Ministerul Educaţiei, Culturii şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Departamentul Ingineria Software şi Automatică

REPORT

Laboratory work 3
Course: Formal Languages & Finite Automata
Theme: Implementing a simple lexer

Elaborated: Latcovschi Cătălin,
FAF-221

Verified: asist.univ.
Dumitru Crețu

Chișinău 2024

Overview

The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages.

The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

Objectives

1. Understand what lexical analysis is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Implementation

1. Initialization (__init__ method):

The lexer defines a set of rules for recognizing different types of tokens in arithmetic expressions: whitespace, floating-point numbers, integers, arithmetic operators (plus, minus, multiply, divide), and parentheses.

```
class ArithmeticLexer:
    def __init__(self):
        # Rules for the arithmetic expressions
        self.rules = [
            (r'[ \t]+', None),  # Ignore whitespace
            (r'\d+\.\d+', 'FLOAT'),
            (r'\d+', 'INTEGER'),
            (r'\+', 'PLUS'),
            (r'-', 'MINUS'),
            (r'\*', 'MUL'),
            (r'/', 'DIV'),
            (r'\(', 'LPAREN'),
            (r'\)', 'RPAREN'),
        ]
        self.rules = [(re.compile(pattern), token_type) for (pattern, token_type) in self.rules]
```

Each rule consists of a regular expression pattern and an associated token type. Whitespace is ignored (assigned None), while other patterns are associated with their respective token types like 'FLOAT', 'INTEGER', etc.

The regular expressions are precompiled for efficiency.

2. **Tokenization (tokenize method):**

The lexer analyzes a given text (an arithmetic expression) character by character to match it against the defined regular expression rules.

```
def tokenize(self, text):
    pos = 0
    while pos < len(text):
        match = None
        for pattern, token_type in self.rules:
            regex_match = pattern.match(text, pos)
            if regex_match:
                match = regex_match
                if token_type:  # Ignore tokens like whitespace
                    yield (token_type, regex_match.group(0))
                break
        if not match:
            # Yield an 'ILLEGAL' token type for further processing instead of raising an error
            yield ('ILLEGAL', text[pos])
            pos += 1
        else:
            pos = match.end(0)
```

If a match is found, the lexer yields a token, which is a tuple consisting of the token type and the matched string (lexeme).

If no match is found for a character, the lexer yields an 'ILLEGAL' token, indicating the presence of an unexpected or forbidden character in the expression.

The process continues until the entire text has been tokenized.

3. **validate_expression Function**

The function takes a sequence of tokens produced by the lexer and validates the structure of the arithmetic expression.

It checks for illegal tokens, ensuring there are no forbidden symbols in the expression.

```
def validate_expression(tokens):
    paren_stack = []
    last_token_type = None

    for token in tokens:
        token_type, token_value = token

        # Check for illegal tokens
        if token_type == 'ILLEGAL':
            return False, f"Illegal character found: {token_value}"

        # Check for balanced parentheses
        if token_type == 'LPAREN':
            paren_stack.append(token_value)
        elif token_type == 'RPAREN':
            if not paren_stack:
                return False, "Unbalanced parentheses"
            paren_stack.pop()
```

It uses a stack (implemented as a list paren_stack) to ensure that parentheses are properly balanced. Open parentheses are pushed onto the stack, and close parentheses cause the stack to be popped.

If a close parenthesis is encountered when the stack is empty, or if the stack is not empty at the end of validation, the expression is deemed invalid due to unbalanced parentheses.

```
        # Check for valid sequences
        if last_token_type in ['PLUS', 'MINUS', 'MUL', 'DIV']:
            if token_type in ['PLUS', 'MINUS', 'MUL', 'DIV', 'RPAREN']:
                return False, "Invalid operator usage"
        if last_token_type == 'LPAREN' and token_type in ['PLUS', 'MINUS', 'MUL', 'DIV', 'RPAREN']:
            return False, "Invalid expression after '('"
        if last_token_type in ['INTEGER', 'FLOAT', 'RPAREN'] and token_type == 'LPAREN':
            return False, "Invalid expression before '('"

        last_token_type = token_type

    if paren_stack:
        return False, "Unbalanced parentheses"

    return True, "Valid expression"
```

The function checks the sequence of tokens to ensure they form a valid arithmetic expression. For example, an operator should not follow another operator directly, and an expression should not start with a closing parenthesis or an operator.

It tracks the type of the last seen token to determine whether the current token is allowed to follow it, based on standard rules of arithmetic expressions.

Results

```
# Example usage
lexer = ArithmeticLexer()
expression = '(3.14 + 2 * (1 - 5))'
tokens = list(lexer.tokenize(expression))
is_valid, message = validate_expression(tokens)
print(f"Expression: {expression}")
print(f"Tokens: {tokens}")
print(f"Validation: {message}")
```

```
C:\Users\Catlin\AppData\Local\Programs\Python\Python39\python.exe C:\Users\Catlin\Desktop\FLFA\lab3\lexer_scanner.py
Expression: (3.14 + 2 * (1 - 5))
Tokens: [('LPAREN', '('), ('FLOAT', '3.14'), ('PLUS', '+'), ('INTEGER', '2'), ('MUL', '*'), ('LPAREN', '('), ('INTEGER', '1'), ('MINUS', '-'), ('INTEGER', '5'), ('RPAREN
Validation: Valid expression
```

Conclusions

In this laboratory work, I successfully implemented a simple lexer to perform lexical analysis on arithmetic expressions. The process involved developing an ArithmeticLexer class, which tokenizes input strings based on predefined patterns representing different elements of arithmetic expressions such as integers, floating-point numbers, and operators. The lexer also identifies illegal characters by yielding 'ILLEGAL' tokens, ensuring robust error handling.

Moreover, I implemented the validate_expression function, which uses the tokens generated by the lexer to validate the structural integrity of arithmetic expressions. This function ensures that all tokens conform to the grammar rules of arithmetic expressions, checks for balanced parentheses, and prohibits invalid sequences of tokens, thereby ensuring the syntactical correctness of the input.

Through this project, I deepened my understanding of lexical analysis, familiarized myself with the operational mechanics of lexers, and applied these concepts to validate arithmetic expressions effectively. The completion of this work has enhanced my skills in designing and implementing components essential for the parsing and interpretation phases in compilers and interpreters.