ECOTE – Crating DFA form Regular Expression

Semester: 20L

Author: Jakub Szumski 295432

Subject: Program reading regular expressions, constructing NFA using Thompson algorithm, converting NFA into DFA and validating input strings generated by this expression (working on DFA).

I. General overview and assumptions

Main topic of my assignment is to create a program that creates ε-NFA form a given regular expression using Thomson algorithm. Then it constructs DFA from a given ε-NFA and validates the input strings. One of the assumptions is that I'll use the English letters as an input alphabet. Furthermore with regards to the regular expression's symbols I'll use "*" as a symbol of Kleene's closure, "|" for union, concatenation is unmarked, thus I'll assume "\x08" (binary code to represent backspace character) as it's representation in my symbol table.

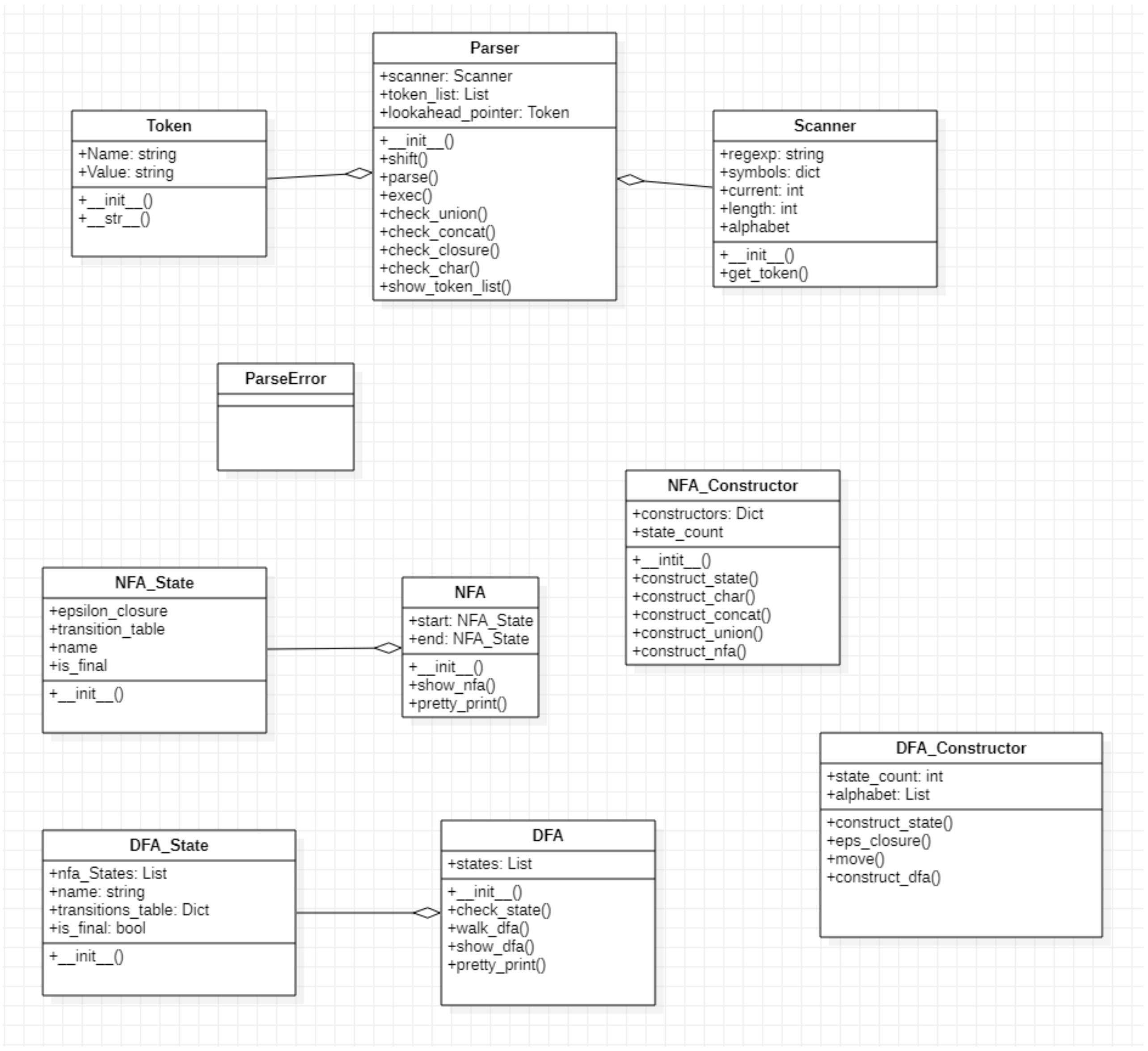
II. Functional requirements

There are two inputs in my program. First one is the regular expression that uses symbols like: (*, |, /x08,) to represent all necessary conjunctions. The alphabet I've used is English alphabet, thus any attempt to introduce other characters or symbols will result in ParseException being thrown. Second one is the input string (or many input strings) that perform a walk over the DFA constructed form ε-NFA which we've obtained form Thompson algorithm form a given regular expression.

