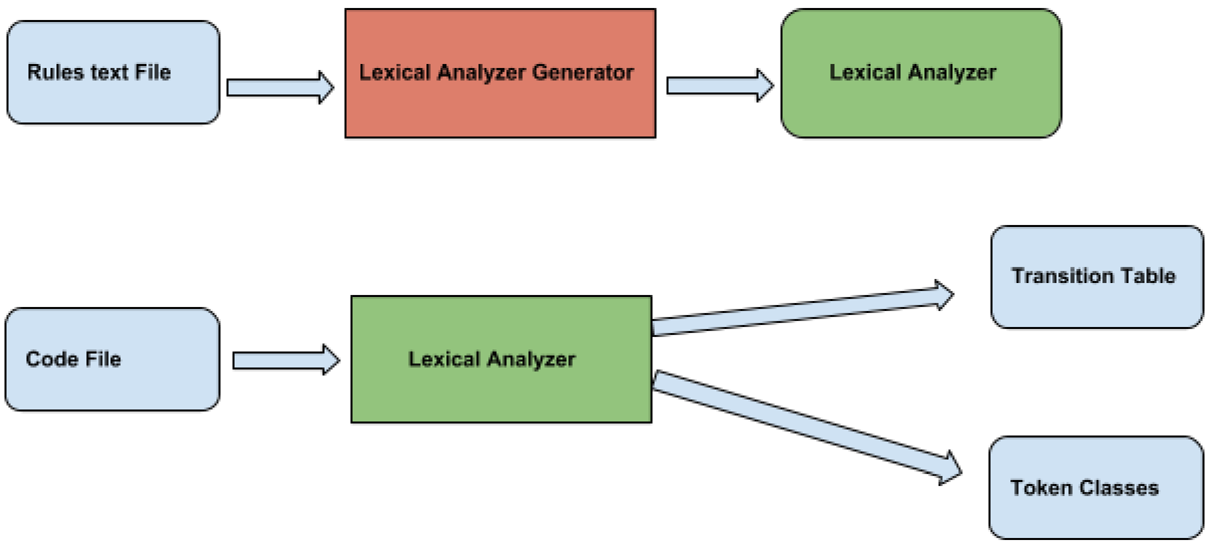
Programming Languages Translation

Project Phase 1 : Lexical Analyzer Generator

Spring 2019



Assumptions

- None but correct input by the user and havin a text editor and a python IDE.

Test Program Results

Tokens

int -> ['int', 'id']

abc123 -> ['id']

, -> [',']

count -> ['id']

, -> [',']

pass -> ['id']

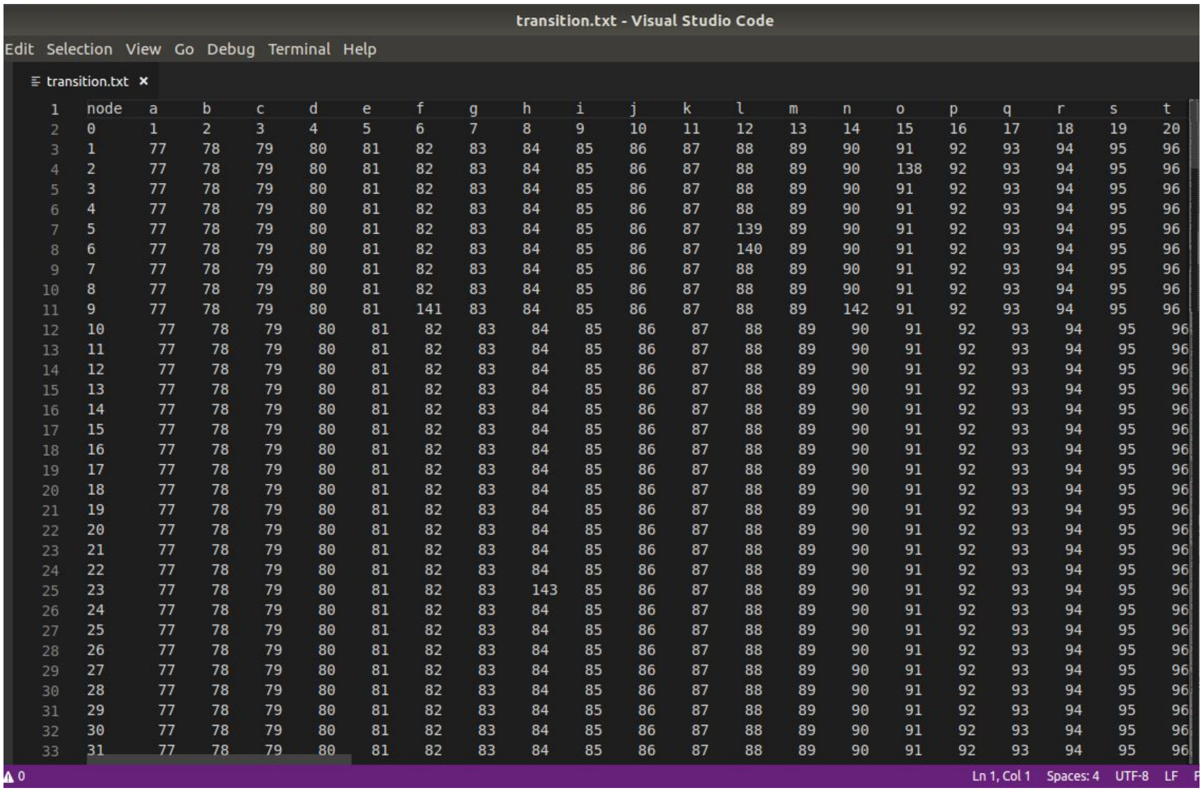
, -> [',']

