Maira Malhi (40128269)
Assignment 1 - COMP 442

Design

The lexical analyzer consists of reading code, to be compiled, char by char in order to produce a sequence of tokens that are then identified and validated depending on the language expressed.Thus, the purpose of a lexical analyzer is the first step in allowing a fully designed compiler to read code. This language would consist of different cases that can be arranged into regular expressions.

I organized my code with the following: the Lexer class, within its constructor, is able to fill up an arrayList of Token objects and catch errors as it reads the file (code to compile). After the Lexer has been built, the driver or any class can use its “getNextToken()” method to extract the next generated token, in order, based on what was read from the file. Token’s have a lexeme (the String value of the actual token), a type (selected from a defined list stored in an enum) and position (simply an object that holds the row line for the token).

Regular Expressions

Definition of Space

x	$Any\ character\ in\ the\ ASCII\ Table$
l	$Letter\ in\ the\ alphabet\ a\ to\ z\ \ or\ A\ to\ Z(UPPER\ or\ lower\ case)$
d	$Any\ Digit\ 0\ -\ 9$
0	$Zero\ ‘0’\ character$
n	$Natural\ Numbers\ (excluding\ zero)$
p	$Positive\ ‘\ +\ ’\ character$
m	$Negative\ ‘\ -\ ’\ character$
a	$Asterisk\ ‘\ *\ ’\ character$
r	$Point\ ‘.’\ character$
$\_$	$Underscore\ \ character$
$reservedword$	$identify\ a\ reserved\ word$
$operator$	$identify\ an\ operator$
$punctuation$	$identify\ an\ punctuation$
y	$roundbracketopen$
z	$roundbracketclosed$
$punctuation$	$identify\ an\ punctuation$

Before designing the entire expression, I decided to break the regular expression into different cases, which involve the different atomic lexical elements as expressions and we can check if they are reserved words, and add the defined operators / punctuation.

ID $(l\,(l \mid (d \mid \_)\,*))$

$SPECIALCASES$ (integer | float | void | class | self | isa |while |if | then |else |read |write | return |localvar |constructor |attribute |function |public |private |and |or |not| == |p| <> |m| <| a| >| / |<=| =| >=| y | { | [|z | } |] p. | , | ; | : |:: |=>)

$FLOAT$ $((((n(d)\ *)|0)(.\,(d)\ *\ n)|(.\,0)(e(p|m)?\,(((n(d)\ *)|0)))?))$

$INTEGER$ $(n(d)\ *)$

$INLINECOMMENT$ $(//x\ *)$

$BLOCKCOMMENT$ $(/a(x)\ *\ /a)$

All together, I can define my language as the following regular expression:

$(SPECIALCASES\ * \mid ID\ * \mid FLOAT\ * \mid INTEGER\ * \mid\ *\ INLINECOMMENT \mid *\ BLOCKCOMMENT)$

(l (l | (d | _)*))*|(s)*| ((((n(d)*)|0)(.(d)*n)|(.0)(e(p|m)?(((n(d)*)|0)))?))*| (n(d)*)*|(/a(x)*/a) *

Note: s is used to replace every case of special case

Essentially, this expression in English would say that I can have any one of these valid tokens multiple times and in any order, which makes sense as the lexical analyser will asses tokens (and later on, with these tokens, the meaning of each line / method /class will be assessed).

