For this part of the assignment you will design, implement and test a scanner for the fm 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, prolog, show, ...), integer and floating point literals, 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 ids, integers, and reals 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 two arguments, a CompilerIO object and a SymbolTable object. The Scanner uses the methods of CompilerIO to read and write files, and the methods of the SymbolTable object to keep track of information about the symbols it should know about (ie, the fm language keywords).

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. Use the provided SymbolTable class for this; you don't have to re-implement a symbol table from scratch. Initialize the table with the keywords and their associated Symbol values in the Scanner constructor then use the table in nextToken to decide if an identifier is in fact a keyword. The objects stored in the SymbolTable are Symbols, keyed with the actual identifier string. So, for example, to store the keyword "if", your constructor might do something like this:

```
keywords.putSymbol(0,"if",new Symbol("if",Token.KW_IF));
```

This assumes that there is an instance variable "keywords" that has a reference to the SymbolTable object provided to the constructor.

java class `Token`

You are provided with a Token class. The Token class includes a field to store the lexical class of the specific Token object (id, integer, lparen, ...). Class Token includes an appropriate symbolic constant name (static final int) for each lexical class. (In fact, it also includes a few unused extras because I have tweaked the language a few times.)

Tokens for identifiers and numeric literals contain additional information: the String representation of the identifier or the numeric value of the integer or real.

Objects of class Token are returned from the Scanner in response to calls to its nextToken method. These objects form the interface between the Scanner and the Parser.

java class `Symbol`

You are provided with a Symbol class. The Symbol class includes a label and a type, plus a hash map to store a list of attributes if needed. In the Scanner, you will create a simple Symbol object for each keyword with its associated Token type and store it in the symbol table, as shown above. When your Scanner recognizes an identifier, it will look it up in the symbol table. If the identifier

is defined, then it is a keyword, and it will return the proper token. If it is not defined, then the Scanner will return a Token of type Token.ID.

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 an fm 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.

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. Classes Integer and Double contain methods for parsing Strings to create numeric values. 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.

Testing

You should use the separate test driver class named ScanTest to exercise your Scanner class and show that it works correctly.

Turn in

You should turn in Scanner.java. The link for this turnin will be on the class calendar page.