First Project: Lexer

The first project requires you to implement a scanner for the language, specified by MiniJava BNF Grammar notebook. Study the specification of MiniJava grammar carefully. To complete this first project, you will use the SLY, a Python version of the lex/yacc toolset with same functionality but with a friendlier interface. Please read the complete contents of this section and carefully complete the steps indicated.

Regular Expressions

Regular expressions are concise ways of describing a set of strings that meet a given pattern. For example, we can specify the regular expression:

```
r'[a-zA-Z_][0-9a-zA-Z_]*'
```

to describe valid identifiers in the MiniJava language. Regular expressions are a mini-language that lets you specify the rules for constructing a string set. This specification mini-language is very similar between the different programming languages that contain the concept of regular expressions (also called RE or REGEX). Thus, learning to write regular expressions in Python will also be useful for describing REs in other programming languages.

You can use this Python code as a model to test your regular expressions:

```
import re
# regular expression
identifier = r'[a-zA-Z_][0-9a-zA-Z_]*'
# test identifier _123
b = re.match(identifier, "_123")
if b:
    print("Identifier matches.")
else:
    print("Error.")
```

Writing a Lexer

The process of “lexing” is that of taking input text and breaking it down into a stream of tokens. Each token is like a valid word from the dictionary. Essentially, the role of the lexer is to simply make sure that the input text consists of valid symbols and tokens prior to any further processing related to parsing.

Each token is defined by a regular expression. Thus, your task here is to define a set of regular expressions for the MiniJava language. The actual job of lexing will be handled by SLY. For a better understanding study the Lex section in the SLY documentation.

Specification

Your lexer must recognize the symbols and tokens of MiniJava Grammar. For instance, in the example below, the name on the left is the token name, and the value on the right is the matching text:

Reserved Keywords:

```
FOR   : 'for'
IF    : 'if'
PRINT : 'print'
```

Identifiers:

```
ID    : any text starting with a letter or '_', followed by any number of le
        digits, or underscores, that is not a reserved word.
```

Some Operators and Delimiters:

```
PLUS    : '+'
MINUS   : '-'
TIMES   : '*'
DIVIDE  : '/'
ASSIGN  : '='
SEMI    : ';'
LPAREN  : '('
RPAREN  : ')'
```

Literals:

```
INT_LITERAL : 123
CHAR_LITERAL : 'a'
STRING_LITERAL : "Hello World\n"
```

For `INT_LITERAL`, you should only consider decimal numbers.

Comments: To be ignored by your lexer

```
//              Skips the rest of the line
/* ... */       Skips a block (no nesting allowed)
```

