Indian Institute of Technology Gandhinagar

Lexer for *Kulant*

CS 327 (Compilers) - Assignment 2

La Retro's Members

Aaryan Darad (21110001)
Abhay Upparwal (21110004)
Harshita Ramchandani (20110074)
Somesh Pratap Singh (19110206)

Under the guidance of

Prof. Abhishek Bichhawat

CONTENTS

Regular Expressions

1. `('Keyword', r'\b(int|word|bigint|char|dotie|bool|list|constant|tuple|if|otif|otw|for|while|get_out|go_on|return|void|try|catch|finally|display|input|start)\b')`:

This regex is for identifying keywords in the provided list, such as int, word, if, for, etc.

\b asserts a word boundary, ensuring that the matched keyword is a whole word and not part of another word.

2. `('Identifier', r'\b[a-zA-Z_][a-zA-Z0-9_]*\b')`:

This regex is for identifying identifiers (variable names).

It starts with a letter or underscore [a-zA-Z_].

Followed by zero or more letters, digits, or underscores [a-zA-Z0-9_]*.

\b asserts a word boundary.

3. `('UnaryOperator', r'-|!|&|sizeof\(\)')`:

This regex matches unary operators like -, !, &, and the function sizeof().

('BinaryOperator', r'==|!=|<=|>=|\|\||&&|[+\-*/=](?!=)'):

This regex matches various binary operators like ==, !=, <=, >=, ||, &&, and common arithmetic operators +, -, *, /, = (excluding cases like ==, !=).

4. `('LogicalLiteral', r'True|False')`:

This regex matches logical literals like True or False.

5. `('StringLiteral', r'"([^"]*)"')`:

This regex matches string literals enclosed in double quotes. The content inside the quotes is captured using ([^"]*).

6. `('Whitespace', r'\s+')`:

This regex matches one or more whitespace characters.

7. `('Parenthesis', r'[()\{\}]')`:

This regex matches parentheses and curly braces.

8. `('Quotation', r'"')`:

This regex matches a single double quote.

9. `('Constant', r'[^"{}\s();]+')`:

This regex matches constants that do not include double quotes, braces, spaces, or semicolons. It recognizes values for the literals and variables assigned.

10. `('EndOfStatement', r';')`:

This regex matches a semicolon, indicating the end of a statement.

Lexer Code Explanation

Code:

```
import re
class kulantLexer:
    def __init__(self):
        self.token_patterns = [
            ('Keyword', r'\b(int|word|bigint|char|dotie|bool|list|constant|tuple|if|otif|otw|for|while|get_out|go_on|return|void|try|catch|finally|display|input|start)\b'),
            ('Identifier', r'\b[a-zA-Z_][a-zA-Z0-9_]*\b'),
            ('UnaryOperator', r'-|!|&|sizeof\(\)'),
            ('BinaryOperator', r'==|!=|<=|>=|\|\||&&|[+\-*/=](?!=)'),
            ('LogicalLiteral', r'True|False'),
            ('StringLiteral', r'"([^"]*)"'),
            ('Whitespace', r'\s+'),
            ('Parenthesis', r'[()\{\}]'),
            ('Quotation', r'"'),
            ('Constant', r'[^"{}\s();]+'),
            ('EndOfStatement', r';')
        ]

    def tokenize(self, source_code):
        tokens = []
        position = 0

        while position < len(source_code):
            match = None

            for token_type, pattern in self.token_patterns:
                regex = re.compile(pattern)
                match = regex.match(source_code, position)

                if match:
                    if token_type == 'StringLiteral':
                        tokens.append(('Quotation', '"'))
                        tokens.append(('StringLiteral', match.group(1)))
                        tokens.append(('Quotation', '"'))
                    else:
                        tokens.append((token_type, match.group(0)))
                    position = match.end()
                    break

            if not match:
                raise Exception(f"Unexpected character at position {position}: {source_code[position]}")

        return tokens



def main():
    filename = "hello_world.lx"
    with open(filename, 'r') as file:
        source_code = file.read()

    lexer = kulantLexer()
    tokens = lexer.tokenize(source_code)

    print(tokens)



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

Our code defines a lexer that breaks down a source code into tokens based on specified patterns specific to Kulant, and it demonstrates the tokenization process for a sample source code file

Token Patterns:

- Token patterns are defined for keywords, identifiers, unary operators, binary operators, logical literals, string literals, whitespace, parentheses, quotations, constants, end-of-statement, printf function, and open/close parentheses.

Tokenization Method:

- The tokenize method processes the source code, iterates through token patterns, and matches them against the source code using regular expressions.
- When a match is found, it categorizes the match into the appropriate token type and appends it to a list of tokens.
- Special handling is provided for string literals to include quotation marks.

Main Function:

- The main function reads the source code from a file ("hello_world.lx").
- It initializes a lexer and tokenizes the source code using the defined token patterns.
- The identified tokens are printed.

Token Types:

- The tokens include keywords, identifiers, string literals, operators, literals,unary operators,binary operators, logical literals, whitespace, parenthesis, quotations, constants and end of statement.

Exception Handling:

- If an unexpected character is encountered during tokenization, an exception is raised, providing information about the position and the unexpected character.

Testcases