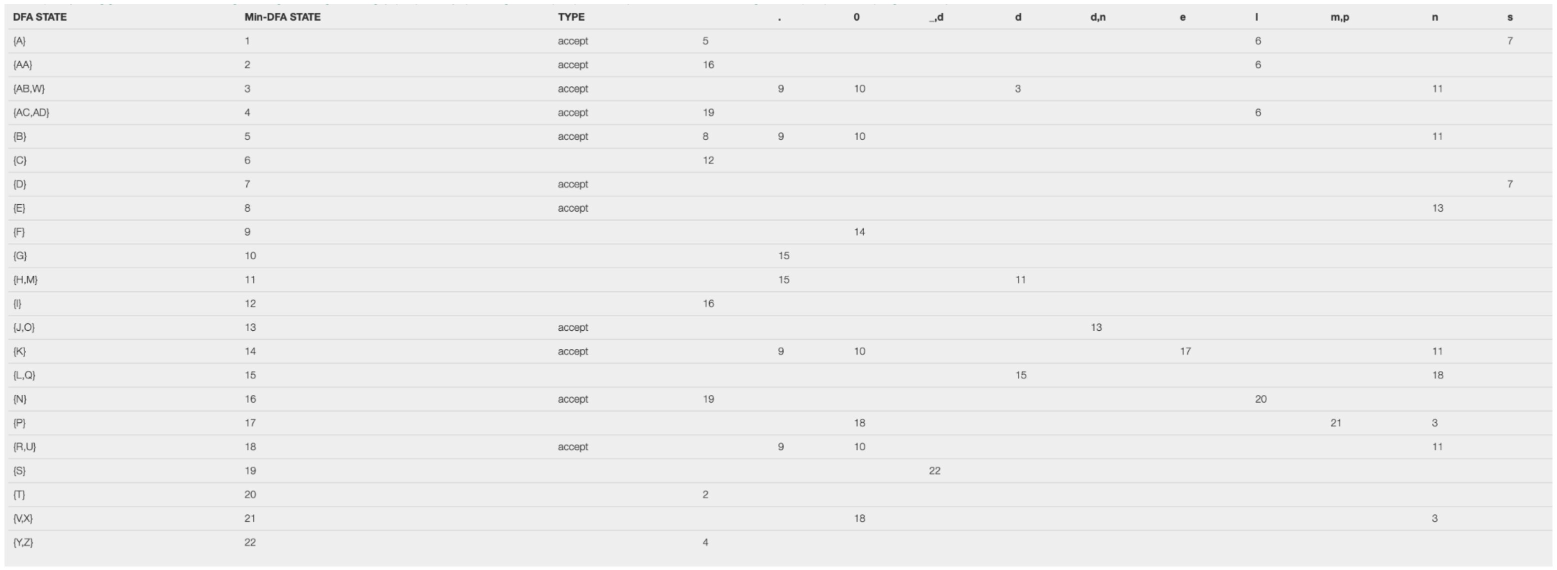
For interactive view of my DFA: (Figures also screenshotted below)
https://cyberzhg.github.io/toolbox/nfa2dfa?regex=KGwgKCBsIHwgKCBkIHwgXyApKikpKnwocykqfCAoKCgobihkKSopfDApKC4oZCkqbil8KC4wKShlKHB8bSk/KCgobihkKSopfDApKSk/KSkqfCAgKG4oZCkqKSo=

Tools

Regarding my tools I used the following:

Java - I chose this as a high level object oriented programming language, it is very versatile and can easily identify different data types (i.e. primitives, reference types are all clearly defined, no ambiguity)

IntelliJ IDE - Very easy to debug and run my code with an integrated terminal . I would like to set up a batch file (similar to the .bat on windows) in order to run all my test cases in the future as they get more complex inputs; however, as i have Macbook, I am doing more research on how to create this (IntelliJ also has built in plugins to use tools like JUnit and other test case programs that makes my coding quick).



DFA STATE	Min-DFA STATE	TYPE		.	0	_,d	d	d,n	e	l	m,p	n	s
{A}	1	accept	5							6			7
{AA}	2	accept	16							6			
{AB,W}	3	accept		9	10		3					11	
{AC,AD}	4	accept	19							6			
{B}	5	accept	8	9	10							11	
{C}	6		12										
{D}	7	accept											7
{E}	8	accept										13	
{F}	9				14								
{G}	10			15									
{H,M}	11			15			11						
{I}	12		16										
{J,O}	13	accept						13					
{K}	14	accept		9	10				17			11	
{L,Q}	15						15					18	
{N}	16	accept	19							20			
{P}	17				18						21	3	
{R,U}	18	accept		9	10							11	
{S}	19					22							
{T}	20		2										
{V,X}	21				18							3	
{Y,Z}	22		4										

