COSC 4785
Compiler Construction
Programming Project 2, Decaf with Flex

1 What to do

You will write a lexical analyzer (scanner) for the Decaf language. This will be done using Flex. As you have already done one assignment similar to this you should not have any huge problems with this.

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 three particular things to consider.

- This language supports both C style comments and C++ style comments. That is a comment can start with the characters "/*" and contains everything until the first occurrence of "*/". Or a comment can begin with "//" and ends at the next newline character.

- 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 it is.

- 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 of the error.

    2. Word errors. If an unrecognized character is followed by some number of letters, digits or '_', scan to the first whitespace or non-word (not error) character and consider the entire "word" the error. Issue an error message indicating the "word" that is in error.

       Example: if the input contains say "A_4TheCoun@t+Not" then the string "A_4TheCoun" is a word, "@t" is an error, "+" is an operator and "Not" is a word.

    3. Quit error. More than 20 errors, issue an error message about too many errors and quit.

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 paren, has no "value" then the value column should be empty.
- Comments and whitespace should not be printed out as tokens, simply ignored in the lexer.
- In the case of errors, the character or word that caused the error will be the token and the error message will be the value.
- The column is the where the token begins. Columns are numbered 1 to *n* starting at the left. This really should not exceed 80 for people using a reasonable editor, but that is NOT any sort of requirement.
- 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 an instructor of mine, 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** it must be able to generate the same language.

3.1 Grammar

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 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**	

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

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

5. 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 will be in **program2.cpp**. Additional header and source code files may also be submitted. Please name them reasonably. I do not care if you use **.h** or **.hpp** as the extension for header files but it would be nice if you are consistent. Assuming that you create classes (or structs) , put their declarations in ***classname*.h** or **.hpp** and their code (definitions) in ***classname*.cpp** where ***classname*** is the class' or structs' name. If you use *make*, please submit the **Makefile** (or **makefile**). You may 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.