UNIVERSITY OF CALGARY

CPSC 411 - Winter 2019

Assignment 1

About This Assignment

- **Subjects:** Lexical Analysis, Scanner, Using Flex
- **Due Date:** Monday, February 4st, at 12:00pm (noon).
- This Assignment has **40 (+ 4 for Bonus) marks** and counts as **10% of your final grade** in the course
- **Late Submissions** will be penalized **10% (4 marks) per day** or portion thereof. Submissions later than one week after the deadline will not be accepted.
- You must put all your submission file (as described in regulation parts) into a single zip file named "**A1.zip**" and upload it on D2L drop-box folder for Assignment 1.

Written Questions (10 Marks)

These questions are designed to help you understand the basic concepts related to the assignment.

1. What is the **role** of **scanner** in a compiler?
2. What is the **principle of longest substring**?
3. Consider the regular expression for a UCalgary email:

   ```
   Email = letter(letter)*\.letter(letter)*digit*@ucalgary\.ca
   ```

   where:

   ```
   letter = [a-z]

   digit = [0-9]
   ```

Complete the following tasks (explain the process):

a. Write down the NFA for **`Email`** using Thompson's Construction method.

b. Transform the NFA to DFA using Subset Construction method.

c. Minimize the number of states for the DFA.

Regulations:

- This part must be done **individually**.
- All answers must be written in **your own words**. Any **quotes** from the Internet or textbook are **unacceptable**.
- All answers to questions 1 and 2 must be **typed** (font size 13pt) into a **single PDF file** named "**1.pdf**".
- Solutions for question 3 must be submitted as a **single PDF** file named "**2.pdf**". You can either type the answers or scan (or take **clear** photos of) your handwritten notes. Please make sure your answers are readable.

Course Project - Phase 1 (30 Marks)

Programming project will direct you to design and build a compiler for an abstract language named C-. Each assignment will cover one component of the compiler: lexical analyzer, parser, semantic analyzer, and code generator. Each phase will ultimately result in a working compiler component which can interface with others.

In this phase of the project you should use Flex to generate a scanner for C- language. Flex is a program that takes as its input a file containing regular expressions, together with the actions to be taken when each expression is matched. The output of Flex is a C source file that is a scanner for the language defined in the input.

Lexical Conventions of C-

You can find the set of tokens in C- language in Table 1. Note that, white space consists of blanks, newline, and tabs. White space is ignored except that it must separate **ID**'s, **NUM**'s and **reserved words**.

In C- language, comments are surrounded by **/*** and ***/**. Comments can be placed anywhere white space can appear (that is, comments cannot be placed within tokens) and they are always single line. Comments may not be nested.

Table 1 - Lexical Conventions of C-

Reserved Words	Special Symbols	Other Tokens
bool	+ - * /	ID = letter letter*
if	< <= > >=	NUM = digit digit*
int	== != =	letter = a|..|z|A|..|Z
else	&& ||	digit = 0|..|9
not	;	
return	,	
true	(	
false	)	
void	[	
while	]	
	{	
	}	
	/*	
	*/	

Your task is to use the above-mentioned C- conventions appropriately in the Flex input file. You can find an introduction to Flex in **Appendix A**.

Scanner Output

The scanner generated by Flex should write the list of tokens in an output file. For example, consider the C- program in Figure 1.

```
/* This is a sample program in C- */
void main (void) {
output(x);
} ~
/* unfinished comment
```

Figure 1 - sample.cm written in C- language

The scanner should generate the output in the format you can see in Figure 2.

```
C- COMPILATION: sample.cm
2: reserved word: void
2: ID, name= main
2: special symbol: (
2: reserved word: void
2: special symbol: )
2: special symbol: {
3: ID, name= output
3: special symbol: (
3: ID, name= x
3: special symbol: )
3: special symbol: ;
4: special symbol: }
4: ERROR: ~
5: ERROR: EOF in comment
```

Figure 2 - Scanner output for sample.cm

Error Handling

For the next phase, errors are communicated to the parser by returning a special error token called **ERROR** (attributes: string value, line number). There are several requirements for reporting and recovering from lexical errors:

- When an invalid character (one that cannot begin any token) is encountered, a string containing just that character should be returned as the error string. Resume lexing at the following character.
- If a comment remains open when EOF is encountered, report this error with the message "**`EOF in comment`**". Do not tokenize the comment's contents simply because the terminator is missing.
- If you see "**`*/`**" outside a comment, report this error as "**`Unmatched */`**", rather than tokenizing it as **`*`** and **`/`**.
- If you see just "**`/*`**" inside a comment("**`/* /* */`**"), it's not an error. However, you don't have to support nested comments like "**`/* /*.../ */`**".
- This phase of the compiler only catches a very limited class of errors. Do not check for errors that are not lexing errors in this assignment. For example, you should not check if variables are declared before use. Be sure you understand fully what errors the lexing phase of a compiler does and does not check for before you start.

