Topic: Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

Course: Formal Languages & Finite Automata

Objectives:

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

Laboratory work:

1.

Lexical analysis (tokenization) is the process of converting a sequence of characters into a sequence of lexical tokens strings with an assigned meaning. It's an important part of the compiler design process. Compilers use lexical analysis to convert human-readable code into a form that a computer can understand and execute. The main purpose of lexical analysis is to perform analysis of the source code, checking for any lexical errors or issues before they can cause more serious problems later in the compiler design process. This includes checking for things like incorrect spelling, incorrect punctuation, and other errors that could impact the readability of the source code.

Implementing lexical analysis involves writing a lexical analyzer, also known as a lexer or scanner, which scans the input text and recognizes the tokens.

How it works is: we have a sequence of characters that we convert into a sequence of tokens (an actual meaning. Like how we don't imagine the characters of a word but the meaning of it). There are predefined rules for every lexeme to be identified as a valid tokn. These rules are defined by grammar rules. In the code, we categorize the tokens as of type KEYWORD, SEPARATOR, IDENTIFIER, OPERATOR, LITERAL. I didn't regard the white spaces and comments.

2.

How tokens are identified in my program:

The lexer receives an input string. The lexer breaks down the input string into smaller units called tokens. The lexer utilizes regular expressions to define patterns for different types of tokens by using Matcher object to search occurences of patterns. When a match is found, the lexer extracts the matched text from the input string. Further the lexer needs to determine the type of token it represents. Since some tokens might share common prefixes, the lexer keeps track of the longest match found so far and its associated token type. Once the longest match for a token type is determined, the lexer creates a `Token` object representing that match, which contains the "type" and "match" information. As tokens are identified and created, the lexer

stores them in a list or some other data structure for further processing or analysis. Then continues the same process till the whole file input is scanned. After tokenizing the entire input string, the lexer outputs the list of tokens.

Patterns for creating tokens:

KEYWORD

To recognise a keyword, we delimit the predefined keywords by "|". When the code executes, it will compile the regular expression pattern that matches one/any of the keywords.

```
Matcher matcher = Pattern.compile("^(" + String.join("|", KEYWORDS) + ")").matcher(input);
```

The "^" prevents the code to read the same word multiple times. Even if we have the keywords identified, we still need to make sure they are not a substring but a full word.

SEPARATOR

To recognise an operator, we specify which are the operators

```
matcher = Pattern.compile("^[=<>]").matcher(input);
```

It extracts the match and if the current match is longer than previous match, it updates largestMatch with the current match. If it is the longest, it will be an OPERATOR.

IDENTIFIER

We define which are the separators, scan through them from input, find match, extract it, check the length, and decide if its an IDENTIFIER.

```
matcher = Pattern.compile("^[_A-Za-z]+").matcher(input);
```

OPERATOR

To recognise an operator, we specify which are the operators

matcher = Pattern.compile("^[=<>]").matcher(input);

It extracts the match and if the current match is longer than previous match, it updates largestMatch with the current match. If it is the longest, it will be an OPERATOR.

LITERAL

To recognise them, we specify which are they

```
matcher = Pattern.compile("^\"([^\"]*)\"|^(true|false)|^[-+]?[0-9]*
```

The code extracts the match and if the current match is longer then previous match, it updates largestMatch with the current match if it is the longest, it will be of type LIRETAL.

3.

