Concordia University

Winter 2019

# Assignment 1

# Lexical Analyzer Documentation

COMP 442

Written by:

Genevieve Plante-Brisebois

ID:40003112

Presented to:

Dr. Joey Paquet

January 28, 2019

**Lexical Specifications:**

Atomic lexical elements of the language:

id:: = Letter alphanum*

alphanum:: = letter | digit| _

integer::= nonzero digit* |0

float::= integer fraction [e[+|-] integer]

fraction::= .digit* nonzero | .0

letter::= a.. z |A..Z

digit: 0+1..9

nonzero::=1..9

operator::= == | <> | < |> | <= | >= | + | - | * | / | = | && | || | !

punctuation :: = ; | , | . | : | :: | ( | ) | { | } | [ | ]

keywords::= if | then | else | for | class | integer | float | read | write |return | main

comments::=/*|*/ |//

The changes that were made were only to put the operators, punctuation and keywords in their respective token identification and regular expression. The rest of the lexical specifications have been kept as they are.

To my understanding an alphanumeric is a single letter or digit but as there can be any combination of those in the id it creates the alphanumerical characteristic.

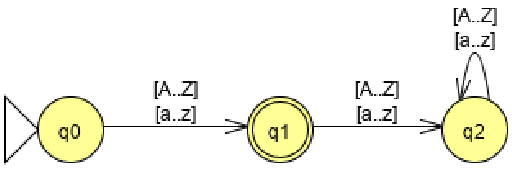
The regular expressions were translated into NFA and then DFA in a manual manner using the knowledge acquired in the COMP 335 class. The conversion to numerical format has been done using the JFLAP software. It is a tool use in COMP 335 which allows to draw the automaton in a clean and clear manner. In the next section, it will be possible to see each individual automata for the regular expression and then the final automata which has all of the lexical specifications in it. The final automata has been built by combining all the automata.

As I am using a patterns library (see tools section) and to simplify the keywords DFA I have simply put that if the proper keywords have been put then we reach a final state.

**Finite State Automaton:**

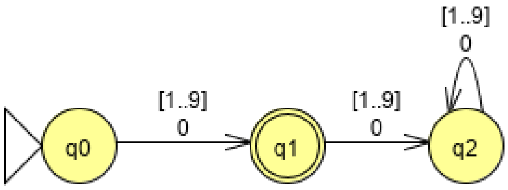
Individual DFA's for the lexical specifications:

Letter:



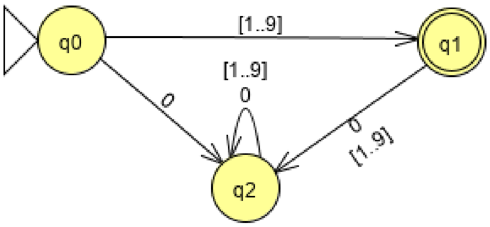
The last state has been added for additional clarity in terms of the fact that only a single char from the presented set is accepted. Else it goes in trap state.

Digit:



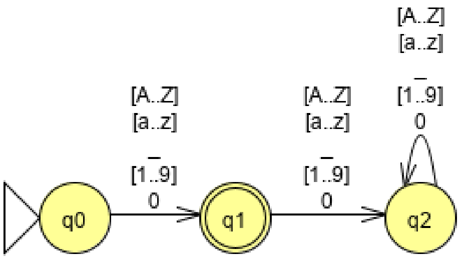
Just like for the letter, the last state has been added for clarity as this is the DFA for a digit and not an integer. If there is more than one digit then it goes in trap state. The terms 0 and [1..9] have been used in order to facilitate the combination of the multiple DFA together as other DFA require both digits and nonzero.

Nonzero:



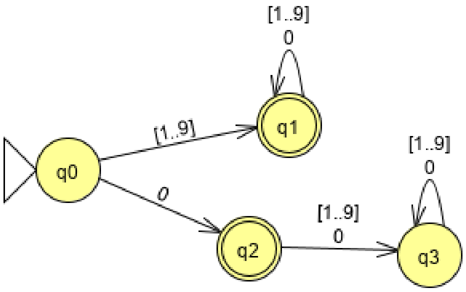
In the case that a zero is used first then all the other attempts made to get a final result will be ending in a trap state. Moreover, as it is a single digit number if there is more than one digit it also ends in a trap state.

