fm language parser

For this assignment, you will write a recursive descent parser that analyzes programs written in the fm language. The parser validates that the program is structured correctly according to the language grammar and it generates symbol table entries. The parser does not yet generate Postscript output; that happens in the next assignment.

The parser reads the tokenized representation of a program (using the scanner from the previous assignment) and validates it against the grammar for the language generated by the associated grammar. The parser also generates symbol table entries for each symbol defined in the program prolog.

There is a definition of the language grammar provided as part of this homework assignment. It may have been updated from the definition given in the previous homework, so be sure that you use the latest revision of the FM Language Specification.

java class `Parser`

(skeleton provided) The parser should be defined as the Java class Parser. There is one public constructor, and one public method. There are *numerous* private methods. We have supplied implementations for many of the methods described below in the skeleton Parser.java file.

`public Parser(CompilerIO io, SymbolTable t, Scanner s)`

(provided) There is one constructor for the Parser class that takes three arguments. The Parser uses the methods of CompilerIO to write information to the output file. The SymbolTable object is used to store information about program variables. There are two tables created in the SymbolTable object before the Parser constructor is called: table 0 is the reserved keywords table used by the Scanner, table 1 is used by the Parser for program symbols. The Scanner object has the nextToken method that the Parser uses to get the stream of Tokens. The constructor stores references to these three objects in private instance variables.

Any other initialization that your Parser needs should also be done in the constructor.

`public boolean parse()`

(provided) The Parser class provides one public method `public boolean parse()` that the main program can call to request that a parse be run on the CompilerIO input file. The `parse` method calls the private method `parseProgram` to start the parse going, and then returns a boolean result: true if the file was parsed without error, false if there were any errors. Note that *program* is the start symbol for our grammar, and so the initial method is named `parseProgram`. This is the general naming convention throughout this parser; the name of the parsing method is the word "parse", followed by the name of the non-terminal that it is designed for.

private parsing methods

(provided) The top level parse method is `parseProgram`. It is provided to you as a template for all the subordinate parse methods. Note that the calls to `matchToken` and `parseMovieBody` are copied directly from the grammar productions.

(implement) The rest of the Parser class is a set of private methods, one to parse each grammar non-terminal (*movieBody, pageBlock, expr,* etc.) plus whatever helper methods you need in order to simplify the parse methods. The javadoc from my implementation is provided in the doc directory of the download.

The parse methods should follow the grammar very closely so that you can keep track of what is happening and understand what has gone wrong while you are debugging. You have been provided with rewritten productions that eliminate left recursion.

You should write javadoc comments for each parse method to identify clearly which productions are implemented by the method. This will help you verify that all of the productions have been implemented correctly.

private helper methods

(provided) You may want to define various private methods to facilitate the operations of the parse methods. This class grows quite quickly because of all the methods needed for the various non-terminals in the language, so you want to avoid cut-and-paste coding as much as possible. Isolate common functions in separate methods and then call them when you need them from the parse methods.

In particular, you need a method that compares the current Token to the expected Token type and generates an error if it is not correct, or accepts it and uses nextToken to advance if it is the expected type. In my implementation I have two methods that do this:
`void matchToken(int type)` and `void matchTokenArray(int[] type)`. Each of these methods throws a SyntaxException if there is a mismatch, and the calling parse method can catch and report the exception.

If you use exceptions to report errors, it is helpful to have a method that can be called to process the exception. In my implementation I have a method
`void processSyntaxException(SyntaxException e)` that is called in the catch block of any parse method that tries to match tokens.

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

administrative functions

In addition to the basic parse function described above, your parser should provide some optional output features.

Each parser procedure should, if requested, be able to print a message each time it is entered and right before it exits. This trace will help you visualize how the parser works, and can be a useful debugging tool. Use the CompilerIO method emitWithPrefix to print the trace messages so that when the trace is being printed, the trace messages appear along with the source program in the parser output. Don't worry if the trace output doesn't appear to be exactly synchronized with the echoed input lines. Some of the trace output corresponding to one input line may not appear until after the next line of the source program has been read and printed. That's perfectly normal - a parser often doesn't realize it's done parsing a construct until the beginning of the next construct has been read.

