COMPILER CONSTRUCTION REPORT

Ben Reynolds – 13309656

Task 1:

To Implement my Trie I used two components the first component was a Node Class. My node class is a very simple class that stores a single character, the set of nodes that the node is connected to, Whether the node is an accepting node or not and also each node as an identity.

The node class also has a number of 'getter' and 'setter' methods in order to either get or set the variables I mentioned above.

My Trie class utilizes my Node class by creating an instance of node for each character added to the Trie.

The Trie class has three main methods:

1. **CheckLetter()** – This method takes a character and a Static/Dynamic Flag. If the flag is set as dynamic, then the method will check if character passed to the method is equal to the any of the nodes that are connected to the current node and if it is then it will move the current node to that node.
   Otherwise it will add the node and add it as a connected node to the current node and move the current node to the newly added node. It will return true in either of the above cases.
   If the flag is set to static it will only check if the character passed to the method is equal to the any of the connected nodes and move current node to that node if yes.

2. **CheckWordExists() –** This method takes in a whole word as an argument and returns the ID of the accepting node of the word if the word exists in the Trie and -1 otherwise. It also makes the final node accepting if the word is accepting.

3. **ProcessWord() –** Takes a word as an argument, determines whether the word is static or dynamic based on the first letter and runs the word threw the CheckLetter() method.
4. I Also created some helper methods in my Trie class, one to print out the Trie for testing functionality of both my Trie and my Lexical Analyzer. A method to create a new highest ID for creating a new Node instance and finally a method to set the current node to the root node.

Sample output of Task1 for adding these words in this order:

1. private - > ID: 7
2. public - > ID: 12
3. protected -> ID: 19
4. static - > ID: 25
5. primary - > ID: 29
6. integer - > ID: 36
7. exception - > ID: 45
8. try - > ID: 48

Task 2:

To Implement my Lexical Analyzer, I started by writing and building a DFA class which later became my Lexer class as the more functionality I added the less of a straight forward DFA it became and I felt it warranted changing its name.

At its roots this class is a DFA. It has a starting state, accepting states, a transition function and an alphabet. I implemented the transition function using a dictionary mapping keys (states) to the input and returning a new state for that transition.

I added some extra functionality like the ability to put back a character. This is used in the case that we have an input like “node8” when the input hits 8 it’ll move from the state that deals with lowercase characters to a special state that tells the DFA to not add the 8 to the current word, check if node is accepted and if it is then add that word to the Symbol Table (Trie.)

The Lexer class reads the input file character by character and uses the whitespace characters or the special put back state to recognize when the token has ended.

The classes driver method reads threw the file calling run_char() for each character. run_char() is where a lot of the special processing takes place. i.e. this method checks if whitespace and thus end of token or calls the transition_state() method if needed, adds to the symbol table and also returns the identifier for the current token.

The massive int problem is dealt with between my transition_state() method and my check_max_int() method. It checks the length of the integer and whether the individual elements are greater then the max allowed values of the the individual numbers of the int. It’s not even close to being a pretty solution but it’s functional.

The class keeps track of a number of items e.g. the current word token so that I can add the word to the add the current word token to the Symbol Trie if the token is accepting.

