MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Mihalachi Mihai FAF-221

Report

Laboratory work no.3

of Formal Languages & Finite Automata

Chișinău – 2024

Theory

Lexical analysis, often referred to as lexing or tokenization, is the first phase of the compiler or interpreter process in programming language processing. It involves breaking down the input text (source code) into smaller units called tokens. Each token represents a meaningful piece of the source code, such as identifiers (variable names), keywords, literals (constants), operators, and punctuation symbols.

The main goal of lexical analysis is to simplify the source code into a format that is easier to work with for subsequent phases of the compiler or interpreter. By identifying and categorizing tokens, lexical analysis provides a structured representation of the source code, which is then used by the parser to create a syntax tree or by the interpreter to execute the program.

Lexical analysis typically involves scanning the input text character by character, recognizing patterns based on predefined rules (specified by regular expressions or finite automata), and emitting tokens for each recognized pattern. This process may also involve discarding whitespace characters and comments, as they usually do not contribute to the semantics of the program. Overall, lexical analysis plays a crucial role in the translation of human-readable source code into a form that can be understood and processed by machines.

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 description:

1. The method used to analyse the lexer that I input myself is separated into code snippets for different kinds of tokens, so I'll take and explain them separately(Note that the inputted code is a string):

    a. For keyword and identifier I use a single code snippet. For these token types I check if the character is a letter (or _ or $), then I check the next characters to see the whole word(or variable name that contains other characters). If the word is a keyword (it belongs to an array that I made with some keywords), then the token is automatically a keyword, if not, then it's classified as an identifier. Here is the code snippet for this:

```
if self.current_char.isalpha() or self.current_char == '_' or
self.current_char == '$':

      identifier = self.current_char

                 self.advance()

      while self.current_char is not None and (self.current_char.isalnum()
or self.current_char == '_'):

                identifier += self.current_char

                self.advance()

      if identifier in self.keywords:

                return Token('KEYWORD', identifier)

      return Token('IDENTIFIER', identifier)
```

    b. For literals I do the same as for the keywords, but instead of looking for letters or specific characters, I just look for numbers. There is also a separate thing for strings, where I check if there is one of the quotes, then I identify the whole string by checking for the closing quote. If there is no closing quote, then i raise an error that the token is not valid because of missing end quote. Here is the code snippet:

```
  if self.current_char.isdigit():

    literal = ''

    while self.current_char is not None and (self.current_char.isdigit()
         or self.current_char == '.'):

        literal += self.current_char

        self.advance()

    return Token('LITERAL', literal)


if self.current_char == '"' or self.current_char == "'":

    return self.tokenize_string()
```

There is a separate function for checking strings:

```
def tokenize_string(self):
        quote_char = self.current_char
        self.advance()
        string_literal = ''
        while self.current_char is not None and self.current_char != quote_char:
                string_literal += self.current_char
                self.advance()
        if self.current_char != quote_char:
                raise ValueError("Unclosed string literal")
        self.advance()
        return Token('LITERAL', string_literal)
```

c. For separators I have a set with the selected sort of separators and if the characters belongs to the set it sets it as a separator:

```
if self.current_char in self.separators:
   separator = self.current_char
   self.advance()
   return Token('SEPARATOR', separator)
```

d. For separators I have the same thing as separators with some added functionality which checks for different kinds of ,=', such as ,==' and ,!='. I check if the character is either a ,!' or ,=', then i check if the next one is a ,='. Here is the code snippet:

```
if self.current_char == '!':
   if self.text[self.pos:self.pos+2] == '!=':
       self.advance()
       self.advance()
       return Token('OPERATOR', '!=')


if self.current_char == '=':
    if self.text[self.pos:self.pos+2] == '==':
        self.advance()
        self.advance()
        return Token('OPERATOR', '==')
    self.advance()
    return Token('OPERATOR', '=')


if self.current_char in self.operators:
    operator = self.current_char
    self.advance()
    return Token('OPERATOR', operator)
```

e. For the final part of the code i check for python comments and have a return „UNKNOWN” if it’s not suitable in any of the categories. For comments I use regex to check if the character has „#”, then nullify the rest of the line. Here is the final code snippet for this method:

```
match = self.comments_pattern.match(self.text, self.pos)
      if match:
              comment = match.group(0)
              self.pos += len(comment)
              break

self.advance()
return Token('UNKNOWN')

return Token('EOF')
```

Conclusion

In conclusion, the laboratory work focused on constructing a sample lexer has provided a comprehensive introduction to the fundamental concepts of lexical analysis and tokenization. Through this endeavor, we have gained valuable insights into the intricate process of interpreting and categorizing the structural components of programming languages.

Throughout the laboratory work, we explored the essential components of a lexer, including the identification of keywords, identifiers, literals, operators, and separators. By implementing these components and defining rules using regular expressions, we developed a robust lexer capable of accurately tokenizing input text.

Looking ahead, the knowledge and skills acquired from this laboratory work lay a solid foundation for further exploration into compiler construction, language design, and related fields. It serves as a springboard for future research and experimentation in the domain analysis field.