III. Implementation

Algorithm itself consists of several parts. In the first phase we separate characters into a groups of tokens. There are two type of tokens: letter which stands for input alphabet and symbol which represents special symbols used to describe regular expression. Each token has two fields: name and value. Scanner scans through a given regular expression and every time it's required it returns the next token on the input. Main part of this phase is Parser which creates token table that represents given regexp. This part is crucial because NFA and DFA will be constructed based on this table. Phase two creates the NFA using appropriate constructors defined in NFA_constructor class. The last part of this algorithm is to create DFA form a given ε-NFA which we do by defining ε-closure and move function. In the end we validate our input string with walk() function to check if given regexp is accepting our input string.

IV. Program Architecture



As mentioned above program is constructed in 3 layers. Each table represents different class.

Parsing

Token	
name	*Name of the token eg. CHAR or LEFT_PAREN*
value	*Value of the token eg. a or (*
__init__	**Init method**
__str__	**Conjunction of name and value of a token eg. CHAR: a**

Scanner	
regexp	*Holds the input pattern of the regular expression*
symbols	*Symbol table defined for regular expressions*
current	*Points to current position in regexp*
length	*Total length of regular expression*
alphabet	*Defined alphabet of input words and regexps*
get_token	**Returns next token in line**
show_pattern	**Presents regular expression**

Parser	
scanner	*Holds the scanner that have regular expression*
token_list	*Stores created list of tokens form scanner's regexp*
lookahead_pointer	*Points to the next regex*
__init__(scanner)	**Initialize class and set lookahead pointer to first char in regexp using get_token method**
shift(name)	**Takes the supposed name of the token and sets lookahead pointer to next in line**
parse	**Starts the "recursion" of the algorithm that creates token list and returns it**
check_union	**Checks if the union is present. If so then memoraize it and dive deeper into regexp**
check_concat	**Checks if characters are concatenated**
check_closure	**Checks for Kleen's closure**
check_char	**Searches for letter or left parentheesis**
show_token_list	**Prints token list**

It's worth explaining here how the parser structure works. If we want to begging parsing process we have to call parse() function. It starts the entire cycle of creating token list in something similar to Reverse Polish Notation. Firstly diving down into functions the character or parenthesis is checked. If found then we add it to token list. First return from the check_char function looks for kleen's closure appering after the character or right parenthesis. If found then we add it after character in token list if not then we proceed with return. Next one in line is concatenation. Last thing we look for is union character. If we won't find any of the mentioned above we return the token list. If we find some appropriate symbol we start our dive again.

ε-NFA

NFA State	
epsilon closure	*List of state that have epsilon transition*
transition table	*Dictionary with key being letter and value is nfa state*
name	*Name of the state eg. s2*
is_final	*Flag for marking the final state*
__init__ (name)	**Initialize nfa_state with given name**

NFA	
Start	*Start state reference*
End	*End State reference*
__init__ (start, end)	**Initialize NFA with given start and end reference**
show_nfa	**Prints the contets of NFA**
pretty print	**Recursive function for printing**

NFA Constructor	
constructors	*Dictionary holding all functions under the key of token names*
state_count	*Counter of states in constructed NFA*
nfa_stack	**A stack holding all constructed NFA**
construct_state	**Returns constructed state with name taken form state count eg. s6**
construct_char	**Creates two states adds transition to start and pushes created nfa onto the stack**
construc_concat	**Pops two nfa's and adds epsilon transition between end of the first and start of the second**
construct_union	**Pop two nfa's, create two states, add eps transiton from state1 to nfa's start and connect end's of nfa's with second state. Construct nfa from given states and push it onto the stack.**
construct_star	**Pop nfa form stack create two states and add appropriate eps transitions, create nfa and push it onto the stack**
construct_nfa	**Create nfa based on the token list, assert the stack is of length 1 if so pop first element**

DFA

DFA State	
nfa states	*List of nfa states inside the dfa state*
transition table	*Dictionary with key being letter and value is dfa state*
name	*Name of the state eg. s2*
is_final	*Flag for marking the final state*
__init__ (name)	**Initialize dfa_state with given name**

