LEXICAL ANALYZER

Build Scanner

Prepared By
Student Name Student ID

Under Supervision
Name of Doctor
Name of T. A.

Introduction to Lexical Analysis and Compiler Phases

The process of compiling source code begins with **lexical analysis**. This initial phase involves scanning the input source code, which is a sequence of characters, and grouping these characters into meaningful units called **lexemes**. For each identified lexeme, the lexical analyzer produces a corresponding **token**. A token represents a categorized unit of meaning, such as an identifier, keyword, operator, or literal.

Lexical analysis serves to simplify the subsequent stages of compilation by abstracting away the character-level details of the source code. The output of the lexical analyzer, a stream of tokens, becomes the input for the next phase.

Phases of Compiler

Lexical Analysis : Breaks code into **tokens** and lexeams

- **Syntax Analysis :** Checks grammar and creates a **syntax tree**.

- **Semantic Analysis:** Checks meaning and correctness.
- **Intermediate Code Generation:** Creates a temporary, machine-independent code.
- **Code Optimization:** Improves the intermediate code for efficiency.
- **Code Generataon:** Creates the final machin code.

What is a **Lexical Analyzer**
is responsible for scanning the source code and converting it into tokens.
It identifies keywords, operators, identifiers, and other elements.
3.Software Tools
3.1 Programed used Visual studio
3.2 Computer program Git GitHub
3.2 Computer program python

4.Implentation of Lexical analysir

```
LETTER = 0
DIGIT = 1
UNKNOWN = 99
EOF = -1

INT_LIT = 10
IDENT = 11
ASSIGN_OP = 20
ADD_OP = 21
SUB_OP = 22
MULT_OP = 23
DIV_OP = 24
```

```python
LEFT_PAREN = 25
RIGHT_PAREN = 26
SEMICOLON = 27
KEYWORD_INT = 28
KEYWORD_DOUBLE = 29
KEYWORD_WHILE = 30

charClass = None
lexeme = ""
nextChar = ""
lexLen = 0
nextToken = None
input_data = "int youssef = 61; double int ahmed = 21; while(1);"
index = 0

def addChar():
    global lexeme, lexLen
    if lexLen <= 98:
        lexeme += nextChar
    else:
        print("Error - lexeme is too long")

def getChar():
    global nextChar, charClass, index
    if index < len(input_data):
        nextChar = input_data[index]
        index += 1
        if nextChar.isalpha():
            charClass = LETTER
        elif nextChar.isdigit():
            charClass = DIGIT
        else:
            charClass = UNKNOWN
    else:
        charClass = EOF

def getNonBlank():
    global nextChar
    while nextChar.isspace():
        getChar()

def lookup(ch):
    global nextToken
    operators = {'+': ADD_OP, '-': SUB_OP, '*':
MULT_OP, '/': DIV_OP,
                 '(': LEFT_PAREN, ')': RIGHT_PAREN,
';': SEMICOLON, '=': ASSIGN_OP}
    addChar()
    nextToken = operators.get(ch, EOF)
    return nextToken

def checkKeyword():
    global nextToken
    keywords = {'int': KEYWORD_INT, 'double': KEYWORD_DOUBLE, 'while': KEYWORD_WHILE}
    nextToken = keywords.get(lexeme, IDENT)

def lex():
    global lexeme, nextToken, lexLen
    lexeme = ""
    lexLen = 0
    getNonBlank()

    if charClass == LETTER:
        addChar()
        getChar()
        while charClass in [LETTER, DIGIT]:
            addChar()
            getChar()
        checkKeyword()

    elif charClass == DIGIT:
        addChar()
        getChar()
        while charClass == DIGIT:
            addChar()
            getChar()
        nextToken = INT_LIT

    elif charClass == UNKNOWN:
```

```python
            lookup(nextChar)
            getChar()

    elif charClass == EOF:
            nextToken = EOF
            lexeme = "EOF"

    print(f"Next token is: {nextToken}, Next lexeme is '{lexeme}'")
    return nextToken

def main():
    global input_data, index
    index = 0
    getChar()

    while nextToken != EOF:
            lex()

if __name__ == "__main__":
    main()
```

Character Class Constants

The constants `LETTER = 0`, `DIGIT = 1`, `UNKNOWN = 99`, and `EOF = -1` are used to represent different categories of characters or states encountered during input processing.

- `LETTER` is assigned to alphabetic characters.
- `DIGIT` is used for numeric digits.
- `UNKNOWN` represents any unrecognized or special characters.
- `EOF` signifies the end of the input.

Token Type Constants

Constants such as `INT_LIT = 10`, `IDENT = 11`, `ASSIGN_OP = 20`, `ADD_OP = 21`, and others are used to represent different token types found in the input.

- `INT_LIT` corresponds to integer literals.
- `IDENT` is used for identifiers, like variable names.
- Operators such as `+` are represented by tokens like `ADD_OP`.
  These constants help the analyzer categorize and handle various components of the source code effectively.

Global Variables

- `charClass` tracks the type of the current character being processed (letter, digit, unknown, or EOF).
- `lexeme` stores the current string of characters forming a token.
- `nextChar` holds the next character to be processed.
- `lexLen` tracks the length of the current lexeme.
- `nextToken` holds the type of the current token being processed.