Lex Skeleton

```python
import argparse
import pathlib
import sys

from sly import Lexer


class MJLexer(Lexer):
    """A lexer for the MiniJava language. After building it, set the
    input text with input(), and call token() to get new
    tokens.
    """

    def __init__(self, error_func):
        """Create a new Lexer.
        An error function. Will be called with an error
        message, line and column as arguments, in case of
        an error during lexing.
        """
        self.error_func = error_func
        self.filename = ""

        # Keeps track of the last token returned from self.token()
        self.last_token = None

    def _error(self, msg, token):
        location = self._make_tok_location(token)
        self.error_func(msg, location[0], location[1])
        self.index += 1

    def find_tok_column(self, token):
        """Find the column of the token in its line."""
        last_cr = self.text.rfind("\n", 0, token.index)
        return token.index - last_cr
    def _make_tok_location(self, token):
        return (self.lineno, self.find_tok_column(token))

    # Error handling
    def error(self, t):
        msg = f"Illegal character {t.value[0]!r}"
        self._error(msg, t)

    # Scanner (used only for test)
    def scan(self, data):
        output = ""
        for token in self.tokenize(data):
            token = (
                f"LexToken({token.type},'{token.value}',{token.lineno},{token.in
            )
            print(token)
            output += token + "\n"
        return output

    # Set of token names.
    tokens = {
        # Reserved Keywords
        "CLASS",
        "EXTENDS",
        "PUBLIC",
        "STATIC",
        "VOID",
        "MAIN",
        "STRING",
        "BOOLEAN",
        "CHAR",
        "INT",
        "IF",
        "ELSE",
        "WHILE",
        "FOR",
        "ASSERT",
        "BREAK",
        "RETURN",
        "NEW",
        "THIS",
        "TRUE",
        "FALSE",
        "LENGTH",
        "PRINT",
        # Literals
        "ID",
        "INT_LITERAL",
        "CHAR_LITERAL",
        "STRING_LITERAL",
        # Operators
        "EQ",
        "NE",
        "LE",
        "GE",
        "AND",
        "OR",
        "ASSIGN",
        "LT",
        "GT",
        "PLUS",
        "MINUS",
        "TIMES",
        "DIVIDE",
        "MOD",
        "NOT",
        # Punctuation
        "DOT",
        "SEMI",
        "COMMA",
        "LPAREN",
        "RPAREN",
        "LBRACKET",
        "RBRACKET",
        "LBRACE",
        "RBRACE",
    }

    # ------------------------------------------------------------------
    # Identifiers and reserved words
    # ------------------------------------------------------------------
    # A dictionary mapping reserved words to token types.
    keywords = {
        "class": "CLASS",
        "extends": "EXTENDS",
        "public": "PUBLIC",
        "static": "STATIC",
        "void": "VOID",
        "main": "MAIN",
        "String": "STRING",
        "boolean": "BOOLEAN",
        "char": "CHAR",
        "int": "INT",
        "if": "IF",
        "else": "ELSE",
        "while": "WHILE",
        "for": "FOR",
        "assert": "ASSERT",
        "break": "BREAK",
        "return": "RETURN",
        "new": "NEW",
        "this": "THIS",
        "true": "TRUE",
        "false": "FALSE",
        "length": "LENGTH",
    }



    # ------------------------------------------------------------------
    # Rules
    # ------------------------------------------------------------------

    # String containing ignored characters (spaces and tabs)
    ignore = " \t"

    # Newlines
    @_(r"<Include a regex here for newline>")
    def ignore_newline(self, t):
        self.lineno += len(t.value)

    # Comments
    @_(r"<Include a regex here for comments>")
    def ignore_comment(self, t):
        self.lineno += t.value.count('\n')

    # Identifiers and keywords
    @_(r"<Include a regex here for ID>")
    def ID(self, t):
        t.type = self.keywords.get(t.value, "ID")
        return t

    # ------------------------------------------------------------------
    # Operators and punctuation (order matters: longer tokens first)
    # ------------------------------------------------------------------
    PLUS = r"\+"   # Regex special characters must be escaped
    MINUS = r"-"
```

```
    # Continue the Lexer Rules
    # ...
```

Testing

For initial development, try running the lexer on a sample input file such as:

```
/* comment */
class VariableTest {
    int j = 3;
    public static void main(String[] args) {
        int i = this.j;
        int k = 3;
        int p = 2 * this.j;

        assert p == 2 * i;
    }
}
```

And the result will look similar to the text shown below.

```
LexToken(CLASS,'class',2,14)
LexToken(ID,'VariableTest',2,20)
LexToken(LBRACE,'{',2,33)
LexToken(INT,'int',3,39)
LexToken(ID,'j',3,43)
LexToken(ASSIGN,'=',3,45)
LexToken(INT_LITERAL,'3',3,47)
LexToken(SEMI,';',3,48)
LexToken(PUBLIC,'public',4,54)
LexToken(STATIC,'static',4,61)
LexToken(VOID,'void',4,68)
LexToken(MAIN,'main',4,73)
LexToken(LPAREN,'(',4,77)
LexToken(STRING,'String',4,78)
LexToken(LBRACKET,'[',4,84)
LexToken(RBRACKET,']',4,85)
LexToken(ID,'args',4,87)
LexToken(RPAREN,')',4,91)
LexToken(LBRACE,'{',4,93)
LexToken(INT,'int',5,103)
LexToken(ID,'i',5,107)
LexToken(ASSIGN,'=',5,109)
LexToken(THIS,'this',5,111)
LexToken(DOT,'.',5,115)
LexToken(ID,'j',5,116)
LexToken(SEMI,';',5,117)
LexToken(INT,'int',6,127)
LexToken(ID,'k',6,131)
LexToken(ASSIGN,'=',6,133)
LexToken(INT_LITERAL,'3',6,135)
LexToken(SEMI,';',6,136)
LexToken(INT,'int',7,146)
LexToken(ID,'p',7,150)
LexToken(ASSIGN,'=',7,152)
LexToken(INT_LITERAL,'2',7,154)
LexToken(TIMES,'*',7,156)
LexToken(THIS,'this',7,158)
LexToken(DOT,'.',7,162)
LexToken(ID,'j',7,163)
LexToken(SEMI,';',7,164)
LexToken(ASSERT,'assert',9,175)
LexToken(ID,'p',9,182)
LexToken(EQ,'==',9,184)
LexToken(INT_LITERAL,'2',9,187)
LexToken(TIMES,'*',9,189)
LexToken(ID,'i',9,191)
LexToken(SEMI,';',9,192)
LexToken(RBRACE,'}',10,198)
LexToken(RBRACE,'}',11,200)
```

