

Lexer and Scanner

Parfene Daniel, FAF-222

March 17, 2024

1 Introduction

In this report, we will discuss the implementation and functionality of a lexer and scanner for a simple programming language. The lexer and scanner are fundamental components of a compiler or interpreter, responsible for breaking down the source code into tokens for further processing.

2 Lexer Implementation

The lexer is implemented using Python and consists of two main classes: `Token` and `Lexer`.

2.1 Token Class

The `Token` class represents a token in the source code. It has attributes for the token type and value, if applicable.

```
1 class Token:
2     def __init__(self, type, value=None):
3         self.type = type
4         self.value = value
5
6     def __str__(self):
7         return f'Token({self.type}, {self.value})'
8
9     def __repr__(self):
10        return self.__str__()
```

2.2 Lexer Class

The `Lexer` class is responsible for scanning the source code and generating tokens. It contains methods for advancing the character position, skipping whitespace, and identifying different types of tokens such as integers, identifiers, and strings.

```
1 class Lexer:
2     def __init__(self, text):
3         # Initialization code
4
5     def error(self):
6         # Error handling code
7
8     def advance(self):
9         # Advance character position
10
11    def skip_whitespace(self):
12        # Skip whitespace characters
13
14    def peek(self):
15        # Peek ahead to the next character
16
17    def integer(self):
18        # Extract integer tokens
19
20    def identifier(self):
21        # Extract identifier tokens
22
23    def string(self):
24        # Extract string tokens
25
26    def get_next_token(self):
27        # Get the next token from the source code
```

2.3 Initialization

The `Lexer` class is initialized with the source code text as input. It sets up attributes such as the text itself, the current position in the text (`pos`), and the current character being processed (`current_char`). Additionally, it initializes a dictionary of keywords for easy identification during tokenization.

```
1 class Lexer:
2     def __init__(self, text):
3         self.text = text
4         self.pos = 0
5         self.current_char = self.text[self.pos]
6         self.keywords = {
7             'new': Token('NEW'),
8             'model': Token('MODEL'),
9             'true': Token('BOOLEAN', True),
10            'false': Token('BOOLEAN', False),
11        }
```

2.4 Advancing Method

The `advance()` method is responsible for moving the lexer's current position to the next character in the source code text. It updates the `pos` attribute and sets the `current_char` attribute accordingly. If the end of the text is reached, `current_char` is set to `None`.

```
1 def advance(self):
2     self.pos += 1
3     if self.pos < len(self.text):
4         self.current_char = self.text[self.pos]
5     else:
6         self.current_char = None
```

2.5 Peek Method

The `peek()` method allows the lexer to look ahead to the next character in the source code text without advancing the current position. It calculates the position of the next character (`peek_pos`) and returns that character if it exists. If the next character is beyond the end of the text, `None` is returned.

```
1 def peek(self):
2     peek_pos = self.pos + 1
3     if peek_pos < len(self.text):
4         return self.text[peek_pos]
5     else:
6         return None
```

2.6 Identifier Method

The `identifier()` method is responsible for extracting identifiers from the source code text. It reads characters sequentially until it encounters a non-alphanumeric character or underscore. This method captures both keywords and user-defined identifiers.

```
1 def identifier(self):
2     result = ''
3     while self.current_char is not None and (self.current_char.
4         isalnum() or self.current_char == '_'):
5         result += self.current_char
6         self.advance()
7     return result
```

2.7 Token Recognition Methods

Other methods within the `Lexer` class, such as `integer()`, `string()`, and `get_next_token()`, work similarly to the `identifier()` method. They are responsible for recognizing and extracting different types of tokens from the source code text. These methods follow the same pattern of iterating through the text, extracting relevant characters, and advancing the lexer's position accordingly.

3 Lexer Usage

To demonstrate the lexer in action, we provide an example expression and use the lexer to generate tokens for it.

```
1 def main():
2     # Example expression
3     expression = """new classifier = new model ("DecisionTree",
4         criterion="entropy", max_depth=5);
5     # More code...
6 if __name__ == '__main__':
7     main()
```

Running the `main()` function with the provided expression will produce a sequence of tokens representing each component of the code.