```
$ 1

for ( i = 1; i = i + 1; i <= 10){

 display(i)

}
```

```
output:

[('Keyword', 'for'), ('Whitespace', ' '), ('Parenthesis', '('),
('Whitespace', ' '), ('Identifier', 'i'), ('Whitespace', ' '),
('BinaryOperator', '='), ('Whitespace', ' '), ('Constant', '1'),
('EndOfStatement', ';'), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Identifier', 'i'), ('Whitespace', ' '), ('BinaryOperator', '+'),
('Whitespace', ' '), ('Constant', '1'), ('EndOfStatement', ';'),
('Whitespace', ' '), ('Identifier', 'i'), ('Whitespace', ' '),
('BinaryOperator', '<='), ('Whitespace', ' '), ('Constant', '10'),
('Parenthesis', ')'), ('Parenthesis', '{'), ('Whitespace', '\n\t'),
('Identifier', 'display'), ('Parenthesis', '('), ('Identifier', 'i'),
('Parenthesis', ')'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Parenthesis', '}')]
```

```
$2

int start(){
for ( i = 1; i = i + 1; i <= 10){
 display(i)
}

return 0;

}
```

Output:

```
[('Keyword', 'int'), ('Whitespace', ' '), ('Keyword', 'start'),
('Parenthesis', '('), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n'), ('Keyword', 'for'), ('Whitespace', ' '),
('Parenthesis', '('), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Constant', '1'), ('EndOfStatement', ';'), ('Whitespace', ' '),
('Identifier', 'i'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Identifier', 'i'), ('Whitespace', ' '),
('BinaryOperator', '+'), ('Whitespace', ' '), ('Constant', '1'),
('EndOfStatement', ';'), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '<='), ('Whitespace', ' '),
('Constant', '10'), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n\xa0'), ('Identifier', 'display'), ('Parenthesis',
'('), ('Identifier', 'i'), ('Parenthesis', ')'), ('Whitespace', '\n'),
('Parenthesis', '}'), ('Whitespace', '\n'), ('Keyword', 'return'),
('Whitespace', ' '), ('Constant', '0'), ('EndOfStatement', ';'),
('Whitespace', '\n'), ('Parenthesis', '}'), ('Whitespace', '\n')]
```

```
$3
int start(){
int a = 4;
int x = 5;
if( a < 10){
      x = x + 1;
}
otif ( 10 < a < 15){
      x = x + 2;
}
otw {
      x = x + 3;
}
display(x)
display(a)
return 0;
}
```

