For this part of the assignment you will design, implement and test a scanner for the D language, which is defined in a separate handout. The scanner, as well as the rest of your compiler, should be written in Java.

A scanner (or lexical analyzer) reads the character (text) representation of a program and transforms it into a stream of tokens representing the basic lexical items in the language. These tokens include punctuation (lparen, rparen, semicolon, ...), keywords (if, int, return, ...), integers, and identifiers. The scanner skips over whitespace and comments in the source code; these are ignored by the compiler and do not appear in the token stream that will eventually be read by the parser.

You should review the grammar definition and decide on all the tokens that you need to recognize. You might want to write down simple deterministic finite automata drawings for the portions of the scanner that will skip whitespace or recognize *id*'s and *int*'s so that you are sure you understand what you are trying to implement.

java class `Scanner`

The scanner should be defined as the Java class Scanner.

There is one constructor for the Scanner class that takes one argument, a CompilerIO object. The Scanner uses the methods of CompilerIO to read and write files.

The Scanner class should provide a method `public Token nextToken()` that the client program can call to obtain tokens sequentially. The `nextToken` method looks at the input characters one after another and essentially executes a DFA to isolate and recognize the significant tokens in the language. It returns a Token object of the appropriate type, depending on the input that it sees.

You may want to define various private methods to facilitate the operations of `nextToken`. In my implementation, I have private helper methods `skipWhitespace`, `getCurrentCharacter`, `acceptCurrentCharacter`, and `invalidCharacter`.

`private void skipWhitespace()` gets and accepts characters until it finds something that is not whitespace. This process advances the current character position. The advance may extend over several lines.

`private char getCurrentCharacter()` returns the current character to the caller. It returns a newline ('\n') if the current character position has been advanced to one past the last character in the most recent input line. `private void acceptCurrentCharacter` () actually advances the character position. If this advance would put the character position more than one past the last character, then a new source line is read in and the character position is reset to 0.

`private void invalidCharacter(char c)` takes a character that has been determined to be inappropriate and prints an error message using the CompilerIO method `printAsmLine.`

You don't need to use the same design, but you need to think about how you will accomplish the various functions.

Aside from implementing the DFA that recognizes the tokens, you need to understand what you are doing with the input strings from the source file. The CompilerIO method `readSrcLine` returns the source file one entire line at a time, but you are scanning it one character at a time. Be sure you think about the details of what happens at end-of-line and end-of-file. The methods `getCurrentCharacter` and `acceptCurrentCharacter` described above are one way to isolate the main scanning process from some of the details related to file reading.

You'll need a table of language keywords and their corresponding lexical tokens in order to distinguish between keywords and all other identifiers. You can use a java.util.HashMap for this; you don't have to re-implement a symbol table from scratch. Initialize the table with the keywords and their associated Token.xyz values in the Scanner constructor then use the table in nextToken to decide if an identifier is in fact a keyword. You need to wrap each Token.xyz value in an Integer object, since primitives like ints cannot be stored directly in a HashMap.

java class `Token`

You need to define a class Token to represent the lexical tokens. The Token class includes a field to store the lexical class of the specific Token object (id, int, lparen, not, ...). Class Token should include an appropriate symbolic constant name (static final ints) for each lexical class.

Tokens for identifiers and integers need to contain additional information: the String representation of the identifier or the numeric (int) value of the integer. Since there's little space overhead involved, it's reasonable to have int and String instance variables in the Token class and ignore these fields if the Token is something other than an integer or identifier.

In my implementation, there are three constructors for the Token class. One constructor takes just one integer argument, the type of the Token. This constructor is used to create all of the Tokens with no associated attributes. Another constructor takes an integer type and a String. This constructor is used to create identifier Tokens. The third constructor takes an integer type and an integer value. This constructor is used to create integer Tokens.

Your Token class must provide methods to access the fields of a Token object. Thus, you must implement `public int getType()`, `public String getLabel()`, and `public int getValue()`.

Class Token should contain an appropriate toString() method that returns a descriptive string representation of a Token depending on the type of the Token.

java class `ScanTest`

There is a simple test program included in the homework download. The test program uses the CompilerIO class from the previous assignment for line-oriented input and output operations that

read and write the input and output files. The test program uses your Scanner class to read a D source program and prints the resulting Tokens to the output file.

Source program lines are echoed to the output file as they are read, to make it easier to see the correlation between the source code and the tokens. Echoing of the input lines is handled automatically by CompilerIO.

java class `TokenTest`

There is a simple test program included in the homework download. The test program creates a few representative Token objects and prints them out using System.out.println and the toString method.

Implementation

While Java provides classes StreamTokenizer and StringTokenizer (and, in 1.4, Pattern and Matcher) to break input streams or Strings into tokens, for this assignment you must implement the scanner without them. Your scanner should examine the source program one character at a time and decode the input into individual tokens manually.

The Java library provides several functions that you may find useful. Class Character contains methods for classifying characters. Class Integer contains methods for parsing Strings to create int values and a constructor for wrapping primitive int values in an Integer object. The StringBuffer class is an efficient way to collect characters one by one into a String.

Be sure your program is well written, formatted neatly, contains appropriate comments, etc. Be careful to precisely specify the state of the scanner in comments describing variables, particularly exactly how far the scan has progressed on the current line, where the next unprocessed character is, and so forth. Use public and private to control access to information; be sure to hide implementation details that should not be visible outside the scanner.

Keep things simple. One could imagine a Token class hierarchy with an abstract class Token, a separate subclass for each kind of Token, each containing a separate toString method, and hierarchies of classes for different groups of operators (addition operators, multiplication operators, etc.). One could imagine it, but it's probably not a great idea. Tokens are simple data objects; even with lots of comments and the symbolic constants for each kind of Token, the entire class definition can easily fit on two or three pages.