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.

```python
class Token:
    def __init__(self, type, value=None):
        self.type = type
        self.value = value

    def __str__(self):
        return f'Token({self.type}, {self.value})'

    def __repr__(self):
        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.

```
class Lexer:
    def __init__(self, text):
        # Initialization code

    def error(self):
        # Error handling code

    def advance(self):
        # Advance character position

    def skip_whitespace(self):
        # Skip whitespace characters

    def peek(self):
        # Peek ahead to the next character

    def integer(self):
        # Extract integer tokens

    def identifier(self):
        # Extract identifier tokens

    def string(self):
        # Extract string tokens

    def get_next_token(self):
        # 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.

```
class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0
        self.current_char = self.text[self.pos]
        self.keywords = {
            ’new’: Token(’NEW’),
            ’model’: Token(’MODEL’),
            ’true’: Token(’BOOLEAN’, True),
            ’false’: Token(’BOOLEAN’, False),
        }
```

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`.

```python
def advance(self):
    self.pos += 1
    if self.pos < len(self.text):
        self.current_char = self.text[self.pos]
    else:
        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.

```python
def peek(self):
    peek_pos = self.pos + 1
    if peek_pos < len(self.text):
        return self.text[peek_pos]
    else:
        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.

```python
def identifier(self):
    result = ''
    while self.current_char is not None and (self.current_char.isalnum() or self.current_char == '_'):
        result += self.current_char
        self.advance()
    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.

```python
def main():
    # Example expression
    expression = """new classifier = new model ("DecisionTree",
    criterion="entropy", max_depth=5);
    # More code...

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

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

4 Console Results

```
Token(NEW, None)
Token(IDENTIFIER, classifier)
Token(ASSIGN, =)
Token(NEW, None)
Token(MODEL, None)
Token(LPAREN, ()
Token(STRING,  DecisionTree )
Token(COMMA, ,)
Token(IDENTIFIER, criterion)
Token(ASSIGN, =)
Token(STRING,  entropy )
Token(COMMA, ,)
Token(IDENTIFIER, max_depth)
Token(ASSIGN, =)
Token(NUMBER, 5)
Token(RPAREN, ))
Token(SEMI, ;)
Token(NEW, None)
Token(IDENTIFIER, regressor)
Token(ASSIGN, =)
Token(NEW, None)
Token(MODEL, None)
Token(LPAREN, ()
Token(STRING,  RandomForestRegressor )
Token(COMMA, ,)
Token(IDENTIFIER, n_estimators)
Token(ASSIGN, =)
Token(NUMBER, 100)
Token(COMMA, ,)
Token(IDENTIFIER, max_depth)
Token(ASSIGN, =)
Token(NUMBER, 10)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, classifier)
Token(DOT, .)
Token(IDENTIFIER, train)
Token(LPAREN, ()
Token(IDENTIFIER, X_train)
Token(COMMA, ,)
Token(IDENTIFIER, y_train)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, classifier)
Token(DOT, .)
Token(IDENTIFIER, evaluate)
Token(LPAREN, ()
Token(IDENTIFIER, X_test)
Token(COMMA, ,)
Token(IDENTIFIER, y_test)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, classifier)
Token(DOT, .)
Token(IDENTIFIER, save_model)
Token(LPAREN, ()
Token(STRING,  classifier_model . pkl )
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, regressor)
Token(DOT, .)
Token(IDENTIFIER, train)
Token(LPAREN, ()
Token(IDENTIFIER, X_train)
Token(COMMA, ,)
Token(IDENTIFIER, y_train)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, regressor)
Token(DOT, .)
Token(IDENTIFIER, evaluate)
Token(LPAREN, ()
Token(IDENTIFIER, X_test)
Token(COMMA, ,)
Token(IDENTIFIER, y_test)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, regressor)
Token(DOT, .)
Token(IDENTIFIER, save_model)
Token(LPAREN, ()
Token(STRING,  regressor_model . pkl )
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, model_loaded)
Token(ASSIGN, =)
Token(IDENTIFIER, load_model)
Token(LPAREN, ()
Token(STRING,  classifier_model . pkl )
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, model_loaded)
Token(DOT, .)
Token(IDENTIFIER, evaluate)
Token(LPAREN, ()
Token(IDENTIFIER, X_test)
Token(COMMA, ,)
Token(IDENTIFIER, y_test)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, is_classifier)
Token(ASSIGN, =)
Token(IDENTIFIER, model_loaded)
Token(DOT, .)
Token(IDENTIFIER, is_classifier)
Token(LPAREN, ()
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, is_regressor)
Token(ASSIGN, =)
Token(IDENTIFIER, regressor)
Token(DOT, .)
Token(IDENTIFIER, is_regressor)
Token(LPAREN, ()
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, print)
Token(LPAREN, ()
Token(STRING,  Is the loaded model a classifier ?)
Token(COMMA, ,)
Token(IDENTIFIER, is_classifier)
Token(RPAREN, ))
Token(SEMI, ;)
Token(IDENTIFIER, print)
Token(LPAREN, ()
Token(STRING,  Is the regressor a classifier ?)
Token(COMMA, ,)
Token(IDENTIFIER, is_regressor)
Token(RPAREN, ))
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.