Alphanum:

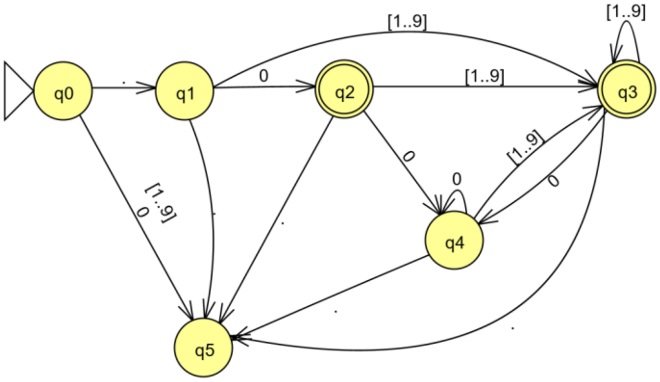


It is my understanding from the regular expression supplied in the lexical specification that the alphanumerical is only one character but it can be a letter, digit or _ . It is with the * in the ID that the alphanumerical nature of the string will be made.

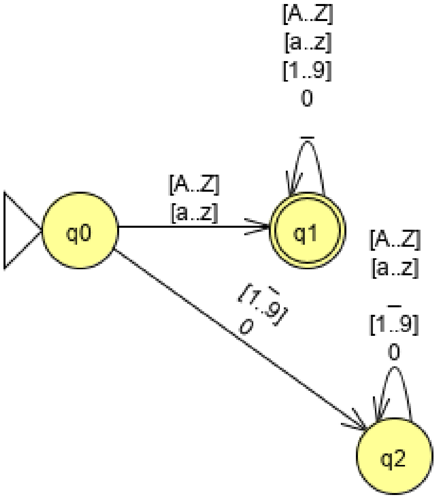
Integer:



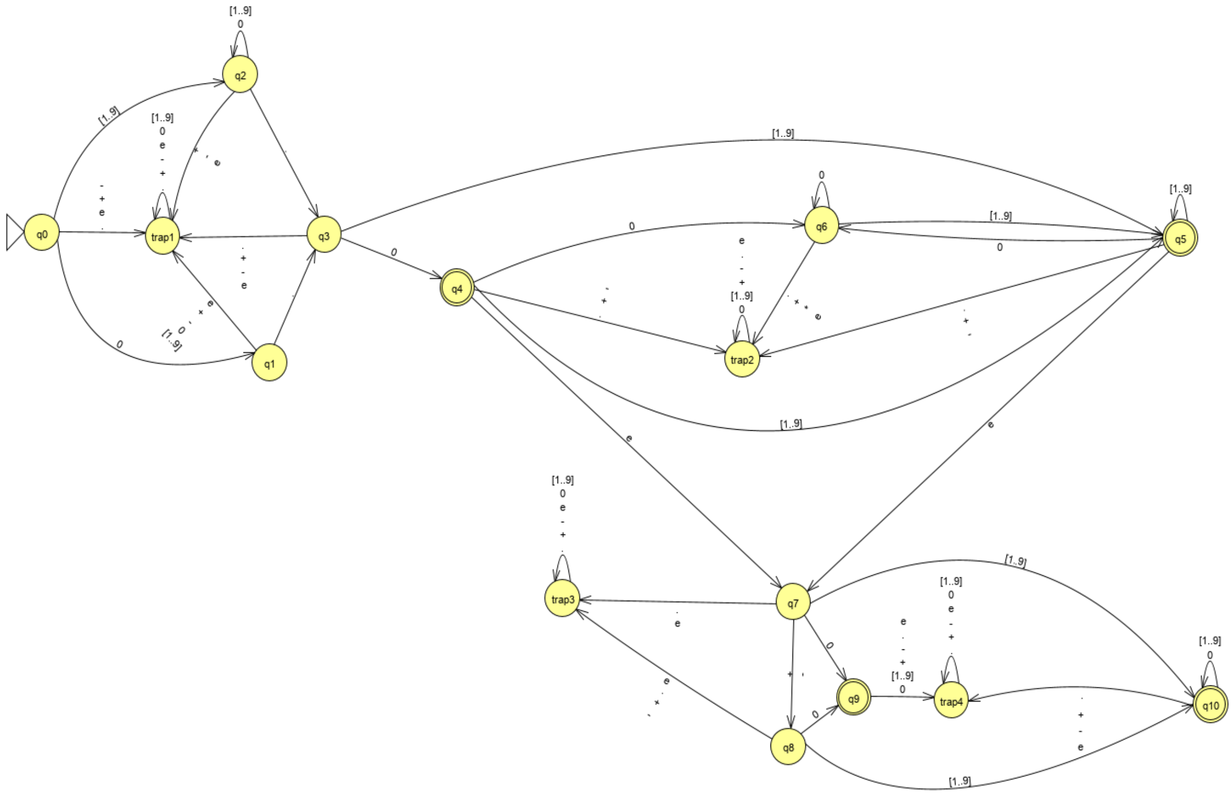
Fraction:



