Compiler Project - Part 1 Lexical Analyzer

Compilers, Fall 2016

In this part, you will write a lexical analyzer for Decaf, a small subset of the Java programming language. The programming language is case sensitive (i.e. does make a distinction between upper case and lower case letters in identifers). Refer to Appendix A for a short description of the Decaf language.
We will be using Java and JFlex (version 1.6.1) to implement the compiler. Refer to the documentation for the JFlex tool for all details regarding JFlex. Some support code (Java enums to use and code template files) for the lexical analyzer can be found under Other material on the course web page.

Step 1

Use JFlex (available from www.jflex.de) to write regular expressions that define sets of strings corresponding to tokens. The header of the Flex source file should look like this:

Listing 1: lexical.flex

```
// ... put any package or import statements that you wish the generated class
//      to have, here...

%%
%class Lexer  // The class name of the generated Lexical Analyzer class
%unicode
%line
%column
%type Token  // The class name of the class returned by the Lexical
             // Analyzer class when yylex() is called
// %debug    // Uncomment to trouble shoot your definitions

%{
  // ... put any member functions that you wish the generated
  //     Lexical Analyzer class to have here. You can then refer
  //     to these member functions from your translation rule
  //     codeblocks.
%}

%eofval{
  // ... return a Token for EOF (End of file) here
%eofval}

Letter           = [a-z] | [A-Z]
Digit            = [0-9]
// ... The rest of your definitions for patterns of tokens go here

%%

// ... put your translation rules here ...
```

```
// {SOME_PREV_DEFINED_MACRO}    { /* The code block to execute
//                                   on seeing this macro */
//                              }
```

The code blocks of your translation rules should return an instance of the Token class (see step 3). The returned instance of the Token class should have all the needed information about the token. Use the provided enum classes, and your SymbolTableEntry class (see step 2) for this needed informations. This should include:

- The TokenCode enum of the token.
- The SymbolTableEntry (see step 2) for the lexeme of the token (if required, otherwise null).
- The OpType enum of the token, if the token stands for an operator from a set of different operators.
- The DataType enum of the token.

The lexical analyser should not return a Token at all for any whitespace or comments. In these cases the code block for the translation rule should be empty. Whitespace is spaces, tabs and newlines. Comments start with /* and end with */ and can span more than a single line[1].

The token for identifiers is defined in the following manner:

$$\begin{aligned} \textbf{letter\_} &\rightarrow [\text{A-Za-z}] \mid \_ \\ \textbf{digit} &\rightarrow [0-9] \\ \textbf{id} &\rightarrow \textbf{letter\_} \; (\; \textbf{letter\_} \mid \textbf{digit} \;)^* \end{aligned}$$

The token returned for identifiers should have a TokenCode of IDENTIFIER and include a symbol table entry for that identifier. If the length of the identifier is greater than 32 then it is considered too long. In this case the TokenCode of the returned token should be ERR_LONG_ID (there is no reason to add long identifiers to the symbol table).

Numbers are defined in the following way:

$$\begin{aligned} \textbf{digits} &\rightarrow \textbf{digit}+ \\ \textbf{optional\_fraction} &\rightarrow (. \; \textbf{digits})? \\ \textbf{optional\_exponent} &\rightarrow \textbf{(E (}+ \mid -\textbf{)? digits)?} \\ \textbf{num} &\rightarrow \textbf{digits optional\_fraction optional\_exponent} \end{aligned}$$

Note that for both integers and real numbers, a token with a TokenCode enum of NUMBER should be returned. The token should also include a DataType enum containing the appropriate data type.

All keywords are reserved and a special token should be returned for each of them.

The following are the groups of operators used and their possible lexemes:

- **incdecop** : '++' '--'
- **relop**: '==' '!=' '<' '>' '<=' '>='
- **addop**: '+' '-' '||'

[1] If you want to you can also implement single line comments, that start with // and end with a new line, or the end of the file.

Step 1 continued on next page...

- **mulop**: '*' '/' '%' '&&'

At the end of the file a token with the TokenCode EOF should be returned. This can be done in the .flex file by placing code returning such a Token in a %eofval{ ... %eofval} section. Refer to the JFlex documentation for more details on %eofval.

