Course Administration

- Exercise 2 Questions
- Project 1 Preliminary Exercise
Exercise 2 Questions

- (1a) I would like some clarification on our currency dynamic array.
- I understand, (correct me if I'm wrong), that if our dynamic array contains [101] it means we have 1 penny, 0 nickels, and 1 dime (assuming we are going in ascending order). So if we wanted to produce our currency array for $100 it would look something like this? [0,0,0,0,0,0,0,0,0,0,1]

- (1b) I am also a little confused as to how exactly our currency is going to be updated.
- How exactly should we update our currency array? For example, if we initialized an object that contained $6.97, where would our calculations, (to find how many and what kind of denominations are needed), go? I understand we could initially do the calculations in that constructor, however, if we were to add to that same object we would need to update our currency array. **Would we be able to have a helper function that aids in doing that?** I see we have a set currency function that is supposed to set our currency array. Can we assume that that calculation is done apart from the class and that the vector passed already contains the updated/optimized denominations count?
DFAs as scanners (aka tokenizers)

- Alphabet = \{a, b, c, x, y, z, \top\}
- Regular expression 1 (RE1)
  - \(a^* (ab | bc) a^+\)
- Regular expression 2 (RE2)
  - \(x^+ (xy | yz | xz) z^*\)
- Combined
  - \((a^* (ab | bc) a^+) | (x^+ (xy | yz | xz) z^*)\)
\((a^* (ab \mid bc) a^+) \mid (x^+ (xy \mid yz \mid xz) z^*)\)
## Programming a DFA

### Table

<table>
<thead>
<tr>
<th>ws</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Err1</td>
<td>7</td>
<td>Err1</td>
<td>Err1</td>
</tr>
<tr>
<td>2</td>
<td>Err2</td>
<td>2</td>
<td>4</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
</tr>
<tr>
<td>3</td>
<td>Err2</td>
<td>Err2</td>
<td>5</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
</tr>
<tr>
<td>4</td>
<td>Err2</td>
<td>6</td>
<td>Err2</td>
<td>5</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
</tr>
<tr>
<td>5</td>
<td>Err2</td>
<td>6</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
</tr>
<tr>
<td>6</td>
<td>RE1</td>
<td>6</td>
<td>RE1</td>
<td>RE1</td>
<td>RE1</td>
<td>RE1</td>
<td>RE1</td>
</tr>
<tr>
<td>7</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>8</td>
<td>9</td>
<td>Err2</td>
<td>Err2</td>
</tr>
<tr>
<td>8</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>Err2</td>
</tr>
<tr>
<td>9</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>Err2</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>RE2</td>
<td>11</td>
</tr>
</tbody>
</table>
#include <iostream>
#include "Lex.h"

using namespace std;

int main ()
{
    string input;
    getline (cin, input);
    cout << "Input:\n" << input << endl;
    Lex tokenizer (input);
    token t = NONE;
    while (t != EOLN)
    {
        t = tokenizer.getNext ();
        cout << tokenizer.getTokenName (t) << '\t';
        cout << tokenizer.getLexeme () << '\n';
    }
    return 0;
}
Lex.h

#ifndef LEX_H
#define LEX_H

#include <iostream>
using namespace std;

enum token {Err2 = -4, Err1, RE2, RE1, NONE, EOLN};

class Lex
{
    public:
        Lex(const string & S);
        token getNext ();
        string getLexeme ();
        string getTokenName (token t);

    private:
        string line;
        int pos;
        string lexeme;
};

#endif
Lex.cpp (1)

#include "Lex.h"

#define DEBUG 0
#define debug if (DEBUG) cout

Lex::Lex(const string & S)
{
    line = S;
    pos = 0;
}

string Lex::getLexeme ()
{
    return lexeme;
}

string Lex::getTokenName (token t)
{
    switch (t)
    {
    case RE1: return "RE1";
    case RE2: return "RE2";
    case Err1: return "Err1";
    case Err2: return "Err2";
    case EOLN: return "EOLN";
    }
    return "NONE";
}
token Lex::getNext ()
{
    lexeme = "";
    while (pos < line.length() && isspace (line[pos]))
        pos++;
    if (pos >= line.length())
        return EOLN;
    char oneChar;
    int col;
    int state = 1;
    debug << pos << " : " << line << endl;
    while (state > 0)
    {
        oneChar = line[pos++];
        col = getColumn (oneChar);
        lexeme += oneChar;
        debug << "state = " << state << "; position = " << pos << "
        ";
        lexeme += oneChar << endl;
        state = table[state][col];
    }
    if (state != Err1)
    {
        lexeme.pop_back();
        pos--;
    }
    return token (state);
}
static int table[12][8] =
{
    {0, 0, 0, 0, 0, 0, 0, 0},
    {1, 2, 3, Err1, 7, Err1, Err1, Err1},
    {Err2, 2, 4, Err2, Err2, Err2, Err2, Err2},
    {Err2, Err2, Err2, 5, Err2, Err2, Err2, Err2},
    {Err2, 6, Err2, 5, Err2, Err2, Err2, Err2},
    {Err2, 6, Err2, Err2, Err2, Err2, Err2, Err2},
    {RE1, 6, RE1, RE1, RE1, RE1, RE1, RE1},
    {Err2, Err2, Err2, Err2, 8, 9, Err2, Err2},
    {Err2, Err2, Err2, Err2, 8, 10, 10, Err2},
    {Err2, Err2, Err2, Err2, Err2, Err2, 11, Err2},
    {RE2, RE2, RE2, RE2, RE2, RE2, 11, RE2},
    {RE2, RE2, RE2, RE2, RE2, RE2, 11, RE2}
};
Inputs and Execution

- **input1**
  abca xyz

- **input2**
  abcaxyz

- **input3**
  abcaabxyz

- **input4**
  abcaxyzzzzaabbcaabcaxxyzzzcz
Project 1

- **Project1Framework**
  - LexicalAnalyzer.h
  - LexicalAnalyzer.cpp
  - SyntacticalAnalyzer.h
  - SyntacticalAnalyzer.cpp
  - Project1.cpp
  - makefile
  - README

- **Test Files**
  - P1-1.pl460
  - P1-1.lst
  - P1-1.p1
  - P1-2.pl460
  - P1-2.lst
  - P1-2.p1

- **In course pickup directory**