Ahmed Morsy	**Id: 4448**	**Merna Zakaria**	**Id: 4106**
Louay Hesham	**Id: 3303**	**Kareem Ahmed**	**Id: 3356**

```
mnt  ->  ['id']
;  ->  [';']
while  ->  ['while', 'id']
(  ->  ['(']
pass  ->  ['id']
!=  ->  ['relop']
10  ->  [ 'digits', 'num']
)  ->  [')']
{  ->  ['{']
pass  ->  ['id']
=  ->  ['assign']
pass  ->  ['id']
+  ->  ['addop']
1  ->  [ 'digits', 'num']
;  ->  [';']
}  ->  ['}']
```

Transition Table

Full table in transition.txt file with the code package.



node	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
2	77	78	79	80	81	82	83	84	85	86	87	88	89	90	138	92	93	94	95	96
3	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
4	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
5	77	78	79	80	81	82	83	84	85	86	87	139	89	90	91	92	93	94	95	96
6	77	78	79	80	81	82	83	84	85	86	87	140	89	90	91	92	93	94	95	96
7	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
8	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
9	77	78	79	80	81	141	83	84	85	86	87	88	89	142	91	92	93	94	95	96
10	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
11	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
12	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
13	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
14	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
15	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
16	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
17	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
18	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
19	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
20	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
21	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
22	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
23	77	78	79	80	81	82	83	143	85	86	87	88	89	90	91	92	93	94	95	96
24	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
25	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
26	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
27	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
28	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
29	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
30	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
31	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96

Bonus

In flex.pdf file

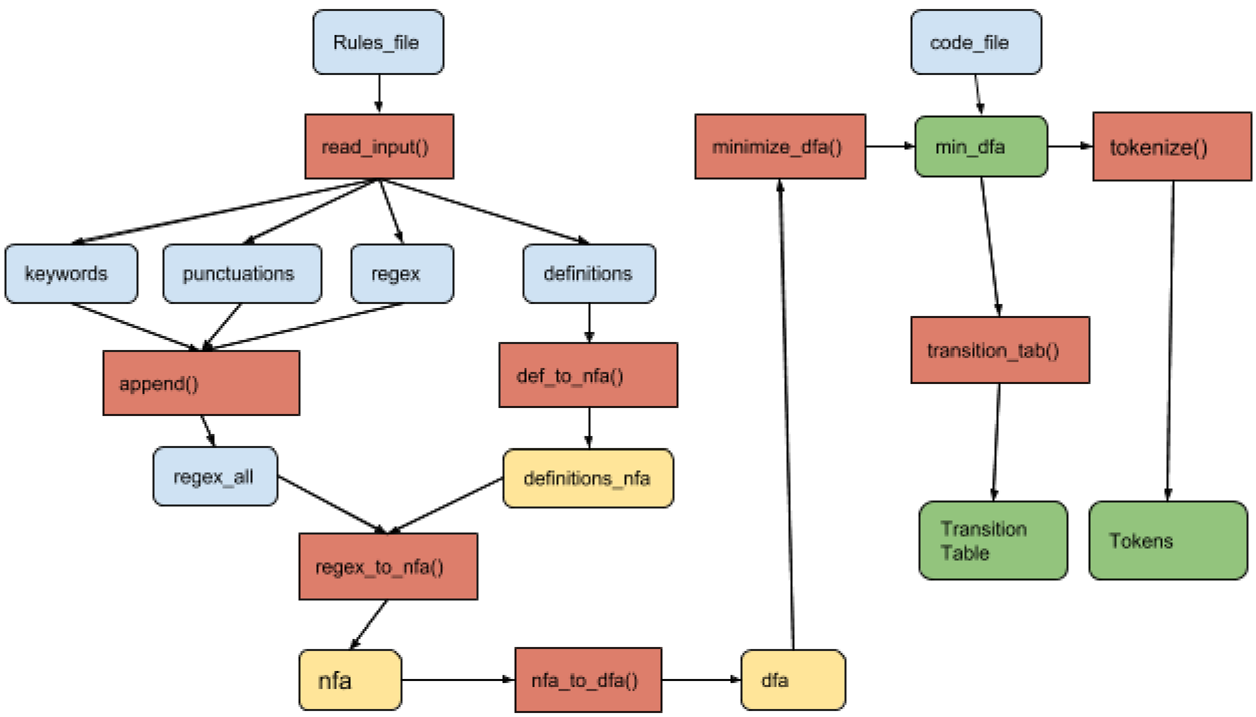
Process

Data Structures and Design Patterns

- **Graphs :**
  A data structure consisting of one or more node objects, one of which is a start node and at least one accept node.
- **Nodes :**
  A data structure representing a state, has transition characters that correspond to other destination nodes, if an input character is not in the transition characters of a node then the node cannot move to a new state.
  A node can be a start node in a graph and/or a finish node (accept state) or a dead state.
- **Node Generators (Singleton , Factory) :**
  A node generator is a singleton class and a Node factory that is used to make instances of the Node class giving each new node a new distinctive Id.