DFA	
states	*List of DFA states*
__init__	**Initialize DFA**
check_state(given_states)	**Take list of NFA states and return corresponding DFA state if it exists**
walk_dfa(word)	**Perform a walk over DFA with a given word if it works on DFA then return True**
show_dfa	**Print DFA**
pretty_print	**Recursive Printing**

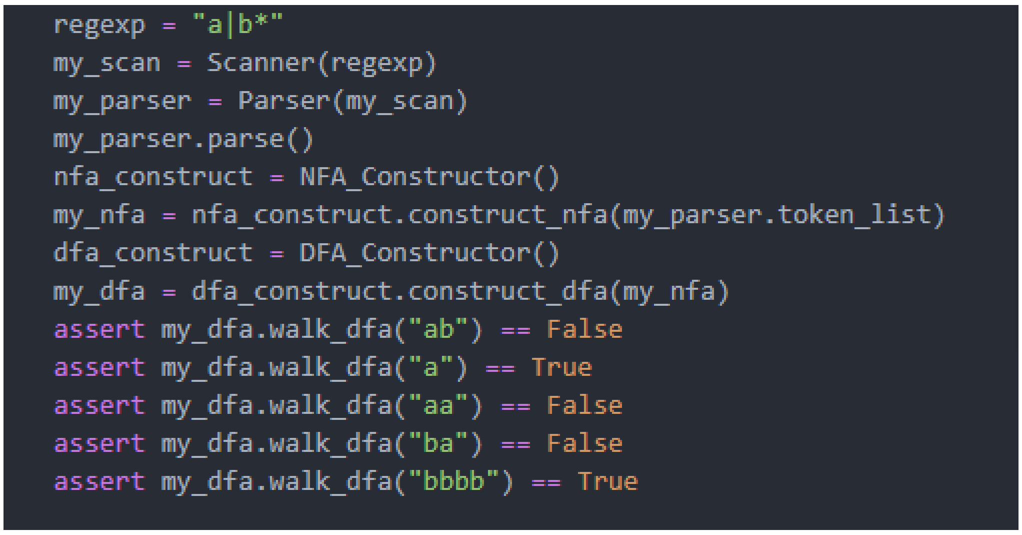
DFA Constructor	
alphabet	*List of all characters defined in alphabet*
state_count	*Counter of states in constructed NFA*
__init__	**Initialize DFA constructor**
construct_state	**Returns constructed state with name taken form state count eg. DFA_s6**
eps_closure(states)	**Perform epsilon closure on the list of states**
move	**With a given state and letter performe move operation**
construct_dfa	**Create DFA and DFA state, perform eps closure on it and add it to list of states. For every state in list of states, perform move function for entire alphabet and then eps closure on this move function result. If the states with the resulting list exists add transition if not create such state and add transition.**

V. Execution of the program

To execute this program we need to do certain steps:

1) Define our regexp
2) Create scanner with a given regexp
3) Create parser with a defined scanner
4) Construct NFA form a token list given from the parser
5) Construct DFA from a nfa constructed by NFA Constructor
6) We can now perform a walk over DFA with given string

Exemplary code:



```
regexp = "a|b*"
my_scan = Scanner(regexp)
my_parser = Parser(my_scan)
my_parser.parse()
nfa_construct = NFA_Constructor()
my_nfa = nfa_construct.construct_nfa(my_parser.token_list)
dfa_construct = DFA_Constructor()
my_dfa = dfa_construct.construct_dfa(my_nfa)
assert my_dfa.walk_dfa("ab") == False
assert my_dfa.walk_dfa("a") == True
assert my_dfa.walk_dfa("aa") == False
assert my_dfa.walk_dfa("ba") == False
assert my_dfa.walk_dfa("bbbb") == True
```

VI. Functional test cases

- Regex: a|b* (Expected patterns: a, ε, b, bb, bbb,bbbb) Input: ab, a, aa, ba, bbbb.
  Expected output: (False, True, True, False, True)
- Regex: (ab|aa)* (Expected patterns: ε, ab, abab, aa, aaaa) Input: ababab, aaa, a, aa, bb
  Expected output: (True, False, False, True, False)
- Regex: (cd* | bha)* jk (Expected patterns: cdjk, cdcdjk, bhabhajk, jk) Input: k, mf, jk, e, t
  Expected output: (False, False, True, False, False)
- Regex: ytut (a | g)* lk (Expected patterns: ytutalk, ytutglk, ytutaaaa,lk) Input: a, ytu, ytutalk
  Expected output: (False, False, True)
- Regex: 01(‘=34567) Expected Result: ParseError detection!
- Regex: (*) Expected pattern: ε Input: a, 4, 2656, Owsekb, ε
  Expected output: (False, False, False, True)