- `input_data` is the full string of source code being analyzed.
- `index` keeps track of the current position within the input string.

Function Descriptions

addChar()
Adds the current character to the `lexeme`, but only if the lexeme's length does not exceed a predefined limit (typically 98 characters). If the limit is exceeded, an error message is displayed.

getChar()
Reads the next character from `input_data` and updates `charClass` depending on whether the character is a letter, digit, or something else. If the end of the input is reached, `charClass` is set to `EOF`.

getNonBlank()
Skips over spaces and tab characters in the input to ignore irrelevant whitespace. This ensures that only meaningful characters are processed.

lookup(ch)
Handles symbols and operators by mapping them to their corresponding token types. For example, the character `+` is mapped to `ADD_OP`. This function also adds the character to the current lexeme.

checkKeyword()
Checks if the current lexeme matches a reserved keyword such as `int`, `double`, or `while`. If so, it assigns the appropriate keyword token. If not, the lexeme is treated as a general identifier.

lex()
This is the main function that handles the tokenization process. It

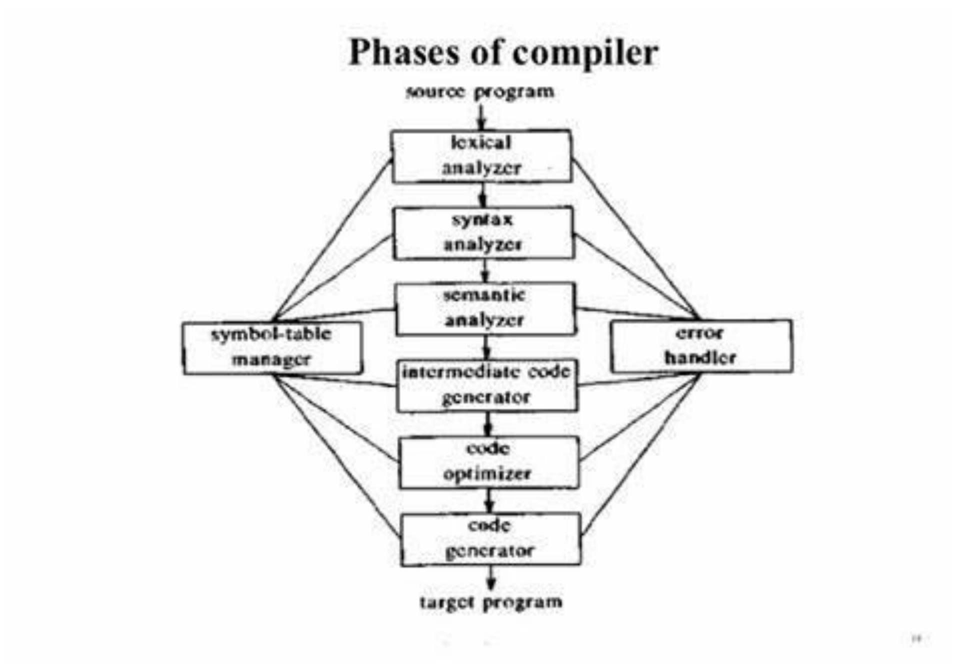
resets the lexeme, skips whitespace, and then processes characters based on their type:

- If the character is a letter, it builds an identifier or keyword.
- If it is a digit, it builds an integer literal.
- If it is a symbol, it uses `lookup()` to classify it.
- If the input ends, it assigns the EOF token.

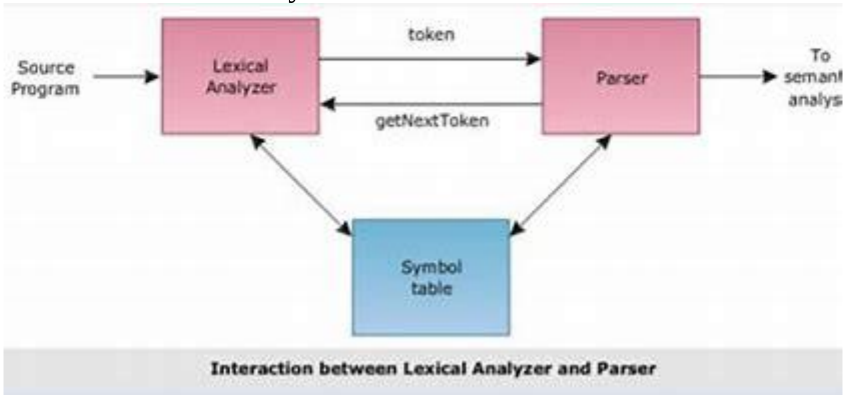
  This function prints and returns the current token and lexeme.

main()
The central controller of the program. It begins by reading the first character, and then continuously calls `lex()` until the end of the input is reached.

if name == "main":
A standard Python check to ensure that the `main()` function runs only when the script is executed directly, not when it is imported as a module.



Phases of lexica lAnalyzer



5. References:

1-Text book concepts of programming language 12/E

2-W3school