- **Python Dictionaries :**
  Built in data structures that map string keys to an object value for each.
- **Python Lists:**
  Built in data structures that are Arraylists (dynamic arrays) of Objects.
- **Python tuples:**
  Built in data structures for keeping pairs of objects.

Algorithms and Techniques



- **Overall :**
  As seen from the figure above which shows the outlines for the Lexical Analyzer generator and for using the generated Lexical Analyzer, the input is a grammar rules text file and the output is minimum DFA.
  The second input code file is then used by the generated DFA to produce the Transition table and the tokens classes one in each line corresponding to tokens in code file.
  The next sections will explain every major step and algorithms used.

- **Regular Expressions to NFA (Thompson's construction):**
  In order to convert a regular expression to NFA, the Thompson construction

algorithm is implemented in code, where a simple graph is made for each symbol in the regular expression consisting of a pair of nodes (start and end) and the transition from start node to end node is through the symbol.
Then the thompson operator rules are implemented, to cover cases of concatenation, kleen closure, or ...etc through constructing more complex graphs using functions implemented in the graph class and NFA class to make more complex combined NFAs.

For example: for concatenation of two symbols, a new graph is formed of the two graphs of the two symbols with the start node having a transition with the first symbol to another node and that other node having a transition to the accept state of the new combined graph using the second symbol.

Note that the epsilon is used in our code as “@”.

- **NFA to DFA (Subset Construction) :**
  In order to convert NFA to DFA, the subset construction is implemented in our code using queues as the main data structure with lists.

  Epsilon transitions for each node is captured and inserted in the queue and all epsilaon transitions of the state and the state itself are treated as one DFA state, if this DFA state can have multiple transitions with one character then all these transitions are treated as a new DFA state.

- **DFA Minimization :**
  DFA is minimized to one with minimal number of states using equivalence theorem

  Step 1 − All the states Q are divided in two partitions − final states and non-final states and are denoted by P0. All the states in a partition are 0th equivalent. Take a counter k and initialize it with 0.

  Step 2 − Increment k by 1. For each partition in Pk, divide the states in Pk into two partitions if they are k-distinguishable. Two states within this partition X and Y are k-distinguishable if there is an input S such that δ(X, S) and δ(Y, S) are (k-1)-distinguishable.

  Step 3 − If Pk ≠ Pk-1, repeat Step 2, otherwise go to Step 4.

Step 4 − Combine kth equivalent sets and make them the new states of the reduced DFA.

Functions

(Main program functions ordered with almost same flow of execution and almost same names)

- **read_input (text_file):**
  Reads grammar rules text file and separates identifiers, numbers, keywords, operators and punctuation symbols.
  <u>Returns:</u>
  keywords : a python list of keywords in input grammar
  punctuations : a python list of punctuations
  regex : a python dictionary of regular expressions, with keys of expression names and values of regex string , complex regular definitons contaning operations are assumed to be regex
  definitions: a python dictionary of basic regular definitions, with keys of expression names and values of definitions string

- **definitions_to_nfa(definitions) :**
  converts dictionary of definitions (key is definition name and value is def string) into a dictionary of NFAs for those definitions.
  <u>Returns:</u>
  definitons_nfas : a dicitonary with keys of definitons names and values of NFAs corresponding to their strings from input.

- **append(keywords,punctuations,regex) :**
  concatenate everything other than definitions to be converted to NFAs at once.
  <u>Returns:</u>
  regex_all : a list of tuples, first element is value and second one is name of element/regex to be converted

- **regex_to_nfa(regex_all,definitions) :**
  converts a list of tuples of regular expressions, keywords and punctuations into a list of graph objects which represent corresponding NFAs given the dictionary of regular definitions and their NFAs.

Returns:
nfas: a python list of graph objects for all regular expressions, keywords and punctuations

- **Combine_nfas(nfas):**
  concatenate everything other than definitions to be converted to NFAs at once.
  Returns:
  nfa: a python list of graph objects for all regular expressions, keywords and punctuations

- **DFA Minimization :**
  Minimize input DFA using Equivalence theorem and returns a DFA with minimal number of states

Code Structure

- Files : one for each class and one per big function, called when used
- NFA and DFA files encapsulate NFA and DFA operations

Usage Instructions and dependencies

- Only Python 3.6 or higher with no external libraries
- Put input rules in grammar.txt
- Put code in code.txt or input string in main.py file
- Run main.py