

# SOFTWARE RE-ENGINEERING

**SE-409**

# TECHNIQUES USED FOR REVERSE ENGINEERING

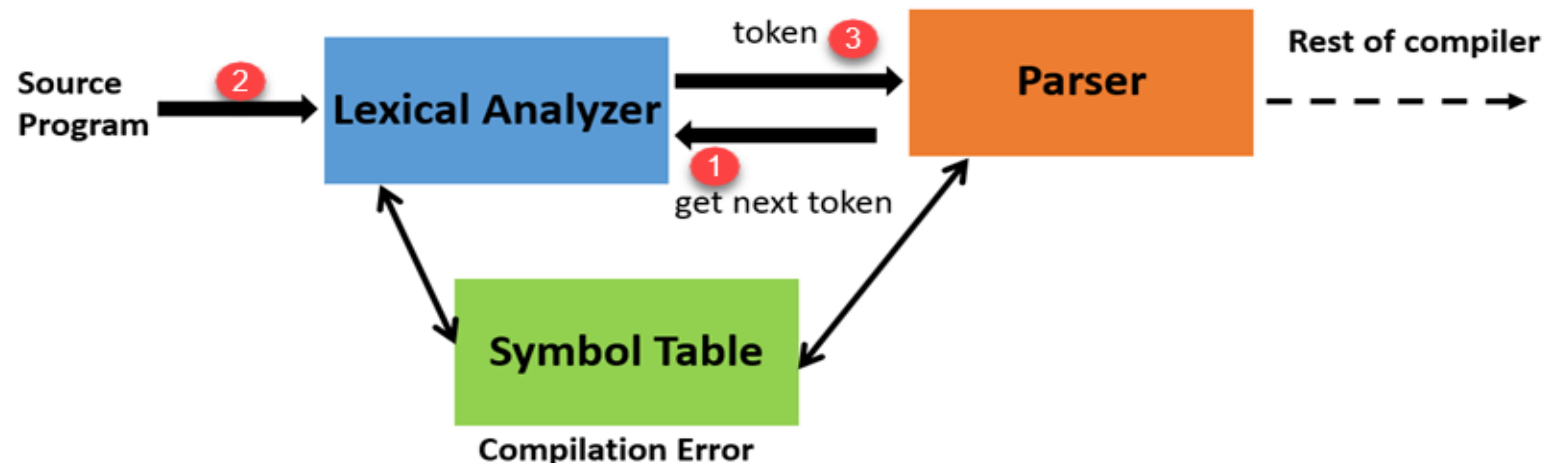
- Fact-finding and information gathering from the source code are the keys to the Goal/Models/Tools paradigm, which partitions a process for reverse engineering into three ordered stages: Goals, Models, and Tools.
- In order to extract information which is not clearly available in source code, automated analysis techniques are used.
- The well-known analysis techniques that facilitate reverse engineering are lexical analysis, syntactic analysis, data flow analysis, program slicing, visualization, etc.

# Lexical Analysis

- Lexical analysis is the process of decomposing the sequence of characters in the source code into its constituent lexical units.
- Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences.
- The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

# Lexical Analysis

- The main task of lexical analysis is to read input characters in the code and produce tokens. Lexical analyzer scans the entire source code of the program. It identifies each token one by one.
- If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer.



# Lexical Analysis

1. 'Get next token' is a command which is sent from the parser to the lexical analyzer.
  2. On receiving this command, the 'lexical analyzer' scans the input until it finds the next token.
  3. It returns the token to 'Parser'.
- Lexical Analyzer skips whitespaces and comments while creating these tokens. If any error is present, then Lexical analyzer will correlate that error with the source file and line number.
  - Lexical analyzer is used by web browsers to format and display a web page with the help of parsed data from JavaScript, HTML, CSS.

# Lexical Analysis

- Fundamental terms: Tokens, Patterns, Lexeme.
- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages, like a word in a sentence.
- **Example of tokens:**
  - Type token like types: int, float, char...
  - Punctuation tokens like symbols: {, }, (, ),....
  - Alphabetic tokens like keywords or identifiers

# Example

In `int count = 10;;`, the tokens are:

- `int` (a type token)
- `count` (an identifier)
- `=` (an assignment operator)
- `10` (a number)
- `;` (a punctuation token)

- A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

- **Example of Lexeme:**

- If we have a token for numbers, a lexeme could be 273 or 3.14.

- For a keyword token, the lexemes could be float, if, or return.

# Example

In the statement `int number = 5;;` the lexemes are:

- `int`
- `number`
- `=`
- `5`
- `;`

- A pattern defines the structure of tokens. It's like a rule that the lexical analyzer uses to recognize lexemes for a particular token type.

### **Examples of Patterns:**

- For a number token, the pattern might be a sequence of digits (e.g., 0-9).
- For a variable token, the pattern could be a sequence of alphabetic characters followed by numbers (e.g., abc123).
- For a keyword token, the pattern might simply match exact words like if, while, or return.