Here is an example of accessing a global function using an object:

```
class ObjExample {
    int n = 3;

    public int doubleMe(int x) {
        return x * x;
    }

    public static void main(String[] args) {
        Main obj = new Main();
        int v = obj.n;
        v = obj.doubleMe(v);
        assert v == obj.n * obj.n;
    }
}
```

Carefully study the output of the lexer and make sure that it makes sense. Once you are reasonably happy with the output, try running some of the more tricky tests designed to stress test various corner cases. The repository provided as a base to implement the project contains a large set of tests to verify your code: check them to see more examples.

Here is another example:

```
class ArrayTest {
    public static void main(String[] args) {
        int[] v = {1, 3, 5, 7, 9};
        assert v[3] == 7;
    }
}
```

```
LexToken(CLASS,'class',1,0)
LexToken(ID,'ArrayTest',1,6)
LexToken(LBRACE,'{',1,16)
LexToken(PUBLIC,'public',2,22)
LexToken(STATIC,'static',2,29)
LexToken(VOID,'void',2,36)
LexToken(MAIN,'main',2,41)
LexToken(LPAREN,'(',2,45)
LexToken(STRING,'String',2,46)
LexToken(LBRACKET,'[',2,52)
LexToken(RBRACKET,']',2,53)
LexToken(ID,'args',2,55)
LexToken(RPAREN,')',2,59)
LexToken(LBRACE,'{',2,61)
LexToken(INT,'int',3,71)
LexToken(LBRACKET,'[',3,74)
LexToken(RBRACKET,']',3,75)
LexToken(ID,'v',3,77)
LexToken(ASSIGN,'=',3,79)
LexToken(LBRACE,'{',3,81)
LexToken(INT_LITERAL,'1',3,82)
LexToken(COMMA,',',3,83)
LexToken(INT_LITERAL,'3',3,85)
LexToken(COMMA,',',3,86)
LexToken(INT_LITERAL,'5',3,88)
LexToken(COMMA,',',3,89)
LexToken(INT_LITERAL,'7',3,91)
LexToken(COMMA,',',3,92)
LexToken(INT_LITERAL,'9',3,94)
LexToken(RBRACE,'}',3,95)
LexToken(SEMI,';',3,96)
LexToken(ASSERT,'assert',5,131)
LexToken(ID,'v',5,138)
LexToken(LBRACKET,'[',5,139)
LexToken(INT_LITERAL,'3',5,140)
LexToken(RBRACKET,']',5,141)
LexToken(EQ,'==',5,143)
LexToken(INT_LITERAL,'7',5,146)
LexToken(SEMI,';',5,147)
LexToken(RBRACE,'}',6,185)
LexToken(RBRACE,'}',7,187)
```

The full list of tokens used by MiniJava is

```
    # Set of token names.
    tokens = {
        # Reserved Keywords
        "CLASS",
        "EXTENDS",
        "PUBLIC",
        "STATIC",
        "VOID",
        "MAIN",
        "STRING",
        "BOOLEAN",
        "CHAR",
        "INT",
        "IF",
        "ELSE",
        "WHILE",
        "FOR",
        "ASSERT",
        "BREAK",
        "RETURN",
        "NEW",
        "THIS",
        "TRUE",
        "FALSE",
        "LENGTH",
        "PRINT",
        # Literals
        "ID",
        "INT_LITERAL",
        "CHAR_LITERAL",
        "STRING_LITERAL",
        # Operators
        "EQ",
        "NE",
        "LE",
        "GE",
        "AND",
        "OR",
        "ASSIGN",
        "LT",
        "GT",
        "PLUS",
        "MINUS",
        "TIMES",
        "DIVIDE",
        "MOD",
        "NOT",
        # Punctuation
        "DOT",
        "SEMI",
        "COMMA",
        "LPAREN",
        "RPAREN",
        "LBRACKET",
        "RBRACKET",
        "LBRACE",
        "RBRACE",
    }
```