Compilers
Homework #2
Due: Friday, September 16, 2022

Reading

Time to start reading about the next component of a compiler - grammars and parsing:

Section 4.1: Syntax Analysis (Introduction)

Section 4.2: Context-Free Grammars

Lexical Analysis of TOY

Starting with your token scanner from Homework #1, you should build a lexical analyzer which will read a text file and break it into its constituent tokens. Your lexical analysis class should have a constructor which accepts a file (or input stream such as standard input) and has a method

```
public Token getNextToken() { … }
```

which can be called repeatedly to break the input file into tokens. Each call to `getNextToken` should return the token which corresponds to the longest possible sequence of characters (e.g., ++ instead of +). It must skip over any white space (spaces, tabs, newlines, returns, etc AND Java style comments: // to end of line).

You should read the input file one line at a time and use instance variables in your lexical analyzer class to keep track of the current line, the current position within this line and the current line number.

You may decide to return a special token representing the end of the file or a boolean method (perhaps `hasNextToken`) indicating if the end of the file has been reached.

Provide a main program which reads an input file and displays each token on a separate line (including the value of the token (if appropriate) and its position within the input file:

```
$ cat > test.toy
x +=++y; // Comment
// Another comment
if (x != 10) return;

$ java ToyScanner test.toy
IDENTIFIER x @ line 1 col 1
PLUS_EQUAL @ line 1 col 3
PLUS_PLUS @ line 1 col 5
IDENTIFIER y @ line 1 col 6
SEMI_COLON @ line 1 col 7
IF @ line 2 col 1
LEFT_PAREN @ line 3 col 3
IDENTIFIER x @ line 3 col 4
BANG_EQUAL @ line 3 col 7
NUMERIC_LITERAL 10 @ line 3 col 10
RIGHT_PAREN @ line 3 col 12
RETURN @ line 3 col 14
SEMI_COLON @ line 3 col 20
END_OF_FILE @ line 4 col 0
```

Tokens

The tokens in our toy language are:

Identifier: `letter (underscore ? (letter | digit))*`

letter: `[a-zA-Z]`

digit: `[0-9]`

underscore: `[_]`

Numeric Literal: `DecimalLiteral | HexadecimalLiteral`

DecimalLiteral: `digit (underscore ? digit)*`

HexadecimalLiteral: `0 ('x' | 'X') HexDigit (underscore ? HexDigit)*`

HexDigit: `[0-9a-fA-F]`

Character Literal: `singleQuote char singleQuote`

String Literal: `DoubleQuote char* doubleQuote`

singleQuote: `['']`

doubleQuote: `["]`

char: `[^'"/] | \n | \r | \' | \" | \\ | \0`

note: `[^'"/]` does not include control characters

Reserved Words: The following names are not identifiers but are rather reserved words. Each should be its own kind of token.

```
program
if
else
while
return
int
char
boolean
void
```

The following names ARE identifiers but will be predefined by the compiler (the user may redefine these identifiers in nested scopes if so desired):

```
true
false
abs
in
out
```

Symbols: Each of the following symbols is a token - but have been placed in logical groupings:

Unary Operators: `-  +  !  ~  ++  --`

Binary Operators: `+  -  *  /  %  &  &&  |  ||  ^  <<  >>`

Relational Operators: `<  <=  >=  >  ==  !=`

Assignment Operators: `=  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=`

Bracketing: `(  )  [  ]  {  }`

Punctuation: `.  ,  ;  :`

Unused: `` `  @  #  $  ? `` Not tokens, so these will produce errors

Errors

You will need to develop a strategy for handling errors. This is the one case in which it is not best to return the longest sequence of characters starting at the current position which is not a token since that will likely consume valid tokens. Possibilities include

Generate an “error token” for each (invalid) character that cannot be the start of a token.

Skip to the beginning of the next line and include the rest of the current line in the “error token”.

Include characters in the “error token” until you reach the beginning of a valid token.

In order to produce more meaningful error messages, you may wish to consider how to handle certain special cases such as numeric literals which contain invalid characters (especially hexadecimal literals), unterminated character and string literals, or string and character literals that contain invalid characters (e.g., \z).

End of File

It may be more convenient to return an “end of file” token since there can be white space after the last token in the file. If you decide instead to export a method hasNextToken, it will need to be prepared to handle the case in which white space is present after the last token in the file.