int, for, …
  - Operators: +, =, ++, /, …
  - Misc. symbols: (,), {, }, …
- Tokens are specified by **regular expressions**.

```
main() {
   int i, j;
   for (i=0; i<50; i++) {
      printf(“i = %d”, i);
   }
}
```

Lexical Analysis vs Parsing

- There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.
- Simplicity of design is the most important consideration.

  The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.
- Compiler efficiency is improved.

  a separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
- Compiler portability is enhanced.

  Input-device-specific peculiarities can be restricted to the lexical analyzer

More about Tokens, Patterns and Lexemes

- **Token:** a certain classification of entities of a program.
  - four kinds of tokens in previous example: identifiers, operators, constraints, and punctuation.
- **Lexeme:** A specific instance of a token. Used to differentiate tokens. For instance, both position and initial belong to the identifier class, however each a different lexeme.
  - Lexical analyzer may return a token type to the Parser, but must also keep track of "attributes" that distinguish one lexeme from another.
  - Examples of attributes: Identifiers: string, Numbers: value
  - Attributes are used during semantic checking and code generation. They are not needed during parsing.
- **Patterns:** Rule describing how tokens are specified in a program. Needed because a language can contain infinite possible strings. They all cannot be enumerated (calculated/specified)
  - Formal mechanisms used to represent these patterns. Formalism helps in describing precisely (i) which strings belong to the language, and (ii) which do not.
  - Also, form basis for developing tools that can automatically determine if a string belongs to a language.

Lexical errors

fi (a==f(x)) - **fi** is misspelled or keyword? Or undeclared function identifier?

- If **fi** is a valid lexeme for the token id, the lexical analyzer must return the token id to the parser and let some other phase of the compiler - handle the error

How?

i. Delete one character from the remaining input.
ii. Insert a missing character into the remaining input.
iii. Replace a character by another character.
iv. Transpose two adjacent characters.

Lexical Errors and Recovery

- Panic mode error recovery
- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by another
- Transposing two adjacent character

THANK YOU