**Unit 2**

**Introduction to Compiler**

**Programming Language Basics - Lexical Analysis**

The lexical analyzer is the first phase of a compiler. Its purpose is to process the input source code and produce a sequence of meaningful units called **tokens**. Tokens represent key elements of the code, such as keywords, identifiers, operators, and punctuation.

**Key Points:**

1. **What is Lexical Analysis?**
   * It reads the input program (a sequence of characters) and groups these into **lexemes**, which are sequences of characters matching patterns (e.g., if, x, +).
   * For each lexeme, it generates a **token**, a data structure containing:
     + Token type (e.g., keyword, operator).
     + Attribute value (additional information, like variable names).
2. **Ways to Implement a Lexical Analyzer:**
   * **Manual Implementation:**
     + Design diagrams or descriptions for lexemes of each token.
     + Write code to identify and return tokens.
   * **Automated Generation:**
     + Use tools like **Lex** or **Flex**.
     + Specify lexeme patterns using **regular expressions**.
     + Compile these patterns into code that functions as the lexical analyzer.
     + Advantages: Faster implementation and easier modification by updating patterns instead of rewriting the entire program.
3. **Using Automata:**
   * Regular expressions are converted into:
     + **Nondeterministic Finite Automata (NFA)** and
     + **Deterministic Finite Automata (DFA)**.
   * A **driver program** uses the automaton to identify tokens from the input.

**Role of the Lexical Analyzer**

The lexical analyzer plays a critical role in the compilation process.

**Primary Functions:**

1. **Token Generation:**
   * Reads the source code, identifies lexemes, and generates tokens.
   * Passes the token stream to the **parser** for syntax analysis.
2. **Interaction with the Symbol Table:**
   * When identifying identifiers (e.g., variable names), it adds them to the **symbol table** with associated details.

**Additional Tasks:**

1. **Removing Comments and Whitespace:**
   * Removes unnecessary elements, such as comments and extra spaces, while preserving the structure of the program.
2. **Error Reporting:**
   * Tracks line numbers and associates errors with their position in the source code.
   * Some analyzers insert error messages directly into the source copy.
3. **Handling Macros (if used):**
   * Expands macros when a preprocessor is part of the compiler.

**Two Steps in Lexical Analysis**

Sometimes, lexical analysis is split into two processes for simplicity:

1. **Scanning:**
   * Handles basic operations like:
     + Deleting comments.
     + Compacting multiple whitespace characters into one.
2. **Lexical Analysis Proper:**
   * Identifies tokens and sends them to the parser.

**Lexical Analysis vs Parsing**

In a compiler, the analysis phase is divided into two sub-phases:

1. **Lexical Analysis**: Processes the input into tokens by identifying and grouping lexemes.
2. **Parsing (Syntax Analysis)**: Uses tokens to check for syntax correctness according to the grammar rules of the language.

This separation offers several advantages:

**Advantages of Separating Lexical Analysis and Parsing**

1. **Simplified Design**
   * **Cleaner Code:** By separating concerns, each phase focuses on a specific task, making the compiler design simpler.
   * **Easier Parsing:** Parsers don’t need to handle comments and whitespace, which the lexical analyzer has already removed.
   * **Better Language Design:** When creating a new programming language, separating lexical and syntactic concerns ensures a cleaner overall design.
2. **Improved Compiler Efficiency**
   * Specialized techniques can be applied for lexical analysis, such as efficient input buffering and pattern matching.
   * These optimizations are not burdened by the complexities of parsing, leading to a faster compiler.

**Tokens, Patterns, and Lexemes**

These three terms are fundamental in **lexical analysis**:

1. **Token**
   * A **token** is a pair:
     + **Token Name:** A symbolic representation of a type of lexical unit (e.g., keyword, identifier, operator).
     + **Attribute Value:** Optional additional information, such as the actual value of an identifier or a literal.
   * The parser processes **token names**, which represent the basic building blocks of the source code.
   * Example: The keyword if has a token name **IF**.
2. **Pattern**
   * A **pattern** is a rule or description that defines the possible forms of lexemes for a token.
     + For keywords, the pattern is the exact sequence of characters (e.g., if).
     + For identifiers or numbers, the pattern is more general (e.g., a letter followed by letters or digits for identifiers).
3. **Lexeme**
   * A **lexeme** is the actual sequence of characters in the source code that matches a pattern for a token.
   * The lexical analyzer identifies and groups lexemes as instances of tokens.
   * Example: The lexeme score matches the pattern for the **ID** token.