Figure 1 - Table of DFA States

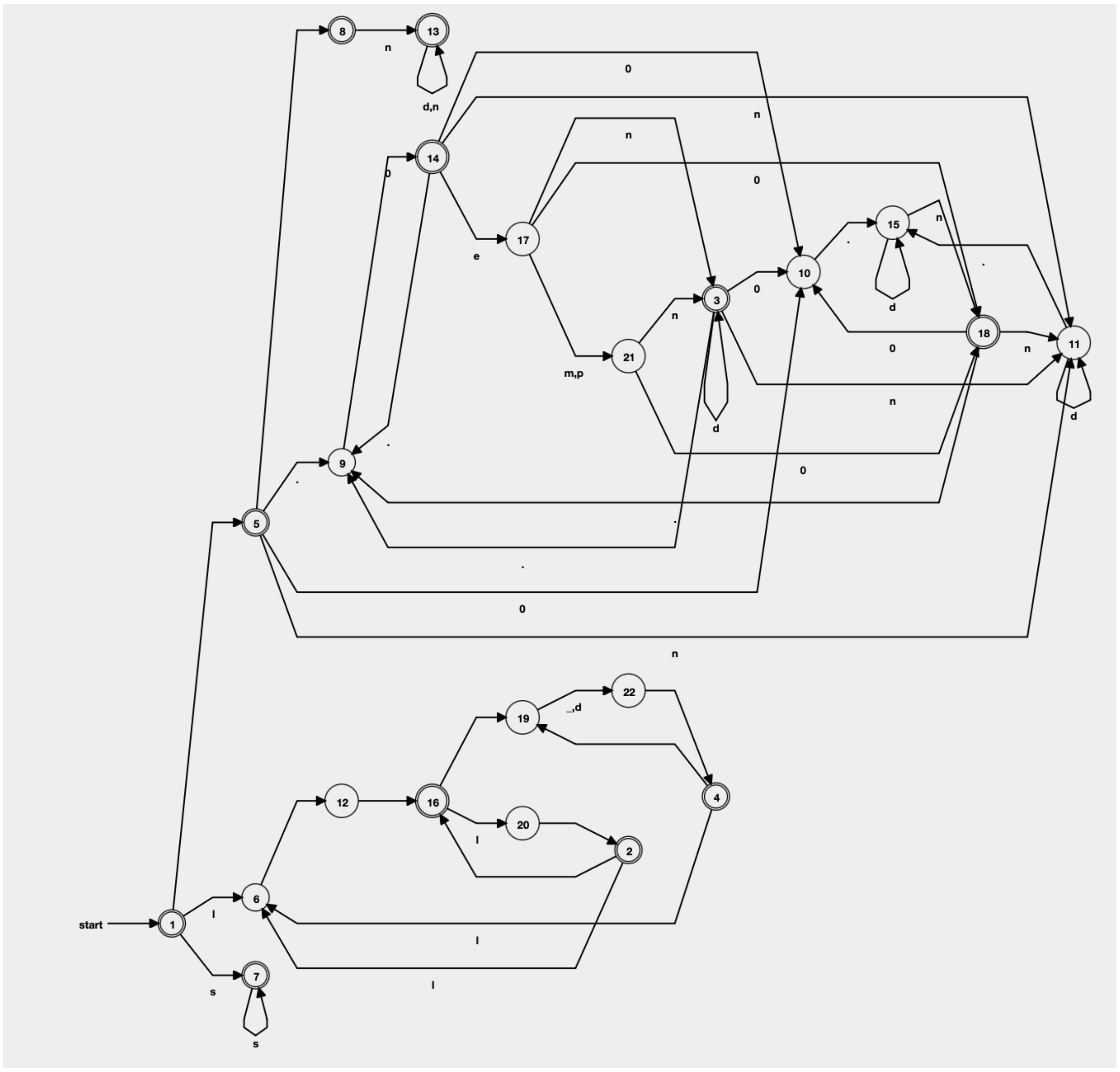


Figure 2 - DFA States for Initial Regular Expression