Implementation

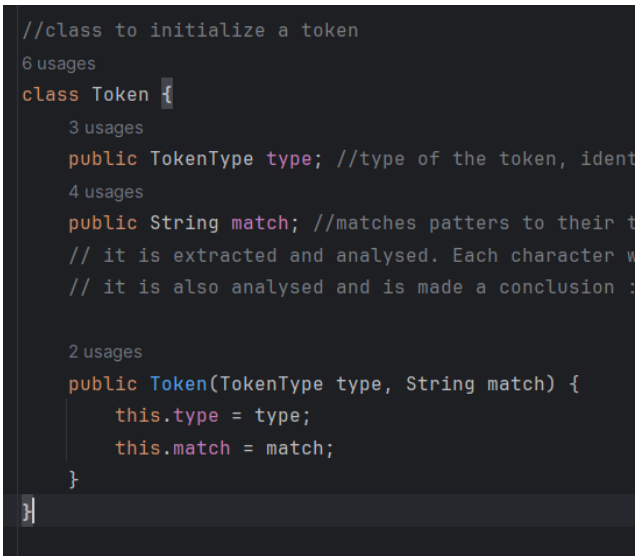
1. We “feed” the code an input file:

```java
public class Adder {

    public static void main(String[] args) {

        double n1 = -4.5, n2 = 3.9, n3 = 2.5;

        if( n1 >= n2 && n1 >= n3)
            System.out.println(n1 + "is the largest number.");

        else if (n2 >= n1 && n2 >= n3)
            System.out.println(n2 + "is the largest number.");

        else
            System.out.println(n3 + "is the largest number.");
    }

}
```

I implemented a class to identify the types of tokens:

```
13 usages
enum TokenType {
    5 usages
    NONE, //when the
    2 usages
    KEYWORD, //when
    2 usages
    SEPARATOR, //wher
    2 usages
    IDENTIFIER, //whe
    2 usages
    OPERATOR,  //wher
    2 usages
    LITERAL, //when
}
```

I implemented a class to initialize a token:



The type is initialized for describing the type of token, identifier, operator, separator, real, keyword, none. The match matches patters to their tokens.

Later, in Lexer class, when a match is found, it is extracted and analyzed. Each character will be analyzed and put in ***largest match***. If there is a character after it, it is also analyzed and is made a conclusion: is it still an identifier or actually an actually another type? Ex. an operator? Or is it a spelling error? Then it is checked if the current match is longer than the previously stored ***largest match***. If it is, it updates ***largest match*** with the current match, indicating that it is the longest match found so far.

Other unrelated stuff the class does is : we scan the file, read every line of it, we tokenize the input by adding each token to a list that will be later printed, and remove the matched text from the input.

To represent the method of tokenizing, let's take the example of how the program finds the keywords. First, we delimit the keywords by "|". When the code executes, it will compile the regular expression pattern that matches one/any of the keywords

```
Matcher matcher = Pattern.compile("^(" + String.join("|", KEYWORDS) + ")").matcher(input);
```

The "^" prevents the code to read the same word multiple times.

```
if (matcher.find()) {
    String match = matcher.group();
```

Even if we have the keywords identified, we still need to make sure they are not a substring but a full word, that's why we use the *largestMatch*. For instance, if we find the input string "***ift***" and suppose the KEYWORDS array contains only the keyword "***if***", without checking for the longest match, the lexer might mistakenly identify "***if***" as a keyword and tokenize it accordingly. But by greedily finding the longest match, the lexer can correctly identify that "***ift***" does not match any keyword entirely and thus should not be tokenized as a keyword.

```
    if (match.length() > largestMatch.length()) {
                largestMatch = match;
                largestMatchType = TokenType.KEYWORD;
            }
    }
```

Therefore, the output of the program looks like this:

```
literal         '3'
separator         ')'
identifier          'System'
identifier          'the'
identifier          'largest'
identifier          'number'
keyword         'else'
keyword         'if'
separator         '('
identifier          'n'
literal         '2'
operator         '>'
operator         '='
identifier          'n'
literal         '1'
identifier          'n'
literal         '2'
operator         '>'
operator         '='
identifier          'n'
literal         '3'
separator         ')'
```

Conclusion:

Tokenising is an important step in the building a compiler as it provides meanings for the characters written. The output is a sequence of tokens that is sent to the parser for syntax analysis. Like this, compilers get to figure out what the code is supposed to do. Lexical analysis will detect lexical errors such as misspelled keywords or undefined symbols early in the compilation process. Once the source code is converted into tokens, subsequent phases of compilation or interpretation can operate more efficiently. Parsing and semantic analysis become faster and more streamlined when working with tokenized input.