COMP 6421 the Report of Assigment1

Liao Xiaoyun

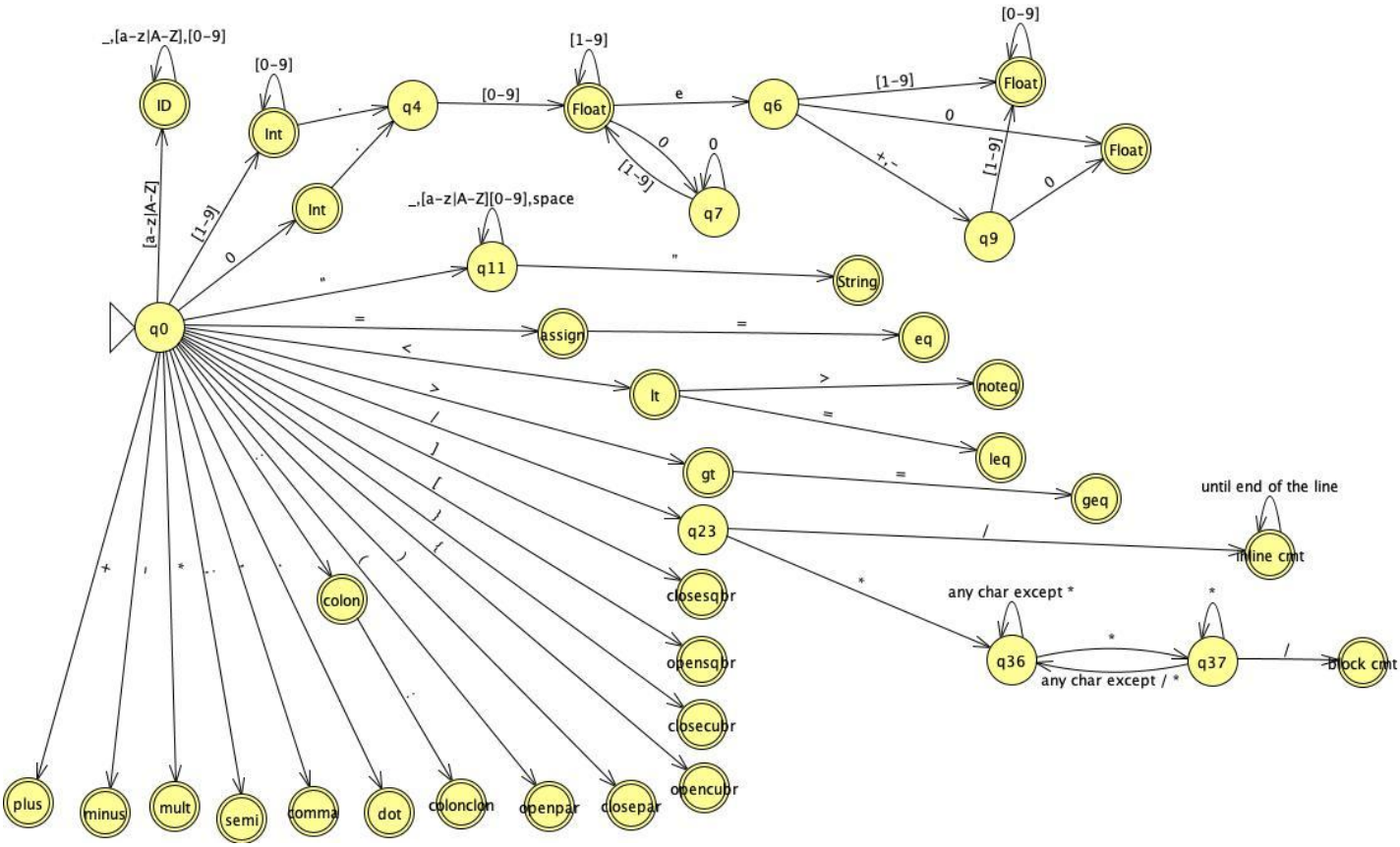
40102049 sharon.liaoxy@gmail.com

1. Lexical specifications

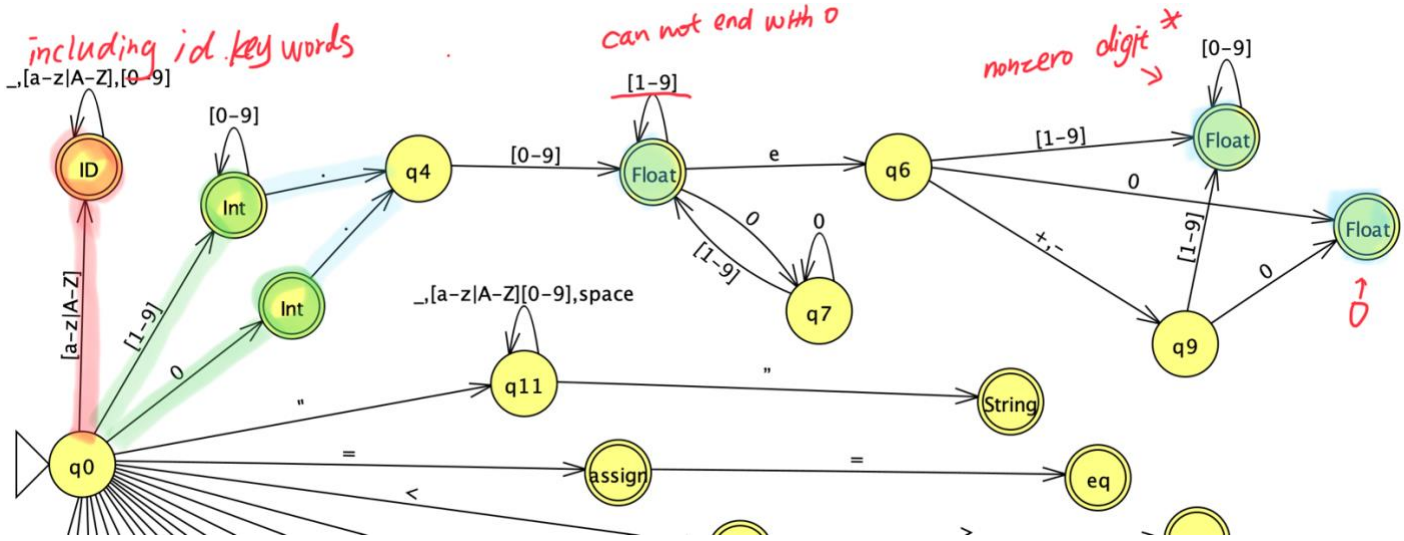
In this section, I used the letter l to represent letters a to z and A to Z, n to represent nonzero, and z to represent zero.

- l ::= [a-z|A-Z]
- n ::= [1-9]
- z ::= 0
- Id ::= l(l|(n|z)|_)*