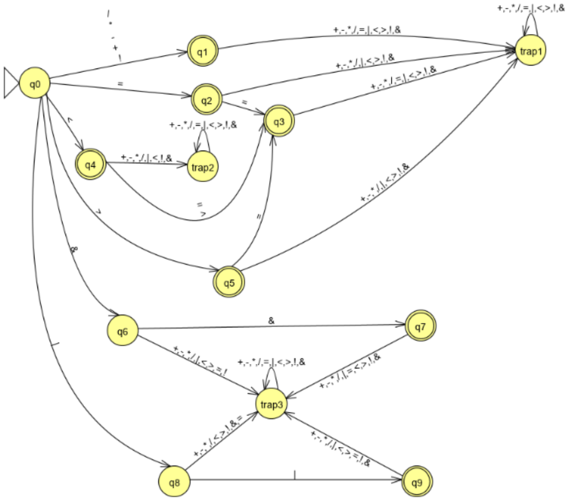
ID:



Float:

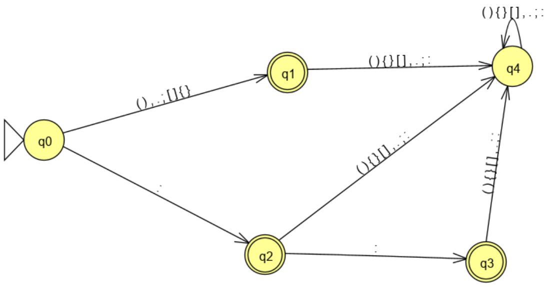


Operator:

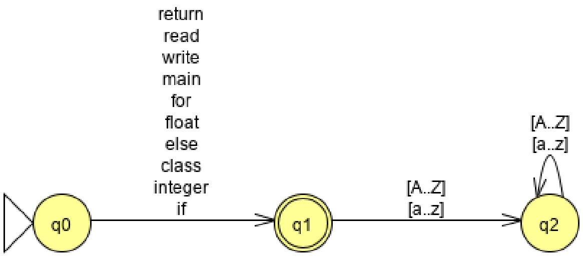


In this graph, the various symbols that can be used are separated by a coma in order to increase readability of the graph. There are ten possible symbols that can be used as operators. They are +,-,*,/,=,|,<,>,!,& . The various combinations of them are described in the lexical specifications and give this DFA

Punctuation:

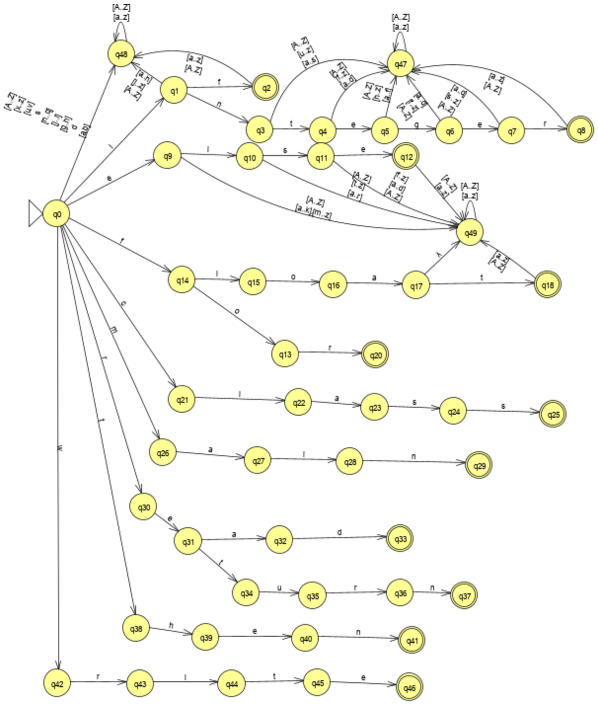


Keywords:

