DUE: Thursday December 2, 2004, 11:59 p.m.
fm language parser

Overall goal: For this assignment, you will write a recursive descent parser that analyzes programs written in the fm language. The parser:
1. validates that the program is structured correctly according to the language grammar. The output of this is a Boolean, true or false.
2. generates symbol table entries. The output of this is that the Symbol table has an entry for every variable declared in the prolog, specifying its name.

Details: The parser reads the tokenized representation of a program (generated by the scanner) 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 – fm-specv2.pdf. Note that this grammar has been re-factored to eliminate left recursion in order to simplify parsing.

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 already created 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, and initialize the `firstSets` HashMap that is described later.

Any other initialization that your Parser needs should be added to the provided 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 (though Parser class accesses this input by calling `Scanner.nextToken()`. 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.

<u>private parsing methods</u>
(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 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. Be sure to use the new grammar definition provided with this homework – this will make parsing easier.

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.

<u>private helper methods</u>
(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 should probably have 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.

To test to see if a given token could be the first token for in a string generated by a particular non-terminal, use `isFirst(Token t,String nonterm)`. This method relies upon the firstSets variable that was initialized in the constructor. This is useful when a given non-terminal has more than one production and you need to decide which one to use based on looking at the next token. Note that isFirst() is also useful for productions that can derive epsilon (the empty string). For instance, prologTail can derive either *prologStatement prologTail* or *epsilon.* To decide which one, looking at the grammar, we find that the only prologStatement is a variableDeclaration, and a variableDeclaration starts with a Token.ID. So what you want to check for is "is the next Token a Token.ID?". One way to do this is to check the type (not the label) of theToken explicitly, the other is to use the isFirst method as follows:

```
if (isFirst(theToken,"prologStatement"))
```
If not then we are matching epsilon, and thus finished with this prologTail.

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

<u>administrative functions</u>
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.

<u>symbol entries</u>
(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 class `Symbol`

The class Symbol is 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, Token, and CompilerIO`

The utility classes from the previous assignments are provided to you. 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. Your error handling strategy shouldn't leave the parser stuck somewhere in the input, repeatedly examining the same token without advancing – it should terminate given any possible input.
- 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.
- 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.

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.

We have provided test files (tA.fm, tB.fm, tC.fm, hello.fm, TetheredBox.fm), and output files (tA.txt, .etc.) – the output files show the results with both symbol tracing and method tracing on. Use the "-symbols" command line option to turn on symbol tracing. Method tracing is on by default.

What to turn in?

Turn in just your Parser.java file. If you create any other new Java files, turn those in too, but don't turn in the provided files ScanTest.java, Token.java, etc.

(optional) You may turn in test files – like test1.fm, test2.fm – that demonstrate that your compiler works. This is most helpful if your compiler does not completely work on the provided test files but does work on simpler cases.

Sample output

The following output was generated with ParseTest by parsing the sample file Hello.fm with method and symbol tracing enabled. This is the resulting file hello.txt. Other sample output files are included in the fm directory of the download file.

```
% // Define a simple flip movie for       % Enter: parsePageStatement
scanning and parsing practice             % Enter: parseCallEnd
% movie ta {                              % Enter: parseExprList
% Enter: parseProgram                     % Enter: parseExpr
%     prolog {                            % Enter: parseTerm
% Enter: parseMovieBody                   % Enter: parseFactor
% Enter: parsePrologBlock                 % Exit:  parseFactor
%          outline : Box(540,720);        % Enter: parseTermTail
% Enter: parsePrologStatements            % Exit:  parseTermTail
% Enter: parsePrologStatement             % Exit:  parseTerm
% Enter: parseVariableDeclaration         % Enter: parseExprTail
% Symbol: outline, type: 3                % Exit:  parseExprTail
% Enter: parseExprList                    % Exit:  parseExpr
% Enter: parseExpr                        % Enter: parseExprListTail
% Enter: parseTerm                        % Enter: parseExpr
% Enter: parseFactor                      % Enter: parseTerm
% Exit:  parseFactor                      % Enter: parseFactor
% Enter: parseTermTail                    % Exit:  parseFactor
% Exit:  parseTermTail                    % Enter: parseTermTail
% Exit:  parseTerm                        % Exit:  parseTermTail
% Enter: parseExprTail                    % Exit:  parseTerm
% Exit:  parseExprTail                    % Enter: parseExprTail
% Exit:  parseExpr                        % Exit:  parseExprTail
% Enter: parseExprListTail                % Exit:  parseExpr
% Enter: parseExpr                        % Enter: parseExprListTail
% Enter: parseTerm                        % Exit:  parseExprListTail
% Enter: parseFactor                      % Exit:  parseExprListTail
% Exit:  parseFactor                      % Exit:  parseExprList
% Enter: parseTermTail                    % Exit:  parseCallEnd
% Exit:  parseTermTail                    %     }
% Exit:  parseTerm                        % Exit:  parsePageStatement
% Enter: parseExprTail                    % Enter: parsePageTail
% Exit:  parseExprTail                    % Exit:  parsePageTail
% Exit:  parseExpr                        % Exit:  parsePageStatements
% Enter: parseExprListTail                % }
% Exit:  parseExprListTail                % Exit:  parsePageBlock
% Exit:  parseExprListTail                % Enter: parsePageBlocksTail
% Exit:  parseExprList                    % Exit:  parsePageBlocksTail
%     }                                   % Exit:  parsePageBlocks
% Exit:  parseVariableDeclaration         % Exit:  parseMovieBody
% Exit:  parsePrologStatement             %
% Enter: parsePrologTail                  % Exit:  parseProgram
% Exit:  parsePrologTail
% Exit:  parsePrologStatements
%     show (2) {
% Exit:  parsePrologBlock
% Enter: parsePageBlocks
% Enter: parsePageBlock
%          outline.draw(36,36);
% Enter: parsePageStatements
```