**Examples of Tokens, Patterns, and Lexemes**

|  |  |  |
| --- | --- | --- |
| **Token Name** | **Informal Description** | **Sample Lexemes** |
| **IF** | Characters i and f | if |
| **ELSE** | Characters e, l, s, e | else |
| **COMPARISON** | Relational operators | <=, !=, >= |
| **ID** | Letter followed by letters/digits | pi, score, D2 |
| **NUMBER** | Any numeric constant | 3.14159, 6.02e23 |
| **LITERAL** | Any string enclosed in quotation marks | "core dumped" |
| **SEMICOLON** | Single semicolon | ; |

**Key Points About Tokens**

1. **Keywords:** Each keyword (e.g., if, else) corresponds to a unique token with its pattern being the exact keyword.
2. **Operators:** Can be grouped into classes, such as **COMPARISON**, or treated individually.
3. **Identifiers (ID):** Represent all user-defined names, matching a general pattern (e.g., a letter followed by letters/digits).
4. **Constants:** Include numbers and string literals, with their own tokens (e.g., **NUMBER**, **LITERAL**).
5. **Punctuation:** Each symbol (e.g., (, ), ;) can have its own token.

**Input Buffering in Lexical Analysis**

Input buffering is a technique used to efficiently read the source program during lexical analysis. Since compilers often need to look ahead by one or more characters to identify lexemes, proper buffering ensures the process is both accurate and time-efficient.

**Challenges in Reading Input**

1. **Lookahead Requirement**:
   * To recognize the end of a lexeme, additional characters must sometimes be inspected.
   * For instance:
     + Identifiers end when a non-alphanumeric character is encountered.
     + Operators like = may belong to multi-character operators (==, <=).
2. **Performance Concerns**:
   * Processing characters individually can be time-consuming, especially for large programs.
   * Efficient buffering reduces the overhead of input handling.

**Buffer Pair Scheme**

A **two-buffer system** is commonly used for input buffering. Here’s how it works:

1. **Structure**:
   * Two buffers of size **N** are used to hold the input characters alternately.
   * The input is divided between these buffers, reducing the need for frequent reloading.
2. **Pointers**:
   * **lexemeBegin**: Points to the start of the current lexeme.
   * **forward**: Scans ahead to identify the end of the lexeme.
3. **Process**:
   * **Forward Movement**:
     + The forward pointer moves to locate the end of a lexeme.
     + Once identified, forward may need to retract slightly (e.g., for multi-character operators).
   * **Buffer Switching**:
     + If forward reaches the end of a buffer, the next buffer is loaded with new input.
     + The system ensures the lexeme is not overwritten during this process.
4. **Efficiency**:
   * As long as the sum of the lexeme's length and lookahead does not exceed **N**, this scheme works seamlessly.

**Sentinels for Optimization**

The **sentinel** technique improves the buffer pair scheme by reducing overhead:

1. **Sentinel Character**:
   * A special character (e.g., **eof**) is added at the end of each buffer.
   * **eof** acts as a marker for the buffer’s end and cannot be part of the source program.
2. **Advantages**:
   * Eliminates the need for explicit checks for buffer boundaries.
   * Combines the buffer-end test with the character-check test.
   * Simplifies the algorithm for advancing the forward pointer.
3. **Process**:
   * If forward encounters **eof**:
     + Check if it is the actual end of input.
     + If not, switch buffers and continue.

**Summary Algorithm**

1. Initialize lexemeBegin and forward.
2. Use the forward pointer to scan the input.
3. On reaching the end of a buffer, check for the sentinel (eof):
   * Reload the alternate buffer if input remains.
4. Record the identified lexeme and return the token.
5. Reset lexemeBegin to the character after the lexeme.

**Benefits of Input Buffering**

1. **Efficient Input Handling**: Reduces the overhead of processing individual characters.
2. **Seamless Lookahead**: Supports large lookaheads without compromising performance.
3. **Error-Free Lexeme Handling**: Ensures lexemes are preserved across buffer boundaries.

**Specification of Tokens**

In lexical analysis, the specification of tokens involves describing the possible patterns that lexemes can take. These patterns are typically specified using strings, languages, and regular expressions. Below is a detailed overview of these specifications and their related concepts.

**1. String Specification**

A **string** is defined as a finite sequence of symbols drawn from a fixed alphabet. The alphabet is a finite set of symbols, often represented by characters or tokens.

