D language parser

For this assignment, you will write a recursive descent parser that analyzes programs written in the D language. The parser validates that the program is structured correctly according to the D language grammar and it generates symbol table entries.

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. Our parser will generate symbol table entries on the fly for each symbol defined in the program.

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

java class `Parser`

The parser should be defined as the Java class Parser.

public constructor and method

There is one constructor for the Parser class that takes one argument, a CompilerIO object. The Parser uses the methods of CompilerIO to write information to the output file, and so a reference to the CompilerIO object should be saved for use by other methods in the class. The constructor should also create a new Scanner object and save a reference to it for later use. Finally, any other initialization that your Parser needs should be done in the constructor.

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 D grammar, and so the initial method is named `parseProgram`.

private parsing methods

The rest of the Parser class is a set of private methods, one to parse each major D construct (*expr*, *statement*, *functionDefinition*, *parameters*, 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. You have been provided with rewritten productions that eliminate left recursion. You may want to rewrite some of these productions further in order to replace the tail recursion with iteration; however, this is not required and I did not do it in my implementation for most of the productions.

You should write javadoc comments for each parse method that 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

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.

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.

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 printAsmLine 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 D 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.

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

Implement private methods `traceEntry` and `traceExit` and call them at the start and end of each parse method to optionally provide the method trace output.

symbol entries

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 *functionDefinition*, *parameter*, and *declaration*. In a full-up compiler these Symbols would be added to a symbol table, but for our purposes we will just print out informative messages in the output file when each Symbol is defined.

In the appropriate place in each parser method that recognizes a symbol definition, add a call to create a new Symbol object, and then print it out using the toString method of class Symbol. Once you have printed the message, you do not need to store the reference for later use. If you were building a symbol table for the program, this is where you would add the Symbol reference to the symbol table.

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 D language program as your parser recognizes them. There are two simple constructors, get/set methods for all fields, and a toString method.

java class `ParserTest`

There is a simple test program included in the homework download. The test program uses the CompilerIO class from a previous assignment 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 D 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 printAsmLine 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 as binary class files. You can use these skeleton 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 skeleton 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.
- It is a good idea to introduce separate parser methods for things that are similar in the grammar, but mean different things in a D program. An example is the rules for *parameters* and *declarations*. While these are syntactically almost the same, the processing needed to handle them in the final compiler is likely to be different. The compiler structure will be cleaner if you define different methods for these non-terminals.
- 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 tables 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 error checking 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. Be sure your error handling strategy doesn'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. 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.

Sample output

The following output was generated with ParseTest by parsing the sample file hello.d with both method tracing and symbol tracing enabled. This is the resulting file hello.asm.

```
; // D test program hello.d  CSE413 6/99, 2/02
;
; // A minimal complete program
;
; int main() {
Enter: parseProgram
Enter: parseFunctionDefinition
GLOBAL FUNCTION main
;   return 0;
Enter: parseStatements
Enter: parseStatement
Enter: parseExpr
Enter: parseTerm
Enter: parseFactor
Exit:  parseFactor
Enter: parseTermTail
Exit:  parseTermTail
Exit:  parseTerm
Enter: parseExprTail
Exit:  parseExprTail
Exit:  parseExpr
; }
Exit:  parseStatement
Enter: parseStatementsTail
Exit:  parseStatementsTail
Exit:  parseStatements
Exit:  parseFunctionDefinition
Enter: parseProgramTail
Exit:  parseProgramTail
Exit:  parseProgram
```