- Int ::= n(n|z)*|z
- Fraction ::= (.(n|z)*n|.z)
- Float ::= (n(n|z)*|z)(.(n|z)*n|.z)(e(p|m)?(n(n|z)*|z))? // p -> + , m -> -
- Char ::= (l|(n|z)|_)|s // s -> space
- String ::= "((l|(n|z)|_)|s)*"
- inlineCmt ::= //(c)*t // c-> any char, t -> end of the line
- blockCmt :: = b(c)*b // b -> /*

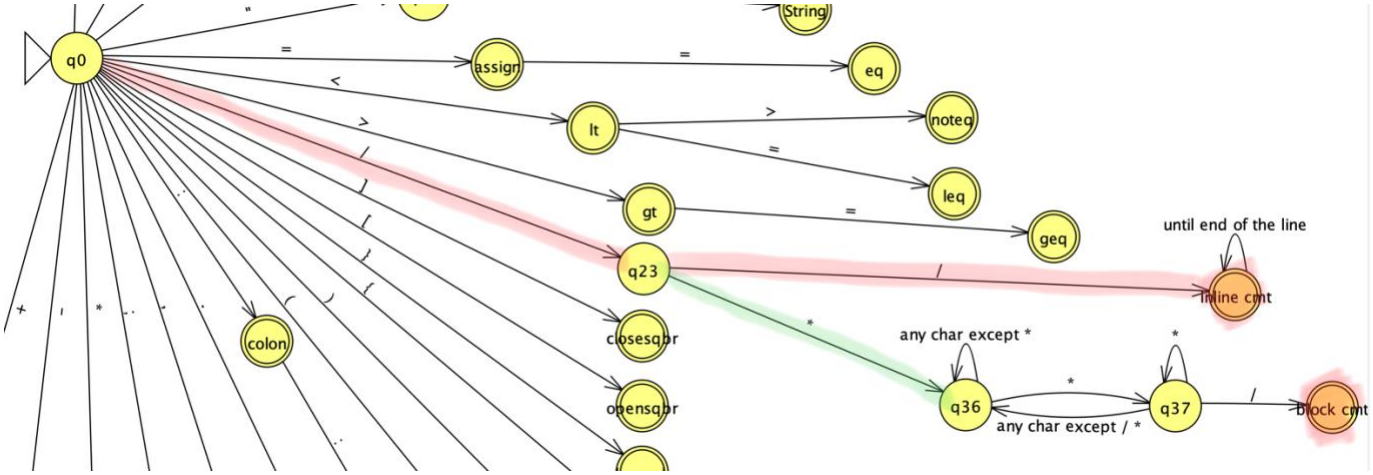
2. Finite state automaton:



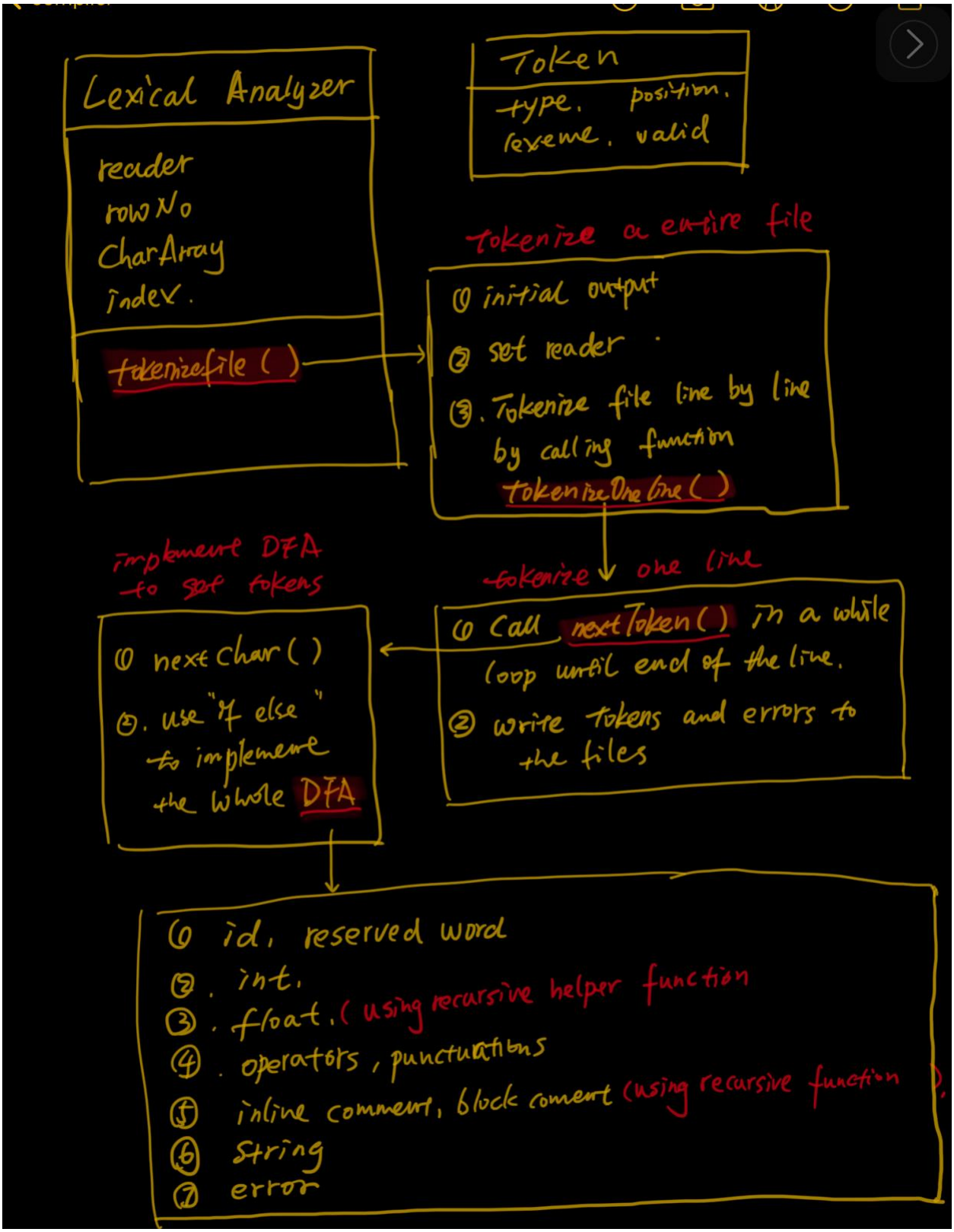
- Id, Int, Float



- Inline comment, block comment



3. Design:


Lexical Analyzer
reader
rowNo
CharArray
index.
tokenizefile()
Token
type. position.
lexeme. valid
tokenize a entire file
① initial output
② set reader
③. Tokenize file line by line by calling function tokenizeOneLine()
implement DFA to get tokens
① nextChar()
②. use "if else" to implement the whole DFA
tokenize one line
① Call nextToken() in a while loop until end of the line.
② write tokens and errors to the files
① id, reserved word
②. int.
③. float. (using recursive helper function
④. operators, punctuations
⑤ inline comment, block coment (using recursive function).
⑥ String
⑦ error

- Token class

  Firstly, define a Token class.

```
public class Token {
    String tokenType;
    String lexeme;
    int position;
    boolean valid;
```

- LexicalAnalyzer class

Then, define a class LexicalAnalyzer to tokenize an entire file, now let's have a look inside it.

```
public class LexicalAnalyzer {
    String filename;
    String outFile;
    String errorFile;
    Scanner reader; // the file reader, used to read the testing file line by line
    int rowNo; // token position
    char[] charArray; // cover current code line to a char array
    int index; //index of charArray
```

- tokenizeFile() function

  In this class, the function tokenizeFile() is used to tokenize an entire testing file.

```
public void tokenizeFile(String filename) {
    try {
        //create the out file and error file
        this.initialOutput(filename);

        //start to read the testing file
        File file = new File( pathname: "src/testFiles/"+filename+".src");
        this.reader = new Scanner(file);

        //tokenize the file line by line
        while (reader.hasNextLine()) {
            String str = reader.nextLine().trim();
            System.out.println("\n"+str);

            //tokenize a single line code
            tokenizeOneLine(str);
        }
        reader.close();
```

- tokenizeOneLine() function

```
private ArrayList<Token> tokenizeOneLine(String codeLine) throws IOException {

    ArrayList<Token> tokens = new ArrayList<>();

    // update rowNo, and cover the current code string
    // to a char array and reset index of array
    initialContext(codeLine);

    // while loop call nextToken() to generate tokens
    while (index < charArray.length-1) {
        Token token = nextToken();
        if (token.tokenType != null) {