**Operations on Strings**

1. **Prefix of String**:  
   A prefix of a string is any initial part of the string, including the string itself and the empty string (ε).
   * Example: For s = abcd, the prefixes are: ∈, a, ab, abc, abcd.
2. **Suffix of String**:  
   A suffix of a string is any ending part of the string, including the string itself and the empty string.
   * Example: For s = abcd, the suffixes are: ∈, d, cd, bcd, abcd.
3. **Proper Prefix of String**:  
   A proper prefix includes all prefixes of a string excluding the empty string (∈) and the string itself.
   * Example: For s = abcd, the proper prefixes are: a, ab, abc.
4. **Proper Suffix of String**:  
   A proper suffix includes all suffixes of a string excluding the empty string (∈) and the string itself.
   * Example: For s = abcd, the proper suffixes are: d, cd, bcd.
5. **Substring of String**:  
   A substring is a part of the string that can be obtained by deleting any prefix or suffix.
   * Example: For s = abcd, substrings include: ∈, abcd, bcd, abc, etc.
6. **Proper Substring of String**:  
   A proper substring excludes the empty string (∈) and the string itself.
   * Example: For s = abcd, the proper substrings are: bcd, abc, cd, ab, etc.
7. **Subsequence of String**:  
   A subsequence is derived by deleting zero or more symbols (not necessarily consecutive) from the string.
   * Example: For s = abcd, subsequences include: abd, bcd, bd, etc.
8. **Concatenation of String**:  
   Concatenation of two strings s and t is the operation of appending t after s.
   * Example: For s = abc and t = def, the concatenation st = abcdef.

**2. Language Specification**

A **language** is a countable set of strings over a fixed alphabet. In the context of lexical analysis, languages define valid sequences of characters or tokens.

**Operations on Languages**

1. **Union (L ∪ M)**:  
   The union of two languages L and M consists of all strings that are in either L or M.
   * Example: If L = {a, b} and M = {c, d}, then L ∪ M = {a, b, c, d}.
2. **Concatenation (L ⋅ M)**:  
   The concatenation of two languages L and M consists of all possible strings formed by appending a string from L with a string from M.
   * Example: If L = {a, b} and M = {c, d}, then L ⋅ M = {ac, ad, bc, bd}.
3. \**Kleene Closure (L*)\*\*:  
   The Kleene closure of a language L consists of all strings formed by concatenating any number of strings from L, including the empty string (∈).
   * Example: If L = {a, b}, then L\* = {∈, a, b, aa, bb, aaa, bbb, ...}.
4. **Positive Closure (L+)**:  
   The positive closure is similar to Kleene closure, but it excludes the empty string (∈). It consists of all strings formed by concatenating one or more strings from L.
   * Example: If L = {a, b}, then L+ = {a, b, aa, bb, aaa, bbb, ...}.

**3. Regular Expressions (Regex) for Token Specification**

A **regular expression** (regex) is a sequence of symbols that defines a pattern for matching lexemes. Regular expressions are used to describe the languages and patterns that tokens can follow.

**Basic Regular Expressions**

1. **Empty String (ε)**:  
   The regular expression ε denotes a language containing only the empty string, i.e., L(ε) = {∈}.
2. **Symbols from the Alphabet (a)**:  
   A symbol a in the alphabet Σ is a regular expression that denotes the language containing just that symbol.
   * Example: If a is a character from the alphabet, then L(a) = {a}.

**Combining Regular Expressions**

1. **Union (r | s)**:  
   The union of two regular expressions r and s denotes the language L(r) ∪ L(s), which contains all strings that are in either L(r) or L(s).
   * Example: r | s means either r or s matches.
2. **Concatenation (rs)**:  
   The concatenation of two regular expressions r and s denotes the language L(r) ⋅ L(s), which consists of all strings formed by concatenating a string from L(r) with a string from L(s).
   * Example: If r = a and s = b, then rs = ab.
3. **Kleene Closure (r\*)**:  
   The Kleene closure of a regular expression r denotes the language L(r)\*, which consists of all strings formed by concatenating zero or more strings from L(r).
   * Example: r\* matches zero or more occurrences of r.
4. **Positive Closure (r+)**:  
   The positive closure of a regular expression r denotes the language L(r)+, which consists of all strings formed by concatenating one or more strings from L(r).
   * Example: r+ matches one or more occurrences of r.

**Regular Definitions**

A **regular definition** provides a way to give names to regular expressions, enabling more complex patterns to be defined in terms of simpler ones. Regular definitions follow a sequence of assignments:

* d1 → r1
* d2 → r2
* ...
* dn → rn

Each di is a distinct name, and each ri is a regular expression over the alphabet Σ and possibly other previously defined names.

**Example Regular Definition:**

For identifiers in programming languages (strings starting with a letter and followed by letters or digits), the regular definition might look like:

* letter → A | B | ... | Z | a | b | ... | z
* digit → 0 | 1 | ... | 9
* id → letter (letter | digit)\*

This defines an identifier as starting with a letter and followed by any combination of letters and digits.

[Recognition of tokens using finite automata](https://www.btechvibes.com/2023/06/recognition-of-tokens-in-compiler-design.html)