Nathan Evarts - Rules for Lexical and Syntactical Analysis

The practical portion of my exam was written in Python. It analyzes code written inside an IDE text file, and then prints the lexemes, along with any syntax errors that might occur inside the file.

This is what the file looks like with all the functions minimized. There are a few unorthodox imports - “re” and “nltk.” These imports helped with the lexical analysis of the file.

The rules for which the syntax is adhered to is **Java**. I figured java was easier than C because I’ve written more of it and understand the rules a bit more.

There are some assumptions to be made about the IDE text file. The syntactical analysis will assume that arguments and statements are excluded from the syntactical correctness of the skeleton of each function. This means that it checks for the skeleton of the function to be correct, and excludes any arguments inside parentheses or brackets. That being said, it cannot ignore the lexemes that the lexical analyser has parsed. **Therefore, it will return an error**. Here is an example.

```
Next lexem is:  for
<FOR FUNCTION ENTERED>
Next lexem is:  (
<ARGS>
Next lexem is:  )
Next lexem is:  {
<BLOCK ENTERED>
Next lexem is:  <
ERROR - '}' MISSING>
```

for ( ) {

<block>

}

The following is the output of the code to the right of it. However, if “<block>” is removed, it correctly parses that there are no errors in the skeleton of the code.

for ( ) {

}

```
Next lexem is:  for
<FOR FUNCTION ENTERED>
Next lexem is:  (
<ARGS>
Next lexem is:  )
Next lexem is:  {
<BLOCK ENTERED>
Next lexem is:  }
<BLOCK EXITED>
```

The same rules are applied to each function in the program.

```
def forFunction(queue):
    print("<FOR FUNCTION ENTERED>")
    lex = queue.pop(0)
    print("Next lexem is: ", lex)
    if(lex != "("):
        print("<ERROR - '(' MISSING>")
    else:
        print("<ARGS>")
        lex = queue.pop(0)
        print("Next lexem is: ", lex)
        if(lex != ")"):
            print("<ERROR - ')' MISSING>")
        else:
            lex = queue.pop(0)
            print("Next lexem is: ", lex)
            if(lex != "{"):
                print("ERROR - '{' MISSING>")
            else:
                #when the block in entered, it is assumed that there are arguments within the block.
                #entering arguments inside the IDE.txt enclosed in { } will cause an error.
                print("<BLOCK ENTERED>")
                lex = queue.pop(0)
                print("Next lexem is: ", lex)
                if(lex != "}"):
                    print("ERROR - '}' MISSING>")
                else:
                    print("<BLOCK EXITED>")
```

The rule was written in the comments of the code as well. This is an example of what each function looks like, as they are all quite similar.

How the Program Works

The program starts by reading the data from the text file.

```
#read txt file to string
with open ("ide.txt", "r") as myfile:
    data = myfile.read()
```

The data is then parsed into tokens using the handy nltk import that I was talking about earlier.

```
#tokenize the string
tokens = nltk.wordpunct_tokenize(data)
```

The tokens are then appended into a list, where I can easily access them, and pop them when I need to. When a lexem matches a keyword for one of the functions, it is passed to the function that will properly analyse the syntax according to the rules of that function. When the function for each keyword is called, the queue is passed to that function, so that the analyzer can pop from it and analyze each lexem coming after it.

```
queue = []

for i in tokens:
    queue.append(i)

while(len(queue) != 0):
    lex = queue.pop(0)
    print("Next lexem is: ", lex)

    if(lex == keywordDict[0]):
        forFunction(queue)
        break

    if(lex == keywordDict[1]):
        ifFunction(queue)
        break

    if(lex == "else"):
        print('ERROR - CANNOT ELSE WITHOUT IF')

    if(lex == keywordDict[3]):
        whileFunction(queue)
        break

    if(lex == keywordDict[4]):
        doFunction(queue)
        break

    if(lex == keywordDict[7]):
        switchFunction(queue)
        break
```