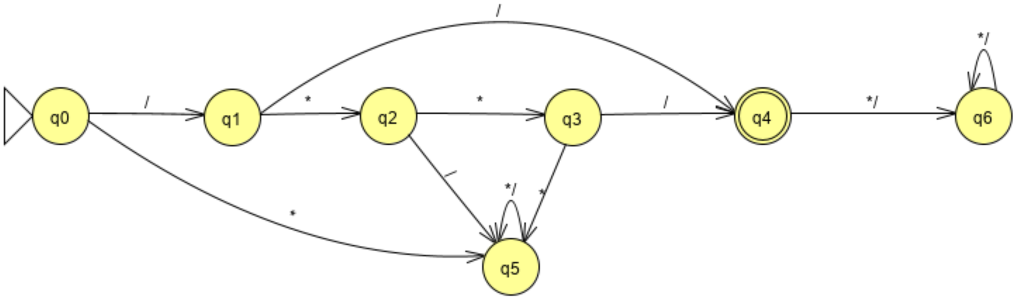


The keywords DFA is quite simple in this format but a more exhaustive DFA could also be done. For the sake of saving time and since it is possible to implement a patterns in Java so to create a bank of keywords that will be recognized when scanning a string this simpler version of this DFA has been drawn. It is also more readable than another model.

Just to show that the other model is rather hard to read here is a partial DFA, so still in NFA state of what it would give to do the exhaustive method:


q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
q13
q14
q15
q16
q17
q18
q20
q21
q22
q23
q24
q25
q26
q27
q28
q29
q30
q31
q32
q33
q34
q35
q36
q37
q38
q39
q40
q41
q42
q43
q44
q45
q46
q47
q48
q49
[A..Z]
[a..z]
[a..h]
[j..z]
[a..s]
[u..z]
[a..d]
[f..z]
[a..k][m..z]
[a..r]
[t..z]
i
n
f
t
e
g
r
l
s
o
a
c
m
d
u
h
w

Comments:


q0
q1
q2
q3
q4
q5
q6
/
*
*/

**Design:**

To make the lexer, I used the individual DFA’s of each in order to make Boolean functions which will verify if the string they are looking at corresponds to the token being asked.

I did not have time to go further in the implementation due to the issues mentioned earlier. However here is how I would continue the structure of the lexer:

I would make a reader which reads the input text and separates the terms when there is a white space. The chars separated by the white space would then become the input string that I would use to compare it to with the Boolean expressions mentioned above. In order of how to compare the cases, I would start from the most precise to the least precise i.e. verify for float, then fraction, then integer, then nonzero, then digit. This ordering is used in order to prevent the matching of a more global definition (in example letter by letter each token being tagged as a letter) instead of their more precise requirements. The moment a true status comes, the string would be added to the arraylist with proper token type and the content.

In the case of a comment, the moment an opening comment block is detected it would keep scanning the input until there is a valid comparison for the closing of the comment block. The whole section would be given to the token as a comment.

There is an inner class of Tokens in order to have an object which has both the token value and the data that it must store. The class only has accessors, a toString, a default constructor and a type specific constructor. Tokens should not be modified once they are set, only accessing the information. If modifications are needed later on in the project, the inner class Token can be found at the bottom of the Lexer class.

If a string does not match to any comparison then error handling will come into play.

**Error Reporting and Recovery:**

In this section it will be possible to find the test cases that are going to be in the driver in order to test the lexer and their expected output. In the case that there is an error there should be an output of either error or invalid token notification.

The test cases will be in the driver which is going to be demonstrating the lexer. We will then be able to compare the test results with the tests written here and compare if the lexer behaves properly.

The tests are not complete as the lexer program submitted is not completely function nor complete yet.