(provided) Implement the method `public void setShowMethods(boolean b)` so that the main program can turn method tracing on and off.

(provided) Implement private methods `traceEntry` and `traceExit.`

(implement) Call the entry and exit methods at the start and end of each parse method to optionally provide the method trace output.

symbol entries

(implement) Once you have the parser running well enough to accept valid programs, you can add some code to actually do something with the information being recognized. In our case, we will generate Symbol objects for each symbol declared in the prolog. You should add the Symbols to the symbol table, and print out an informative message in the output file when each Symbol is defined if requested.

In method `parseVariableDeclaration` add a call to create a new Symbol object and add it to table 1 of the SymbolTable object, and call traceSymbol to optionally print it out using the toString method of class Symbol. For example.

```
Symbol var = new Symbol(prevToken.getLabel(),Symbol.VARIABLE);
symbolTable.putSymbol(SC_VARIABLE,var.getLabel(),var);
traceSymbol(var);
```

Notice that the type of the Symbol is Symbol.VARIABLE. This is different from the way Symbols are used in the Scanner. All of the Symbols defined in the Parser are of type Symbol.VARIABLE.

(provided) Implement the method public void setShowSymbols(boolean b) so that the main program can turn symbol tracing on and off.

java class `SyntaxException`

The class SyntaxException is provided to you for use in defining and reporting errors if you like. You do not have to use this class and you do not have to use exceptions at all if you don't want to.

java classes `Symbol and Token`

The classes Symbol and Token are provided to you for use in defining and printing the symbols in the source language program as your parser recognizes them.

java class `ParserTest`

There is a simple test program included in the homework download. The test program uses the CompilerIO class from the previous assignments for line-oriented input and output operations that read and write the input and output files. The test program uses your Parser class to read a source program and print the resulting output to the output file.

Source program lines are echoed to the output file as they are read. Any messages generated by your Parser should also go to the output file, which will happen if you use emitWithPrefix to write out the messages. You can control the output of your Parser using ParserTest command line switches.

java classes `Scanner` and `CompilerIO`

The utility classes from the previous assignments are provided to you as binary class files. You can use these classes or your own implementations as you like, but remember that we will use these classes when doing the grading so your Parser must work correctly using the provided classes.

Implementation notes

- You are not required to do extensive error processing or recovery. However, you should print an error message if the parser encounters a syntax error while parsing, and the `parse` method should return false to the original caller. In other words, your program should not accept invalid programs.
- Recursion in the grammar is often used to define sequences of various things like the list of function definitions that make up a program, the *factor*s to be multiplied together to calculate the value of a *term*, etc. You can often use a simple loop to handle these sequences.
- Define simple helper methods to avoid redundant code in the parser. If you find yourself using cut-n-paste to copy chunks of code repeatedly, that is likely to be a sign that you should abstract those operations into a separate method.
- Print the source code lines, trace output, and symbol table entries as you go; don't attempt to build a huge string containing multiple lines of output or otherwise buffer the data in your code.
- While production quality error recovery is not required, it is good to have a simple strategy for handling syntax errors. A key principle is that no matter what sort of junk is found in the input, the parser should continue to make progress through the source file. Your error handling strategy shouldn't leave the parser stuck somewhere in the input, repeatedly examining the same token without advancing.
- Start small. Get a few pieces of the parser for a small part of the grammar working first. Check it out with simple test files. Then add to this until you have a parser for the complete language.

Does it work?

We will run ParseTest using your Parser and compare the output with the reference implementation's output. Symbol table output will be enabled, but method tracing will not. Sample output files are included in the bin directory of the download file.

Your Parser must recognize and correctly parse valid programs, and must generate the correct symbol table entries. However, it is not required that you use the same method names or exact calling structure that we used in our implementations, and so your method traces may look considerably different from ours.

Turn in

Write a brief summary of your experience on this project and a description of your Parser class.

1. Name and UWNetID
2. Which parts of your Parser work well? Are you able to parse all the sample fm files?
3. Which parts do not work?
4. Was there a particular issue or concept that caused problems in getting your Parser to work correctly?
5. Any other comments with respect to this homework?

Turn in the summary file and your Parser.java file using the Catalyst turn-in link on the web site.