Advanced Topics – Compilers and Interpreters

Lab: Lexical Analysis

In this lab, we will build one of the key components that make up both compilers and interpreters: the input scanner. The input scanner is responsible for reading the input string, determining the individual lexemes according to a given set of rules, and producing a string of tokens. Usually a parser will consume the token string.

In addition to lexical analysis of the input, the scanner usually performs other functions such as eliminating comments. (Comments are for people to read, we are transforming our input program into a program for the machine to execute. The code you write is for people to read!) .

Our scanner will be a simple scanner that returns the actual lexemes as tokens. It is an example of a hand-crafted scanner, and simplicity will be the guiding requirement. Even so, our scanner will serve us well during the remainder of the course.

We will call our scanner `scanner` and it will be in a class called `Scanner.`

In this lab, as in all labs this semester, you are required to conform to the style guide. You will, however, be allowed to use the primitive `char` class. You **may not** use Java built in tokenizers or scanners (`StringTokenizer, StreamTokenizer` for instance). You may use only `String` class methods that are part of the approved subset. And you may not use the Java `Character` class.

Exercise: Tokenizing the Input

The `Scanner` class maintains three instance variables: `eof` which indicates that the input stream is at the end of file, `in` the input stream of characters and `currentChar` the character currently being examined. The instance field `currentChar` is of type `char`, a primitive data type in Java that represents a single character. The `char` type is useful because it enables us to write code like the following, which tests if `currentChar` is one of the digits 0-9. (Note that `char` literals are enclosed in single-quotes, to distinguish them from `String` literals.)

```
return currentChar >= '0' && currentChar <= '9';
```

Let's suppose we're reading characters, hoping to determine the next token. So far, we've read the characters `'I'` and `'F'`. What token do these characters represent?

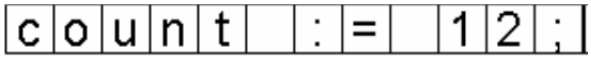
Answer: We don't know. It all depends on the next character. This might be the first part of a variable name, like `"IF2SLOW"`. Alternatively, if the next character is a space, then we can conclude that we've reached the end of the token representing the *keyword* `"IF"`.

STOP:

What if the next character had been a newline? An open parenthesis? (You are responsible for answering these questions. Your answer must be in your notebook, in proper English, and using complete sentences. You will lose a substantial number of points for not having an answer. Translation: you cannot earn an "A" grade of any sort without a properly constructed answer ready for your teacher to grade. You must write your answer **before** going on.)

AFTER YOU HAVE ANSWERED:

Therefore, in order to return a token, we always need to read one character after the token. Thus, when our `Scanner` is constructed, it immediately reads the first character into `currentChar`, before the `nextToken` method is ever called. The value of `currentChar` will always be one character ahead of the token just returned. Suppose we're reading the following characters:



Upon initializing the `Scanner`, `currentChar` will be `'c'`. The first call to `nextToken` will return `"count"`, and `currentChar` will now be `' '`. The second call to `nextToken` will return `":="`, and `currentChar` will now be `' '`. The third call to `nextToken` will return `"12"`, and `currentChar` will now be `';'`. Incidentally, currently our `Scanner` can't tell the difference between a variable name, a procedure name, etc., and we'll just refer to all of these as *identifiers*. Later, you may decide to implement a fancier `Scanner` that returns tokens consisting of more than just the lexeme.

Implement the (private) helper method `getNextChar`. The method `getNextChar` sets the instance field `currentChar` to the value read from the input stream using the stream `read` method. Note that `read` returns an `int` value and you will need to typecast appropriately. Also note that the `read` method will return -1 when the input stream is at end of file, and this fact should be used to set the instance filed `eof`. Also, you should add a public method called `hasNext` to your Scanner which returns false if the input stream is at end of file.

Other than the constructor, the only other method that will ever call `getNextChar` is the helper method `eat`. The method `getNextChar` should be private.

Note that you will need to handle the possibility of a catastrophic `IOErrorException` in the method `getNextChar.` You should handle this error using a `try-catch-finally` block and abort the program if the exception occurs.

Now, implement the (private) helper method `eat` which takes in a `char` value representing the expected value of `currentChar.` The method compares the value of the input parameter to `currentChar` and if they are the same, it advances the input stream one character by calling `getNextChar`. If the values differ, the method should throw a `ScanErrorException` with a message similar to, "Illegal character - expected `<currentChar>` and found `<char>".`

Be sure to carefully document your methods according to the style guide prior to writing the methods. Of course, only the most competent students (and software engineers) will document and create a thorough test of these methods *before* proceeding.

Exercise: Letters, Digits and White Space

Write the helper method `isDigit` which takes in a `char` and returns a `boolean`. The method examines the input `char` to see if it is a digit according to the regular expression:

digit := [0-9]

This is useful, and should be public and static.

Write the helper methods `isLetter` and `isWhiteSpace` that take in a `char` and produces a `boolean` according to the regular expressions:

letter := [a-z A-Z]
white space := [' ' '\t' '\r' '\n']

These methods should also be `public` and `static`.

Be sure to comment and test your methods *before* continuing.

Exercise: Actions

Our simple Scanner is only capable of recognizing three different types of tokens: numbers, identifiers, and operands. The regular expressions defining each type of token are given below.

number := digit(digit)*
identifier := letter (letter | digit)*
operand := ['=' '+' '-' '*' '/' '%' '(' ')']

The scanning strategy is to look ahead one character in the buffer, and use that character to direct the scanning process. For example, if the character is a digit, then the `Scanner` should parse a

digit; if it is a letter then the `Scanner` should parse an identifier; otherwise it should attempt to match an operand. If the match is unsuccessful, the `Scanner` should report an error.

Note that this `Scanner` is really simplified. It returns the actual lexeme with no additional information. With this `Scanner`, the parser will have to do additional work. Also, note that the `Scanner`, as it is defined, does not deal with comments – something you will need to fix!

Write the three (private) `Scanner` methods `scanNumber, scanIdentifier,` and `scanOperand` which scan the input and return a `String` representing the lexeme found in the input stream or throws a `ScanErrorException` if no lexeme is recognized.

Finally, write the public method `nextToken` which skips any leading white space and then examines the value of `currentChar,` calling the appropriate methods to scan the next token in the input stream. The method `nextToken` returns a `String` representing the lexeme found. The method `nextToken` should return the value `"END"` if the input stream is at end-of-file when `nextToken` is called.

Be sure to properly document and adequately test this class. If your `Scanner` has errors, you will have a lot of difficulty with the follow-on lab assignments.

If you finish early….

- Add support for single line comments starting with //. The `Scanner` should read and ignore comments since they are not needed for the following stages.

- Add support for block comments consisting of a string surrounded by /* and */.

- Really impress me and add support for nested block comments (/* /* */ */). You must create a separate design document for this.

- Design a more powerful `Scanner` that adds additional information beyond the lexeme and is capable of scanning and identifying keywords. You may define your own set of keywords for this exercise. Your design documentation must include the definition of all regular expressions used as well as documentation for any finite state automata that you use.

- Separate out special characters and operators and different types of lexemes/tokens.