Test1.txt content:

```
integer x = 13;
/* this program is a test program */
//will the lexer put out the tokens properly?
integer y = 56;
float b = 0.13e342;
x = y + b;
write if else //here are sample of id
l
s
lek_nskasd dsak love live laugh 7_ska
```

Issues noted on this test file:

There seems to be trouble with the comment handling. The program keeps running but does not seem to see the closing comment block.

Test2.txt content:

```
A a
mi
l9 n8
9s
22
Sf
.32  .sf
4
768
432
Tests for alphanumerical:
_
Jkl
09jk0)
Tests for integers:
09u
0890
890
678756690
30673290
0273470
aslk
0934sk
Tests for floats:
12.0
12.0354
12.0.
```

12.02213.

12…

12.2156-e13

12.0e1565

12.e12

12.65e89

12.0e0215

12.sd20

Tests for fractions:

.023156

0.2130

.1235

.sda

Sads.123

.55

.5

.98

.0

.0230

Tests for punctuation:

.

\ :

< dfev >

< >

! ?

Asd.

.asd

:  ;

Tests for comments:

/*

*/

//

Issues noted on this file:

There is an issue with the float type. It is not recognized properly (output as integer punctuation integer) and I get an exception with the _ character instead of an error.

**Use of Tools:**

Programming language: Java

Programming environment: Eclipse

Version Control: Git

Documentation: Word

Regex, NFA and DFA : JFLAP

Libraries: import java.util.ArrayList;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

SEETING UP THE TOOLS:

JFLAP:

It is only needed to download the program from this link http://www.jflap.org/ and then, when the program is running, choose the finite automaton format. It will display the tools in order to build one. It is also possible to enter a RegEx and the program will convert it into an NFA. However, conversion to DFA has to be done manually.

Eclipse:

Eclipse is on all the Engineering and Computer Science computer laboratories. I used the default settings of Eclipse. My version is the latest version of both java jdk and eclipse.

GitHub:

I created a repository and then imported/exported my program and documentation as needed from this repository.

Libraries:

I used the Arraylist in order to store the Tokens in an orderly fashion. It is not the hybrid method seen in class but due to lack of time and intense computer issues, I went with the most time efficient in terms of coding wise. I plan on improving the lexer for the next assignment.

The pattern and matcher libraries have been used mostly for the smaller regular expressions and for the base cases (letters, digits, nonzero, alphanum, keywords) as this library allows to easily manage regular expressions and does the matching according to the input. The code in the lexer only needs to verify for words (guided by white spaces) and then take the word or character as the input for the matcher and determine if there is a match. It makes some of the processes less intense code wise.

REASONS OF CHOICE:

The main reasons for the tools used in this assignment are that they are the tools I am the most familiar with. Having entered the class on the last possible date, I had to make sure I understood the past two weeks' worth of material and completed the assignment in one week while handling other classes and assignment. If there are better tools for the following assignments, I will take the time to look into it now that I have caught up with the material.

Java is the language I am most familiar with and Eclipse is my favorite IDE to code in Java. As for the DFA's transition model on numerical format, I used JFLAP, a software I had to use by demand of my professor in COMP 335. I find the resulting graphs are very clear and clean. It is also a very versatile and easy to use tool which allows to work quickly. The cons of using JFLAP is that there are a lot of steps which I have to do manually, which increases the chances of errors. However, for efficiency and productivity, I used tools which were more familiar to me.

In the next assignment, it would be good to use AtoCC. It allows to double check the work that is being made. It also auto creates the transition tables and makes the automata related task faster. However in the current assignment I chose to go with a software that I already knew the way it worked even if it meant more manual work. I downloaded the program but I was having a hard time to familiarize myself with it and I mostly wanted to have a functioning lexer for the assignment even if there are more chances of errors in my regular expressions and DFA.

Another tool that could be used in the next assignments is ShareLatex. It can allow for very professional formatting and the AtoCC has functionality with it in order to insert the images of the graphs. However, using ShareLatex is very time consuming compared to Word and as such this will be a deciding factor in which tool I will use for documentation in the next assignments. For the IDE, I will keep using Eclipse unless there is a more recommended IDE by the professor or the TA as we are building on top of each assignment. I will keep a familiar IDE in order to be more efficient when coding.