# Lexical Analysis (Example 1)

```
int main()  
{  
    // 2 variables  
    int a, b;  
    a = 10;  
    return 0;  
}
```

Mention All the valid tokens are?

# Lexical Analysis (Example 1)

➤ **Answer** : 'int' 'main' '(' ')' '{' 'int' 'a' ',' 'b' ';' 'a' '=' '10' ';' 'return' '0' ';' '}'

Above are the valid tokens. Observe that it omitted the comments.

➤ As another example & solve the following statement:

**printf (“Lexical Analysis”);**

## Lexical Analysis (Example 2)

*# This is a python code*

x = 10

y = x + 5

print("The result is:", y)

During lexical analysis, the Python interpreter will break down this code into tokens:

❖ **Comments: # This is a python code**

- Ignored: Comments are ignored by the lexical analyzer.

❖ **Tokens in Line 2: x = 10**

- x → Identifier (variable name)
- = → Assignment operator
- 10 → Numeric literal (integer)

❖ **Tokens in Line 3: y = x + 5**

- y → Identifier
- = → Assignment operator
- x → Identifier
- + → Arithmetic operator
- 5 → Numeric literal (integer)

❖ **Tokens in Line 4: print("The result is:", y)**

- print → Keyword (Python built-in function)
- ( → Left parenthesis(symbol)
- "The result is:" → String literal
- , → Comma(symbol)
- y → Identifier
- ) → Right parenthesis(symbol)

# Syntactic Analysis

- Syntactic analysis is defined as analysis that tells us the logical meaning of certainly given sentences or parts of those sentences.
- In simple words, Syntactic analysis is the process of analyzing natural language with the rules of formal grammar.
- It analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.
- Syntax Analysis in Compiler Design process comes after the Lexical analysis phase.

# Syntactic Analysis

- The syntactic analysis basically assigns a semantic structure to text. It is also known as syntax analysis or parsing.
- The word 'parsing' is originated from the Latin word 'pars' which means 'part'. The syntactic analysis deals with the syntax of Natural Language.
- Consider the following sentence:

**Sentence: School go a boy**

- The above sentence does not logically convey its meaning, and its grammatical structure is not correct. So, Syntactic analysis tells us whether a particular sentence conveys its logical meaning or not and whether its grammatical structure is correct or not.

# Difference between Lexical and Syntactic analysis

1. Every gardener likes the sun.
2. Like sun the every gardener?

In both sentences, all the words are the same, but only the first sentence is syntactically correct and easily understandable. But we cannot make these distinctions using Basic lexical processing techniques

- The aim of lexical analysis is in Data Cleaning and Feature Extraction with the help of techniques such as **Stemming, Lemmatization, Correcting misspelled words, etc.**
- But on the contrary, in syntactic analysis, our target is to : **Find the roles played by words in a sentence, Interpret the relationship between words, Interpret the grammatical structure of sentences.**

# Why do you need Syntactic Analyzer?

- Check if the code is valid grammatically
- The syntactical analyzer helps you to apply rules to the code
- Helps you to make sure that each opening brace has a corresponding closing balance
- Each declaration has a type and that the type must be exists

# Program Slicing

- The essential idea of program slicing is to identify only those statements that are of relevance to the slicing criterion – i.e. those that affect or are affected by some statement.
- A program slice is a portion of a program with an execution behavior identical to the initial program with respect to a given criterion, but may have a reduced size.
- Slicing can move in two directions – forwards and backwards.

# Backward slicing

- Starts from a variable or a set of variables and identifies the statements that influence their values. It is useful for understanding the causes of a particular variable's value.

# Example

- Assuming you have a hypothetical code snippet that looks something like this:

Criterion:  $S < [3]: \text{sum} >$

1.  $a = 10$

2.  $b = 20$

3.  $\text{sum} = a + b$

4.  $c = \text{sum} * 2$

5.  $d = \text{sum} - 5$

Based on the example code, the backward slice for sum would be:  
[3]->[2]->[1]

```
1. a = 10  
2. b = 20  
3. sum = a + b
```

# Forward slicing

- Starts from a statement or a point in the program and identifies the statements that are influenced by it. It is useful for understanding the consequences of a particular statement.

# Example

Criteria:  $S \leq [3]: \text{sum}$

- 1.  $x = 5$
- 2.  $y = 10$
- 3.  $\text{sum} = x + y$
- 4.  $\text{result} = \text{sum} * 2$
- 5.  $\text{final\_result} = \text{result} - 3$

The forward slice starting from sum would be:  
[3]->[4]->[5]

- 3. `sum = x + y`
- 4. `result = sum * 2`
- 5. `final_result = result - 3`

# Why Do We Need Slicing?

- Debugging: Focus on parts of program relevant for a bug.
- Program understanding: Which statements influence this statement?
- Change impact analysis: Which parts of a program are affected by a change? What should be retested?

# Reading Assignment

Cover the following **TECHNIQUES USED FOR REVERSE ENGINEERING:**

- **Control Flow Analysis**
- **Data Flow Analysis**
- **Visualization**