```
Output:
[('Keyword', 'int'), ('Whitespace', ' '), ('Keyword', 'start'),
('Parenthesis', '('), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n'), ('Keyword', 'int'), ('Whitespace', ' '),
('Identifier', 'a'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Constant', '4'), ('EndOfStatement', ';'),
('Whitespace', '\xa0\n'), ('Keyword', 'int'), ('Whitespace', ' '),
('Identifier', 'x'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Constant', '5'), ('EndOfStatement', ';'),
('Whitespace', '\n'), ('Keyword', 'if'), ('Parenthesis', '('),
('Whitespace', ' '), ('Identifier', 'a'), ('Whitespace', ' '),
('Constant', '<'), ('Whitespace', ' '), ('Constant', '10'),
('Parenthesis', ')'), ('Parenthesis', '{'), ('Whitespace', '\n\t'),
('Identifier', 'x'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Identifier', 'x'), ('Whitespace', ' '),
('BinaryOperator', '+'), ('Whitespace', ' '), ('Constant', '1'),
('EndOfStatement', ';'), ('Whitespace', '\n'), ('Parenthesis', '}'),
('Whitespace', '\n'), ('Keyword', 'otif'), ('Whitespace', ' '),
('Parenthesis', '('), ('Whitespace', ' '), ('Constant', '10'),
('Whitespace', ' '), ('Constant', '<'), ('Whitespace', ' '),
('Identifier', 'a'), ('Whitespace', ' '), ('Constant', '<'),
('Whitespace', ' '), ('Constant', '15'), ('Parenthesis', ')'),
('Parenthesis', '{'), ('Whitespace', '\n\t'), ('Identifier', 'x'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Identifier', 'x'), ('Whitespace', ' '), ('BinaryOperator', '+'),
('Whitespace', ' '), ('Constant', '2'), ('EndOfStatement', ';'),
('Whitespace', '\n'), ('Parenthesis', '}'), ('Whitespace', '\n'),
('Keyword', 'otw'), ('Whitespace', ' '), ('Parenthesis', '{'),
('Whitespace', '\n\t'), ('Identifier', 'x'), ('Whitespace', ' '),
('BinaryOperator', '='), ('Whitespace', ' '), ('Identifier', 'x'),
('Whitespace', ' '), ('BinaryOperator', '+'), ('Whitespace', ' '),
('Constant', '3'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Parenthesis', '}'), ('Whitespace', '\n'), ('Identifier', 'display'),
('Parenthesis', '('), ('Identifier', 'x'), ('Parenthesis', ')'),
('Whitespace', '\n'), ('Identifier', 'display'), ('Parenthesis', '('),
('Identifier', 'a'), ('Parenthesis', ')'), ('Whitespace', '\n'),
('Keyword', 'return'), ('Whitespace', ' '), ('Constant', '0'),
('EndOfStatement', ';'), ('Whitespace', '\n'), ('Parenthesis', '}')]
```

```
$4
int start(){
int i;
for ( i = 1; i = i+1; i <= 10){
      display(“inside for”);
}
int i = 3
while (i != 10){
      print(“inside while”);
      i = i+1
}
return 0;
}
```

```
Output:
[('Keyword', 'int'), ('Whitespace', ' '), ('Keyword', 'start'),
('Parenthesis', '('), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n'), ('Keyword', 'int'), ('Whitespace', ' '),
('Identifier', 'i'), ('EndOfStatement', ';'), ('Whitespace', '\xa0\n'),
('Keyword', 'for'), ('Whitespace', ' '), ('Parenthesis', '('),
('Whitespace', ' '), ('Identifier', 'i'), ('Whitespace', ' '),
('BinaryOperator', '='), ('Whitespace', ' '), ('Constant', '1'),
('EndOfStatement', ';'), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Identifier', 'i'), ('BinaryOperator', '+'), ('Constant', '1'),
('EndOfStatement', ';'), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '<='), ('Whitespace', ' '),
('Constant', '10'), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n\t'), ('Identifier', 'display'), ('Parenthesis',
'('), ('Constant', '“inside'), ('Whitespace', ' '), ('Keyword', 'for'),
('Constant', '”'), ('Parenthesis', ')'), ('EndOfStatement', ';'),
('Whitespace', '\n'), ('Parenthesis', '}'), ('Whitespace', '\n\n'),
('Keyword', 'int'), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Constant', '3'), ('Whitespace', '\n'), ('Keyword', 'while'),
('Whitespace', ' '), ('Parenthesis', '('), ('Identifier', 'i'),
('Whitespace', ' '), ('UnaryOperator', '!'), ('BinaryOperator', '='),
('Whitespace', ' '), ('Constant', '10'), ('Parenthesis', ')'),
('Parenthesis', '{'), ('Whitespace', '\n\t'), ('Identifier', 'print'),
('Parenthesis', '('), ('Constant', '“inside'), ('Whitespace', ' '),
('Keyword', 'while'), ('Constant', '”'), ('Parenthesis', ')'),
('EndOfStatement', ';'), ('Whitespace', '\n\t'), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Identifier', 'i'), ('BinaryOperator', '+'), ('Constant', '1'),
('Whitespace', '\n'), ('Parenthesis', '}'), ('Whitespace', '\n'),
('Keyword', 'return'), ('Whitespace', ' '), ('Constant', '0'),
('EndOfStatement', ';'), ('Whitespace', '\n'), ('Parenthesis', '}')]
```