            tokens.add(token);
            System.out.println(token.toString());
        }
    }

    // write tokens and errors to the files
    writeResult(tokens);
    return tokens;
}
```

- nextToken() function

```
public Token nextToken() {
    // create a new token
    Token token = new Token();

    //use StringBuffer to store the lexeme
    StringBuffer lexeme = new StringBuffer();

    char c =  nextChar();

    while (c == ' ' | c == '\t' ){
        c = nextChar();
    }
```

- Match Id and reserved words

```
// match id and reserved words
if (isLetter(c)) {
    lexeme.append(c);
    c = nextChar();
    while (isLetter(c) || isDigit(c) || c == '_') {
        lexeme.append(c);
        c = nextChar();
    }
    //token end
    // check if id end correctly
    if (isTransferChar(c)){
        // reserved words are prior than id, so check if token is a reserved word

        // if true then save token as a reserved word otherwise save as an id
        if (isReservedWord(lexeme.toString())) {
            //save token (type, lexeme, is valid, rowNO)
            saveToken(token, lexeme.toString(), lexeme, valid: true);
        } else {
            saveToken(token,  tokenType: "id", lexeme, valid: true);
        }
        // need back up one char
        backupChar();

    } else {
        // catch the entire error token
        // ex: abg@1133
        while ( !isTransferChar(c)){
            lexeme.append(c);
            c = nextChar();
        }
        saveToken(token,  tokenType: "id", lexeme, valid: false);
        backupChar();
    }
```

- Match int and float number

```
//  match int
    // start with 0
} else if( c == '0') {
    lexeme.append(c);
    c = nextChar();
    // match float
    if (c == '.') { // 0.1222
        lexeme.append(c);
        c = nextChar();
        //a helper function, which is a recursive function, to check if this token is valid
        floatState1Check(c,lexeme,token);
    } else {
        // check if this token end correctly, ex: 0abbb, invalid; 0 abb, good
        checkIfEndCorrect(c,token, tokenType: "intnum",lexeme);
    }
// start with 1-9
} else if(isDigit(c)) {
    lexeme.append(c);
    c = nextChar();
    while ( isDigit(c) ) {
        lexeme.append(c);
        c = nextChar();
    }
    // match float
    if (c == '.') { // 0.1222
        lexeme.append(c);
        c = nextChar();
        floatState1Check(c,lexeme,token);
    } else {
        checkIfEndCorrect(c,token, tokenType: "intnum",lexeme);
    }
```