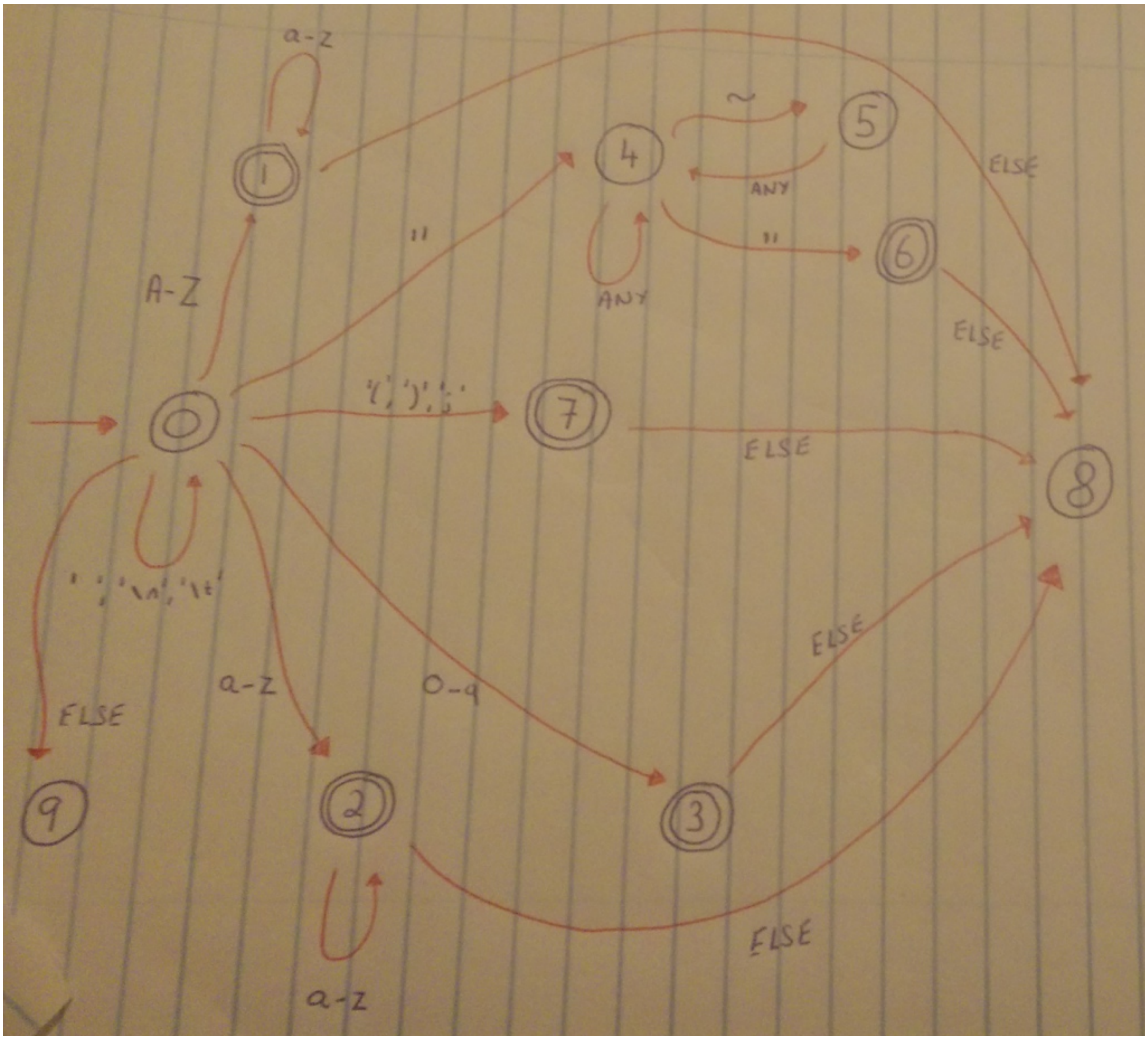
Issues:

- The ~ Character doesn’t get added in the scenario ~~

Sample Outputs: - For Given Input Files.

Input File 3:	*Input File 4:*	Input File 5:
<ID,2>	<ID,4>	<ID,5>
<ID,-1>	<ID,12>	<ID,8>
<ID,5>	<ID,-1>	<STRING,['"', 's', 't', 'r', 'i', 'n', 'g', 's', ' ', 'o', 'f', ' ', 't', 'h', 'i', 'n', 'g', 's', '"']>
<ID,8>	<ID,14>	<ID,29>
<ID,13>	<LPAR, 0 >	<ID,5>
<ID,2>	<INT,12>	<ID,8>
<ID,-1>	<ID,19>	<STRING,['"', 'c', 'h', 'i', 'c', 'k', 'e', 'n', 's', '\n', 'w', 'i', 't', 'h', '\n', 'w', 'i', 'n', 'g', 's' '"']>
<ID,5>	<ID,23>	
	<RPAR, 0 >	
	<ID,26>	

Image of DFA:



Task 3 - Flex:

For Task 3 I used Flex, I wrote a scanner.flx file to describe my grammar and then a C program to run my program on a test input.

I used regular expressions in order to describe my language in my .flx file. I have included a sample output of my c program below.

To Run flex I used installed flex and ran flex scanner.flx followed by gcc program.c lex.yy.c before finally piping my input file into my c program with command ./a.out < input.txt

Sample output for given input file:

```
<ID:  ben>
<ID:  reynolds>
<INT:  01>
<ERROR:  ?>
<INT:  000001>
<STRING:  "BenReynolds">
<LPAR:  (>
<RPAR:  )>
<SEMICOLON:  ;>
<ERROR:  ?>
```