4 Console Results

```
1 Token(NEW, None)
2 Token(IDENTIFIER, classifier)
3 Token(ASSIGN, =)
4 Token(NEW, None)
5 Token(MODEL, None)
6 Token(LPAREN, ()
7 Token(STRING, DecisionTree )
8 Token(COMMA, ,)
9 Token(IDENTIFIER, criterion)
10 Token(ASSIGN, =)
11 Token(STRING, entropy )
12 Token(COMMA, ,)
13 Token(IDENTIFIER, max_depth)
14 Token(ASSIGN, =)
15 Token(NUMBER, 5)
16 Token(RPAREN, ))
17 Token(SEMI, ;)
18 Token(NEW, None)
19 Token(IDENTIFIER, regressor)
20 Token(ASSIGN, =)
21 Token(NEW, None)
22 Token(MODEL, None)
23 Token(LPAREN, ()
24 Token(STRING, RandomForestRegressor )
25 Token(COMMA, ,)
26 Token(IDENTIFIER, n_estimators)
27 Token(ASSIGN, =)
28 Token(NUMBER, 100)
29 Token(COMMA, ,)
30 Token(IDENTIFIER, max_depth)
31 Token(ASSIGN, =)
32 Token(NUMBER, 10)
33 Token(RPAREN, ))
34 Token(SEMI, ;)
35 Token(IDENTIFIER, classifier)
36 Token(DOT, .)
```

```

37 Token(IDENTIFIER, train)
38 Token(LPAREN, ())
39 Token(IDENTIFIER, X_train)
40 Token(COMMA, ,)
41 Token(IDENTIFIER, y_train)
42 Token(RPAREN, ))
43 Token(SEMI, ;)
44 Token(IDENTIFIER, classifier)
45 Token(DOT, .)
46 Token(IDENTIFIER, evaluate)
47 Token(LPAREN, ())
48 Token(IDENTIFIER, X_test)
49 Token(COMMA, ,)
50 Token(IDENTIFIER, y_test)
51 Token(RPAREN, ))
52 Token(SEMI, ;)
53 Token(IDENTIFIER, classifier)
54 Token(DOT, .)
55 Token(IDENTIFIER, save_model)
56 Token(LPAREN, ())
57 Token(String, classifier_model . pickle )
58 Token(RPAREN, ))
59 Token(SEMI, ;)
60 Token(IDENTIFIER, regressor)
61 Token(DOT, .)
62 Token(IDENTIFIER, train)
63 Token(LPAREN, ())
64 Token(IDENTIFIER, X_train)
65 Token(COMMA, ,)
66 Token(IDENTIFIER, y_train)
67 Token(RPAREN, ))
68 Token(SEMI, ;)
69 Token(IDENTIFIER, regressor)
70 Token(DOT, .)
71 Token(IDENTIFIER, evaluate)
72 Token(LPAREN, ())
73 Token(IDENTIFIER, X_test)
74 Token(COMMA, ,)
75 Token(IDENTIFIER, y_test)
76 Token(RPAREN, ))
77 Token(SEMI, ;)
78 Token(IDENTIFIER, regressor)
79 Token(DOT, .)
80 Token(IDENTIFIER, save_model)
81 Token(LPAREN, ())
82 Token(String, regressor_model . pickle )
83 Token(RPAREN, ))
84 Token(SEMI, ;)
85 Token(IDENTIFIER, model_loaded)
86 Token(ASSIGN, =)
87 Token(IDENTIFIER, load_model)
88 Token(LPAREN, ())
89 Token(String, classifier_model . pickle )
90 Token(RPAREN, ))
91 Token(SEMI, ;)
92 Token(IDENTIFIER, model_loaded)
93 Token(DOT, .)

```

```

94 Token(IDENTIFIER, evaluate)
95 Token(LPAREN, ())
96 Token(IDENTIFIER, X_test)
97 Token(COMMA, ,)
98 Token(IDENTIFIER, y_test)
99 Token(RPAREN, ))
100 Token(SEMI, ;)
101 Token(IDENTIFIER, is_classifier)
102 Token(ASSIGN, =)
103 Token(IDENTIFIER, model_loaded)
104 Token(DOT, .)
105 Token(IDENTIFIER, is_classifier)
106 Token(LPAREN, ())
107 Token(RPAREN, ))
108 Token(SEMI, ;)
109 Token(IDENTIFIER, is_regressor)
110 Token(ASSIGN, =)
111 Token(IDENTIFIER, regressor)
112 Token(DOT, .)
113 Token(IDENTIFIER, is_regressor)
114 Token(LPAREN, ())
115 Token(RPAREN, ))
116 Token(SEMI, ;)
117 Token(IDENTIFIER, print)
118 Token(LPAREN, ())
119 Token(STRING, Is the loaded model a classifier ?)
120 Token(COMMA, ,)
121 Token(IDENTIFIER, is_classifier)
122 Token(RPAREN, ))
123 Token(SEMI, ;)
124 Token(IDENTIFIER, print)
125 Token(LPAREN, ())
126 Token(STRING, Is the regressor a classifier ?)
127 Token(COMMA, ,)
128 Token(IDENTIFIER, is_regressor)
129 Token(RPAREN, ))
130 Token(SEMI, ;)

```

5 Conclusion

In conclusion, the lexer presented here is a crucial component of any compiler or interpreter, responsible for breaking down the source code into tokens for further processing. By understanding its implementation and functionality, we gain insight into the initial stages of the compilation process.