```
$ 5
int start(){
bool a = True && False;
display(a);
word str = “Kulant”;
Word str1 = “Pro”;
str = str+str1
display(str)
word x = “abhaykumar”;
display(x[6]);
return 0;
}
```

```
Output:
[('Keyword', 'int'), ('Whitespace', ' '), ('Keyword', 'start'),
('Parenthesis', '('), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n'), ('Keyword', 'bool'), ('Whitespace', ' '),
('Identifier', 'a'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Identifier', 'True'), ('Whitespace', ' '),
('UnaryOperator', '&'), ('UnaryOperator', '&'), ('Whitespace', ' '),
('Identifier', 'False'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Identifier', 'display'), ('Parenthesis', '('), ('Identifier', 'a'),
('Parenthesis', ')'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Keyword', 'word'), ('Whitespace', ' '), ('Identifier', 'str'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Constant', '“Kulant”'), ('EndOfStatement', ';'), ('Whitespace',
'\n'), ('Identifier', 'Word'), ('Whitespace', ' '), ('Identifier',
'str1'), ('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', '
'), ('Constant', '“Pro”'), ('EndOfStatement', ';'), ('Whitespace',
'\n'), ('Identifier', 'str'), ('Whitespace', ' '), ('BinaryOperator',
'='), ('Whitespace', ' '), ('Identifier', 'str'), ('BinaryOperator',
'+'), ('Identifier', 'str1'), ('Whitespace', '\xa0\n'), ('Identifier',
'display'), ('Parenthesis', '('), ('Identifier', 'str'),
('Parenthesis', ')'), ('Whitespace', '\n'), ('Keyword', 'word'),
('Whitespace', ' '), ('Identifier', 'x'), ('Whitespace', ' '),
('BinaryOperator', '='), ('Whitespace', ' '), ('Constant',
'“abhaykumar”'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Identifier', 'display'), ('Parenthesis', '('), ('Identifier', 'x'),
('Constant', '[6]'), ('Parenthesis', ')'), ('EndOfStatement', ';'),
('Whitespace', '\n'), ('Keyword', 'return'), ('Whitespace', ' '),
('Constant', '0'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Parenthesis', '}')]
```

```
$6
int start(){
for ( i = 1; i = i+1; i <= 10){
      display(“inside for”);
      if (i == 7){
            get_out;
      }
}
return 0;
}
```

```
output:

[('Keyword', 'int'), ('Whitespace', ' '), ('Keyword', 'start'),
('Parenthesis', '('), ('Parenthesis', ')'), ('Parenthesis', '{'),
('Whitespace', '\n'), ('Keyword', 'for'), ('Whitespace', ' '),
('Parenthesis', '('), ('Whitespace', ' '), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '='), ('Whitespace', ' '),
('Constant', '1'), ('EndOfStatement', ';'), ('Whitespace', ' '),
('Identifier', 'i'), ('Whitespace', ' '), ('BinaryOperator', '='),
('Whitespace', ' '), ('Identifier', 'i'), ('BinaryOperator', '+'),
('Constant', '1'), ('EndOfStatement', ';'), ('Whitespace', ' '),
('Identifier', 'i'), ('Whitespace', ' '), ('BinaryOperator', '<='),
('Whitespace', ' '), ('Constant', '10'), ('Parenthesis', ')'),
('Parenthesis', '{'), ('Whitespace', '\n\t'), ('Identifier', 'display'),
('Parenthesis', '('), ('Constant', '“inside'), ('Whitespace', ' '),
('Keyword', 'for'), ('Constant', '”'), ('Parenthesis', ')'),
('EndOfStatement', ';'), ('Whitespace', '\n\t'), ('Keyword', 'if'),
('Whitespace', ' '), ('Parenthesis', '('), ('Identifier', 'i'),
('Whitespace', ' '), ('BinaryOperator', '=='), ('Whitespace', ' '),
('Constant', '7'), ('Parenthesis', ')'), ('Parenthesis', '{'), ('Whitespace',
'\n\t\t'), ('Keyword', 'get_out'), ('EndOfStatement', ';'), ('Whitespace',
'\n\t'), ('Parenthesis', '}'), ('Whitespace', '\n'), ('Parenthesis', '}'),
('Whitespace', '\n'), ('Keyword', 'return'), ('Whitespace', ' '),
('Constant', '0'), ('EndOfStatement', ';'), ('Whitespace', '\n'),
('Parenthesis', '}')]
```