COSC 4785
Compiler Construction
Programming Project 2, Decaf with Flex

1 What to do

You will write a complete lexical analyzer (scanner) for the complete Decaf language. You have already started on this in program 1 and now you just need to expand that work. That means you should not have any huge problems with this. Note here are you need to PERMANENTLY get rid of the code that processes "floats" because we will **not** be needing that.

The scanner is to be written using C++. You will continue to have to use Linux. Just accept that this will be the way for the rest of the course. Another point: follow the instructions and read the text and the Flex manual. For instance if the instructions say that you only print out a specific set of things, that is EXACTLY what is meant. Not additional things, no prompts, nothing but exactly what is asked for. If there is any question at all about output, input, file names, anything, ask before you make some arbitrary decision.

For this assignment there are four particular things to consider.

- When we get to the assignments that use **bison** your scanner will have to return integers that represent what lexeme (token) was encountered. Additionally, the information on line numbers, column numbers, and the "value" of certain tokens (*id* and *number*) will have to be made available to the code that calls yylex(). That means that in this assignment you will have to make those values available outside of yylex() and return integers that YOU know represent the lexemes. If you printed information from yylex() for the first assignment, STOP. The only place output will occur is in **main()**.

- This language supports both C style comments and C++ style comments. That is a comment can start with the two characters "/*" and contains everything until the first occurrence of "*/". Or a comment can begin with "//" and ends at the next newline character. These must be correctly recognized and IGNORED in the scanner. That means they do NOT break your program, nothing is printed to the output to indicate that they were encountered, and you handle situations like

  int /* a new variable */ newvar;

  which should result in a keyword (int), an identifier, and a semicolon being being the values returned from yylex() as well as the beginning column for "newvar" being correct.

  In the grammar document "//" is listed as a token. This is true BUT other than being used in the *flex* input to identify the start of a C++/Java style comment it is

NOT used for anything and most especially there is NO value returned by *yylex()* to indicate that this token was encountered.

- Unary minus versus binary minus. Do not worry about differentiating for this particular assignment. Later, when you are constructing the parser, you will more easily be able to determine which is which. Means nothing really in the lexical analyzer.

- You should indicate three different type of errors:

  1. Single character errors. Any unrecognized single character followed by an operator or whitespace. Issue an error message with the character as the cause (value) of the error.
  2. Word errors. If an unrecognized character is followed by some number of letters, digits, '_', or unrecognized characters, scan to the first whitespace or non-identifier (not error) character and consider that entire "word" the error. Issue an error message giving the text of the "word" that is in error as the value.

     Example: if the input contains say "A_4TheCoun@t+Not" then the string "A_4TheCoun" is an identifier, "@t" is a word error, "+" is an operator and "Not" is an identifier.
  3. Quit error. More than 20 errors, quit on the 21st. Count these in main() NOT in *flex*.

2 Output

This will be similar to the first assignment. The output will be formatted like:

Line	Column	Token	Value
xxxx	xxxx	xxxx	xxxx

- If a token, such as a keyword or special character like right parenthesis, has no "value" then the value column should be empty. The 'tokens' should be simple, short, descriptive strings like THIS, IF, RPAREN, DOT, COMMA, and so on all in upper case.

- Comments and whitespace (spaces, tabs, newlines) should not be printed out as tokens. They will simply be ignored by the lexer although they must be properly counted for column position.

- In the case of errors, there will be three possible tokens:

  – ER_CH and the character is the value,

  - ER_WD and the word is the value, and
  - ERRORS has no value but is given on the 21st error. And the program will then exit with no further output.

- The column is the where the token begins. Columns are numbered 1 to n starting at the left. This **will not** exceed 80 columns. Use reasonable editor settings when creating input files.

- Lines are numbered 1 to m starting at the "top."

3 Decaf language syntax

This is a simplified language that will be used for the Compiler Construction I course. Is is intended to be the focus of the programming assignments from this point on.

This grammar is based on Java so it should be familiar to all of you. The original version of this was called Decaf (for obvious reasons) so we will stick with that. Although maybe Sanka would be better (but that is a trademark). I borrowed this grammar from a professor of mine, Dr. Brad Vander Zanden.

This is the basic grammar. You may, in fact undoubtedly will, have to modify it to fit your conceptions and to flat make it work. **But** whatever changes you might make in the future, you will NOT change it this time.

3.1 Grammar

In this grammar, the use of "*" indicates 0 or more occurences of the previous regular expression and "+" means 1 or more occurences. Expressions enclosed in "<>" are grouped elements. This is done to prevent confusion that could result if parenthses were used..