Regulations:

- The project can be completed **Individually** or in a **Group of Two**.
- You must create a **User Manual** (PDF format) for your solution explaining **how to use the Flex input file to generate the scanner**. Also, you should **report** what works in your solution and what does not (be honest).
- If the project is accomplished in a group, **full name** and **UCID** of both members should be written at the beginning of the user manual.
- Your submission for this part must be a Flex input file name "**a1.flex**" and a user manual in PDF format named "**manual.pdf**". If the project is accomplished in a group, only one of the members should submit the solution and the user manual.

Bonus Part (Optional - 4 Marks)

There are up to 4 marks available for the implementation of **multi-line comments** in the project.

Useful Tips

- **Attending to tutorials** will give you enough knowledge to complete the assignment.
- You can find a definitive (and helpful) manual for flex in http://westes.github.io/flex/manual/.

Appendix A - Flex Introduction

Flex allows you to implement a lexical analyzer by writing rules that match on user-defined regular expressions and performing a specified action for each matched pattern. Flex compiles your rule file (e.g., "lexer.flex") to C source code implementing a finite automaton recognizing the regular expressions that you specify in your rule file. Fortunately, it is not necessary to understand or even look at the automatically generated (and often very messy) file implementing your rules. Rule files in flex are structured as follows:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

Figure 3 - structure of Flex input file

The Declarations and User subroutines sections are optional and allow you to write declarations and helper functions in C. The Definitions section is also optional, but often very useful as definitions allow you to give names to regular expressions. For example, the definition **`DIGIT [0-9]`** allows you to define a digit. Here, **`DIGIT`** is the name given to the regular expression matching any single character between **`0`** and **`9`**. The following table gives an overview of the common regular expressions that can be specified in Flex:

Regular Expression	Description
`x`	**the character "x"**
`"x"`	**an "x", even if x is an operator.**
`\x`	**an "x", even if x is an operator.**

[xy]	**the character x or y.**
[x-z]	**the characters x, y or z.**
[^x]	**any character but x.**
.	**any character but newline.**
^x	**an x at the beginning of a line.**
<y>x	**an x when Flex is in start condition y.**
x$	**an x at the end of a line.**
x?	**an optional x.**
x*	**0,1,2, ... instances of x.**
x+	**1,2,3, ... instances of x.**
x|y	**an x or a y.**
(x)	**an x.**
x/y	**an x but only if followed by y.**
{xx}	**the translation of xx from the definitions section.**
x{m,n}	**m through n occurrences of x**

The most important part of your lexical analyzer is the rules section. A rule in Flex specifies an action to perform if the input matches the regular expression or definition at the beginning of the rule. The action to perform is specified by writing regular C source code. For example, if a digit represents a token in our language (note that this is not the case in C-), the rule:

```
{DIGIT} {
     yylval.symbol = inttable.add_string(yytext);
     return DIGIT_TOKEN;
}
```

records the value of the digit in the global variable **yylval** and returns the appropriate token code. An important point to remember is that if the current input (i.e., the result of the function call to **yylex()**) matches multiple rules, Flex picks the rule that matches the largest number of characters. For instance, if you define the following two rules:

```
[0-9]+ { // action 1}

[0-9a-z]+ {// action 2}
```

and if the character sequence **2a** appears next in the file being scanned, then **action 2** will be performed since the second rule matches more characters than the first rule. If multiple rules match the same number of characters, then the rule appearing first in the file is chosen. When writing rules in Flex, it may be necessary to perform different actions depending on previously encountered tokens. For example, when processing a closing comment token, you might be interested in knowing whether an opening comment was previously encountered. One obvious way to track state is to declare global variables in your declaration section, which are set to true when certain tokens of interest are encountered. Flex also provides syntactic sugar for achieving similar functionality by using state declarations such as:

```
%Start COMMENT
```

which can be set to true by writing **BEGIN(COMMENT)**. To perform an action only if an opening comment was previously encountered, you can predicate your rule on **COMMENT** using the syntax:

```
<COMMENT> {

// the rest of your rule ...

}
```

There is also a special default state called **INITIAL** which is active unless you explicitly indicate the beginning of a new state. You might find this syntax useful for various aspects of this assignment, such as error reporting.