If an unrecognised character is encountered (for example $, # ...) the Lexical Analyser should return a token with TokenCode ERR_ILL_CHAR.

Step 2 - Symbol table and SymbolTableEntry class

Write a class implementing a symbol table that will hold SymbolTableEntry objects for all identifiers and numbers encountered. The symbol table implementation should support:

- Looking up an identifier/number to see if it has already been encountered. If it has, then return the SymbolTableEntry for it.
- Creating a SymbolTableEntry and add it to the list of entries.
- Accessing all SymbolTableEntry objects in the symbol table in the order they were encountered.

Also, write a class for a SymbolTableEntry that should hold the lexeme of the identifier/number (in later phases of the project we will add some more attributes to the SymbolTableEntry class).

Step 3 - Token class

Write a Token class that will be the class of objects returned by the generated Lexical Analyser. The token class should have accessible properties for TokenCode, DataType, OpType and SymbolTableEntry. The instances of the Token class will be returned to your Main class (which in the next phase will be replaced by a Parser implemented by you).

Step 4 - Main class

Write a class named TokenDumper, containing a runnable main method that reports all tokens in the file whoose filename it is provided with, together with any relevant data for the tokens. Finally it should print out the contents of the symbol table in the order that identifiers/numbers are encountered in the source program.

To help you out here is a rough draft of the code:

Listing 2: TokenDumper.java

```
import java.io.*;

public class TokenDumper {
  public static void main(String [] args) throws IOException {
    Lexer lexer = new Lexer(new FileReader(args[0]));

    while(true) {
      Token t = lexer.yylex();
      System.out.print(t.getTokenCode().toString());
      // TODO: Print out relevant data for token in parenthesis
```

```
      if (/* At the end of file */)
        break;
    }
    // TODO: Print out the symbol table
  }
```

Note that in line 5, the Lexer class is the class generated by JFlex (the name is specified by the %class option). Also note that in line 8, the method yylex() is the method of the generated class that returns the next token[2].

Program Output

Your main program should print the token code of each token, followed by the relevant other information about that token. At the end the program should print out the contents of the symbol table in the order that the identifiers / numbers were encountered in the source program. To test the lexical analyser write Decaf programs and test your program on them, one such test program should look like this and generate the following output:

Listing 3: DecafTest.decaf

```
class Program {
  /* This is a simple test program */
  int i, j;
  real number;

  static void main() {
    i = 0;
    j = 1;
    if (i < j) {
      number = 12.34;
    }
    else {
      number = 12.34E7;
    }
    i = i + j;
  }
}
```

Listing 4: Program output on DecafTest.decaf

```
CLASS IDENTIFIER(Program) LBRACE INT IDENTIFIER(i) COMMA IDENTIFIER(j) SEMICOLON
REAL IDENTIFIER(number) SEMICOLON STATIC VOID IDENTIFIER(main) LPAREN RPAREN LBRACE
IDENTIFIER(i) ASSIGNOP NUMBER(0) SEMICOLON IDENTIFIER(j) ASSIGNOP NUMBER(1)
SEMICOLON IF LPAREN IDENTIFIER(i) RELOP(LT) IDENTIFIER(j) RPAREN LBRACE
IDENTIFIER(number) ASSIGNOP NUMBER(12.34) SEMICOLON RBRACE ELSE LBRACE
IDENTIFIER(number) ASSIGNOP NUMBER(12.34E7) SEMICOLON RBRACE IDENTIFIER(i)
ASSIGNOP IDENTIFIER(i) ADDOP(PLUS) IDENTIFIER(j) SEMICOLON RBRACE RBRACE EOF

0  Program
```

[2]The actual name of the the method, yylex(), can be changed by giving the appropriate option in the .flex file. Refer to the documentation for JFlex if you are interested in changing this.

Program Output continued on next page...

```
1  i
2  j
3  number
4  main
5  0
6  1
7  12.34
8  12.34E7
```

Obviously, you will also need to write more Decaf programs to test your Lexical Analyser!

