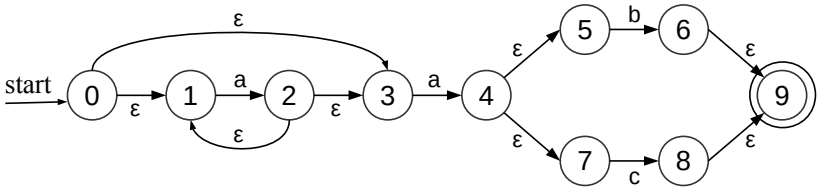
Homework #5 Solution

Question 1 (4 pt.)

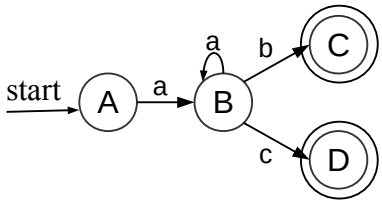
Consider regular expression `a*a(b|c)`.

a) (1 pt.) Use the McNaughton-Yamada-Thompson algorithm to construct an NFA that accepts the language represented by this regular expression. Do not skip any steps of the algorithm, even if you think that some of the inserted states or transitions might be redundant.



b) (1 pt.) Convert the generated NFA into an equivalent DFA, using the algorithm presented in class, based on the *ε-closure* and *move* operations on the NFA states. Construct the DFA transition table, where each set of reachable NFA states is associated with an equivalent DFA state, and give the transition graph for the resulting DFA.

S_{NFA}	s_{DFA}	a	b	c
{0, 1, 3}	A	B	-	-
{1, 2, 3, 4, 5, 7}	B	B	C	D
{6, 9}	C	-	-	-
{8, 9}	D	-	-	-



c) (2 pt.) Write a C++ program that takes an input string as an argument and prints whether this string is matched by the regular expression above. Use class `DFA` (included in header file `DFA.h` available in the additional material) to simulate the automaton you constructed in the previous question. Attach a file named `q1.cc` with your code. Your source code should build without modifications when linked together with file `DFA.cc`.

```
#include <iostream>
#include "DFA.h"

int main(int argc, char **argv)
{
	// Check syntax
	if (argc != 2)
	{
		std::cerr << "Syntax: ./q1 <string>\n";
		return 1;
	}

	// Construct DFA
	DFA dfa;
	dfa.setFinalState(2);
	dfa.setFinalState(3);
	dfa.addTransition(0, 'a', 1);
	dfa.addTransition(1, 'a', 1);
	dfa.addTransition(1, 'b', 2);
	dfa.addTransition(1, 'c', 3);

	// Parse string
	bool accepted = dfa.accepts(argv[1]);
	std::cout << (accepted ? "String accepted\n" : "String rejected\n");
	return 0;
}
```

Question 2 (4 pt.)

Write a Flex program that scans the input code shown below, and splits it into tokens. For each token, the program should print its name and the associated lexeme. Upload your code in a file named `q2.l`.

You can load the Flex input from a text file, instead of the standard input, by opening the file with an `std::ifstream` object, and passing a reference to it in the constructor of class `yyFlexLexer`:

```
std::ifstream f("code.txt");
yyFlexLexer lexer(&f);
```

Also, you can add `#include` files in the first section of the Flex file using a block headed by `%{` and ended by `%}`, as follows:

```
%{
#include <fstream>
%}
```

This is the program to scan, which you can store, for example, in a file named `code.c`:

```
/* Main function */
int main( )
{

    float m, n;
    printf("\nEnter a number: ");
    scanf("%f", &m);

    // Function call
    n = square(m);
    printf("\nSquare of the given number %f is %f", m, n);
}
```

Solution

```
%option noyywrap

%{
#include <fstream>
%}

LETTER_ [a-zA-Z_]
NUMBER [0-9]

%%

\/\*[^\/\*]*\*\/ {
      std::cout << "Token COMMENT: " << yytext << '\n';
}

\/\/[^\n]* {
      std::cout << "Token CPP_COMMENT: " << yytext << '\n';
}

\"[^\"]*\" {
      std::cout << "Token STRING: " << yytext << '\n';
}

"," {
      std::cout << "Token COMMA: " << yytext << '\n';
}

"&" {
      std::cout << "Token ADDRESS_OF: " << yytext << '\n';
}

"(" {
      std::cout << "Token OPEN_PAR: " << yytext << '\n';
}

")" {
      std::cout << "Token CLOSE_PAR: " << yytext << '\n';
}

"{" {
      std::cout << "Token OPEN_BRACKET: " << yytext << '\n';
}

"}" {
      std::cout << "Token CLOSE_BRACKET: " << yytext << '\n';
}

"=" {
      std::cout << "Token EQUAL: " << yytext << '\n';
}

";" {
      std::cout << "Token SEMICOLON: " << yytext << '\n';
}

float {
```

```
      std::cout << "Token FLOAT: " << yytext << '\n';
}

{LETTER_}({LETTER_}|{NUMBER})* {
      std::cout << "Token ID: " << yytext << '\n';
}

[ \n\t]+ {
      std::cout << "Token SPACE: " << yytext << '\n';
}

. {
      std::cout << "Token ERROR: " << yytext << '\n';
}

%%

int main()
{
      std::ifstream f("code.txt");
      yyFlexLexer lexer(&f);
      lexer.yylex();
      return 0;
}
```

Question 3 (1 pt.)

Specify the languages that the following context-free grammars represent. You can use either a formal set notation, or a text-based description—no need for both. If you choose a description, make sure that it is concise and accurate, with no room for ambiguity.

For example:

Grammar	Set notation	Description
$S \rightarrow aSb \mid \epsilon$	$L = \{ a^n b^n \mid n \geq 0 \}$	Language composed of the empty string, and strings containing a sequence of *a*'s followed by the same number of *b*'s.

a) $S \rightarrow aSa \mid aBa$
$B \rightarrow bB \mid b$

Solution: $L = \{ a^n b^m a^n \mid n > 0,\ m > 0 \}$

b) $S \rightarrow aSb \mid aSbb \mid \epsilon$

Solution: $L = \{ a^n b^m \mid 0 \leq n \leq m \leq 2n \}$

Question 4 (1 pt.)

Provide context-free grammars for the following languages on the binary alphabet $\Sigma = \{0,\ 1\}$:

a) Language formed of strings that start and end with the same binary digit, not including the empty string.

Solution: $S \rightarrow 0A0 \mid 1A1$
$A \rightarrow 0A \mid 1A \mid \epsilon$

b) Strings with an odd number of digits.

Solution: $S \rightarrow 0A \mid 1A$
$A \rightarrow 0S \mid 1S \mid \epsilon$