- Match operators and punctuations

```
}else if ( c == '.') {
    saveToken(token,   tokenType: "dot", new StringBuffer().append("."), valid: true);
}else if ( c == ',') {
    saveToken(token,   tokenType: "comma", new StringBuffer().append(","), valid: true);
}else if ( c == ':') {
    lexeme.append(c);
    c = nextChar();
    if ( c == ':') {
        lexeme.append(c);
        saveToken(token,   tokenType: "coloncolon", lexeme, valid: true);
    } else {
        saveToken(token,   tokenType: "colon", lexeme, valid: true);
        backupChar();
    }
```

- Match inline comment and block comment

```
}else if ( c == '/') {
    lexeme.append(c);
    c = nextChar();
    if ( c == '/') { // catch the entire comment line
        while (index < charArray.length-1) {
            lexeme.append(c);
            c = nextChar();
        }
        lexeme.append(c);
        saveToken(token,   tokenType: "inlinecmt", lexeme, valid: true);
    } else if ( c == '*') { // catch the entire block comment
        lexeme.append(c);
        c = nextChar();
        token.position = rowNo; // record the rowNo of the first line
        // define a recursive helper function to catch the entire block comment
        blockcmtEndCheck(c,lexeme,token);
    } else {
        // div
        saveToken(token,   tokenType: "div", lexeme, valid: true);
        backupChar();
    }
```

- Match a string

```
} else if ( c == '\"') {
    lexeme.append(c);
    c = nextChar();
    // catch the entire string
    while (!(index == charArray.length | c == '\"')) {
        lexeme.append(c);
        c = nextChar();
    }
    lexeme.append(c);
    // check if string ends correctly
    if(c == '\"') {
        saveToken(token,  tokenType: "stringlit", lexeme, valid: true);
    } else {
        saveToken(token,  tokenType: "stringlit", lexeme, valid: false);
    }
```

- Driver

```
public class Driver {
    public static void main(String[] arg){
        System.out.println("compiler - lexical analyzer");
        LexicalAnalyzer lexerAnalyzer = new LexicalAnalyzer();

        lexerAnalyzer.tokenizeFile( filename: "lexpositivegrading");
        lexerAnalyzer.tokenizeFile( filename: "lexnegativegrading");
        lexerAnalyzer.tokenizeFile( filename: "myTest");
    }
}
```

4. Use of tools

- https://cyFberzhg.github.io/toolbox/min_dfa?regex=KGF8Yikq

  I used this tool to generate DFA for each regular expression,

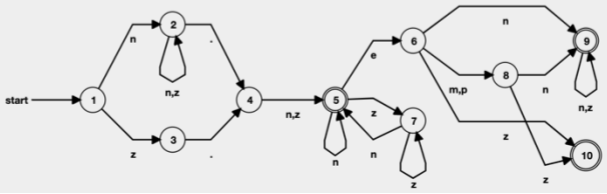
  Foe example:

  Float ::= (n(n|z)*|z)(.(n|z)*n|.z)(e(p|m)?(n(n|z)*|z))? // p -> + , m -> -

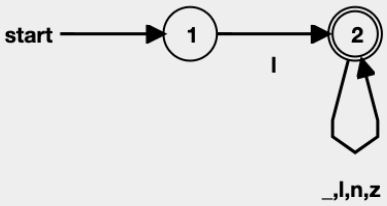
Input: (n(n|z)*|z)(.(n|z)*n|.z)(e(p|m)?(n(n|z)*|z))?

DFA: https://cyberzhg.github.io/toolbox/nfa2dfa?regex=KG4obnx6KSp8eikoLihufHopKm58LnopKGUocHxtKT8obihufHopKnx6KSk/

DFA STATE	Min-DFA STATE	TYPE	.	e	m,p	n	n,z
{A}	1					2	
{B,E,F}	2		4				2
{C}	3		4				
{D}	4						5
{G,H}	5	accept		6		5	
{I}	6				8	9	
{J}	7					5	
{K,M}	8					9	
{L,O,P}	9	accept					9
{N}	10	accept					



Id ::= l(l|(n|z)|_)*



- JFLAP

  After collecting all DFAs of regular expressions, I used JFLAP to gather all DFAs to a single DFA, which include all final states.