1.	Program	→	ClassDeclaration$^+$
2.	ClassDeclaration	→	**class identifier** ClassBody
3.	ClassBody	→	{ VarDeclaration* ConstructorDeclaration* MethodDeclaration* }
4.	VarDeclaration	→	Type **identifier** ;
5.	Type	→	SimpleType | Type [ ]
6.	SimpleType	→	**int** | **identifier**
7.	ConstructorDeclaration	→	ϵ | **identifier** (ParameterList) Block
8.	MethodDeclaration	→	ResultType **identifier** (ParameterList) Block
9.	ResultType	→	Type | **void**
10.	ParameterList	→	ϵ | Parameter <, Parameter >*
11.	Parameter	→	Type **identifier**
12.	Block	→	{ LocalVarDeclaration* Statement* }
13.	LocalVarDeclaration	→	Type **identifier** ;
14.	Statement	→	;
		|	Name = Expression ;
		|	Name (Arglist) ;
		|	**print** (Arglist) ;
		|	ConditionalStatement
		|	**while** (Expression) Statement
		|	**return** OptionalExpression ;
		|	Block
15.	Name	→	**this**
		|	**identifier**
		|	Name . **identifier**
		|	Name [Expression]

16.	Arglist	→	ϵ | Expression < , Expression >*
17.	ConditionalStatement	→	**if** (Expression) Statement
		|	**if** (Expression) Statement **else** Statement
18.	OptionalExpression	→	ϵ | Expression
19.	Expression	→	Name
		|	**number**
		|	**null**
		|	Name (ArgList)
		|	**read** ( )
		|	NewExpression
		|	UnaryOp Expression
		|	Expression RelationOp Expression
		|	Expression SumOp Expression
		|	Expression ProductOp Expression
		|	(Expression)
20.	NewExpression	→	**new** SimpleType (Arglist)
		|	**new** SimpleType < [Expression] >* < [ ] >*
21.	UnaryOp	→	+
		|	-
		|	!
22.	RelationOp	→	==
		|	!=
		|	<=
		|	>=
		|	<
		|	>
23.	SumOp	→	+
		|	-
		|	||
23.	ProductOp	→	*
		|	/
		|	%
		|	&&

3.2 Operator Precedence

Operator precedence is as follows (from lowest to highest):

1. RelationOp
2. SumOp
3. ProductOp
4. Unaryop

Within each group of operators, each operator has the same precedence.

3.3 Operator Associativity

All operators are left associative. For example, **a + b + c** should be interpreted as **(a + b) + c**.

4 Lexical conventions

1. The **identifier** is an unlimited sequence of letters, numbers and the underscore character. An **identifier** *must* begin with a letter or underscore. It may contain upper and lower case letters in any combination.
2. The **number** is an unsigned sequence of digits. It may be preceded with a unary minus operator to construct a negative number. It may also be preceded with a unary plus operator, which does not change its value.
3. The tokens for these two are **IDENT** and **NUMBER**.
4. The decaf language has a number of reserved **keywords**. These are words that cannot be used as identifiers. However, as the language is also case sensitive, versions of the keywords containing uppercase letters can be used as identifiers. It is a compile time error to use a keyword as an identifier. Keywords are separated from **identifiers** by the use of whitespace or punctuation. The keywords are as follows:

int	**void**	**class**	**new**
print	**read**	**return**	**while**
if	**else**	**this**	**null**

Uppercase the keyword to get its token. Except **null** becomes **NULLT**. There is a conflict with C++ if you use **NULL**.

5. The following set of symbols are operators in the decaf language:

[	]	{	}
!=	==	<	>
<=	>=	&&	||
!	+	-	*
/	%	;	,
(	)	=	//
.			

   The tokens for these operators are:

LBRACK	**RBRACK**	**LBRACE**	**RBRACE**
NEQ	**EQ**	**LT**	**GT**
LEQ	**GEQ**	**AND**	**OR**
NOT	**PLUS**	**MINUS**	**TIMES**
DIV	**MOD**	**SEMI**	**COMMA**
LPAREN	**RPAREN**	**ASSIGN**	**COMMENT**
DOT			

6. Whitespace is space, tab and newline. Other than using it to delimit keywords (and the newline for a // comment) it is not required. It will **not** be passed to the parser.

5 Submission

You will submit on WyoCourses only your source code files. The lexical analyzer will be in **program2.lpp** and the main program (driver) will be in **program2.cpp**. Additional header and source code files may also be submitted. Please name them reasonably. Use the extension **.hpp** for header files. Assuming that you create classes (or structs), put their declarations in ***classname*.hpp** and their code (definitions) in ***classname*.cpp** where ***classname*** is the class' or structs' name. Your *make* file will be named **Makefile**. Create a single *tar* archive of all the files. It will be named **program2.tar** or **program2.tgz** if it is compressed. Make sure that you use *gzip* as the compression program, read the *tar* man page. Make sure that you do NOT include any directories in the archive and no hidden or special files. Including unneeded files will result in penalties.