Hand-in Instructions

- Due date is Monday September 12th
- You can work in a group of two or three students if you want.
- Hand in the following electronically through MySchool in a **.zip** archive named **SourceCode.Zip**
  - All source code
  - A single Decaf test file called test.decaf that uses all possible tokens.
  - A single .jar file named LexAnal.jar that can be run using the command:
    `java -jar LexAnal.jar filename.decaf`
  - Keep in mind that your code will be tested by running the above command on a variety of decaf programs. It is in your best interest that this command works flawlessly when the archive is unzipped and the above command runned on the extracted jar file.

Appendix A: Decaf

Decaf is a language that is a very simplified subset of Java. It is defined in full by the BNF below. Here are some notes on the differences between Java and Decaf:

- Decaf is not an object oriented language. Programs in Decaf are written in a single class, that should by convention be named 'Program'. The class is only used as a skeleton around the actual program code. There are no accessibility modifiers (no 'public', 'private' or 'protected') and the keywords 'abstract', 'const' are not used. There is also expected to be a function named 'main' taking no parameters. This is the main function of the program that will be executed.
- There are two scopes of variables in Decaf, global scope and method scope. Global variables are declared first, inside the 'class' declaration where member variables are declared in Java. These declarations must precede any method declarations. Local variables (method scope) must be declared first in the method declaration before any statements.
- The only types that can be directly used in Decaf are int and real. There is also an implicit boolean type used in 'if' and 'for' statements for conditional checking. In these places, an integer or real value of zero is interpreted as FALSE, anything else is TRUE. Boolean variable declarations are not supported.
- Functions can have a return type of 'void' which is a separate type. Variable declarations of type 'void' are not supported.
- Since object are not supported, all functions must be declared static and therefore start with the 'static' keyword.
- The for loop has the following constraints on the three parts it takes as argument:
  - The first part must be a variable assignment
  - The second part is interpreted as a boolean condition, that when false at the beginning of the block, breaks out of the loop.
  - The third part must either be an increment or decrement statement.
- Arrays of int or real can be declared either globally or locally. However, they can not be used as parameters to functions.

The BNF in Appendix B defines Decaf in more detail.

Appendix B: BNF for Decaf

The following is a context free grammar for Decaf in BNF form (tokens are in **bold**, nonterminals in *italics*). Note that the terminals **id, num, incdecop, relop, addop, mulop** have already been described above.

program ::= **class** **id** **{** *variable_declarations* *method_declarations* **}**

variable_declarations ::= *variable_declarations* *type* *variable_list* ; | ϵ

type ::= **int** | **real**

variable_list ::= *variable* | *variable_list* , *variable*

variable ::= **id** | **id** [**num**]

method_declarations ::= *method_declaration* *more_method_declarations*

more_method_declarations ::= *more_method_declarations* *method_declaration* | ϵ

method_declaration ::= **static** *method_return_type* **id** (*parameters*)
{ *variable_declarations* *statement_list* **}**

method_return_type ::= *type* | **void**

parameters ::= *parameter_list* | ϵ

parameter_list ::= *type* **id** | *parameter_list* , *type* **id**

statement_list ::= *statement_list* *statement* | ϵ

statement ::= *variable_loc* = *expression* ;
| **id** (*expression_list*) ;
| **if** (*expression*) *statement_block* *optional_else*
| **for** (*variable_loc* = *expression* ; *expression* ; *incr_decr_var*) *statement_block*
| **return** *optional_expression* ;
| **break** ;
| **continue** ;
| *incr_decr_var* ;
| *statement_block*

optional_expression ::= *expression* | ϵ

statement_block ::= { *statement_list* }

incr_decr_var ::= *variable_loc* **incdecop**

optional_else ::= **else** *statement_block* | ϵ

expression_list ::= *expression* *more_expressions* | ϵ

more_expressions ::= , *expression* *more_expressions* | ϵ

expression ::= *simple_expression* | *simple_expression* **relop** *simple_expression*

simple_expression ::= *term* | *sign* *term* | *simple_expression* **addop** *term*

term ::= *factor* | *term* **mulop** *factor*

factor ::= *variable_loc* | **id** (*expression_list*) | **num** | (*expression*) | ! *factor*

variable_loc ::= **id** | **id** [*expression*]

sign ::= + | −