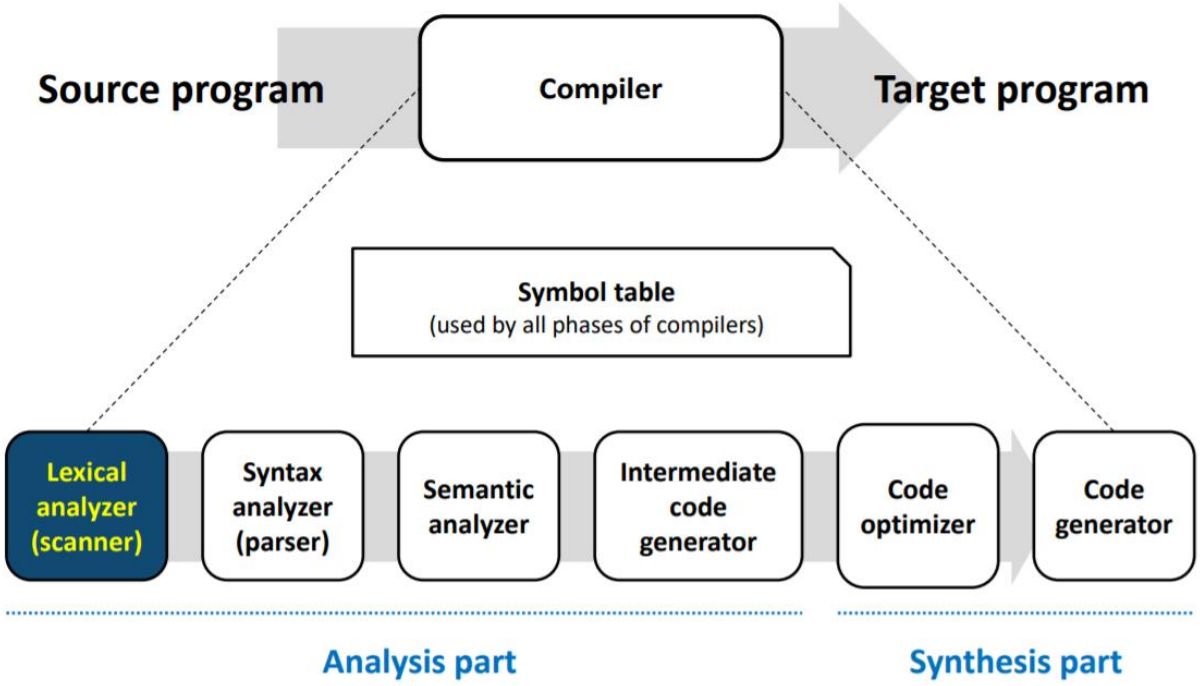
Léo Paol – 50181573
Christophe Mei – 50181587

Compiler

Term Project #1

Implementation of a lexical analyzer



Léo Paol – 50181573
Christophe Mei – 50181587

I. Definition of tokens and regular expressions

Token name	Value examples	Regular Expression
VAR	int, INT	`int|INT`
CHAR	char, CHAR	`char|CHAR`
INTVAL	0, -1, 10, -20, 999	`0|((-|ε)nzdigit digit*)`
CHARVAL	"I am 20 years old"	`"(digit|letter|blank)*"`
ID	func1, i, foo	`letter(letter|digit)*`
IF	if, IF	`if|IF`
ELSE	else, ELSE	`else|ELSE`
WHILE	while, WHILE	`while|WHILE`
RETURN	return, RETURN	`return|RETURN`
OP	+, -, *, /	`+|-|*|/`
ASSIGN	=	`=`
COMP	<, >, ==, !=, <=, >=	`((<|>|=|!)=)|<|>`
TERM	;	`;`
LSCOPE	{	`{`
RSCOPE	}	`}`
LPAREN	(	`(`
RPAREN	)	`)`
COMMA	,	`,`
WSPACE	\t, \n, blank	`(\t|\n|blank)+`

Alphabets

digit = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

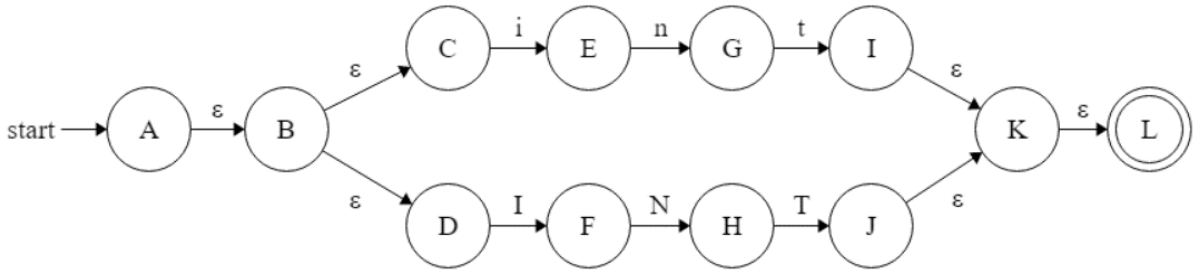
nzdigit = {1, 2, 3, 4, 5, 6, 7, 8, 9}

letter = {a, b, c, d, e, f, g, h, I, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z,
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}

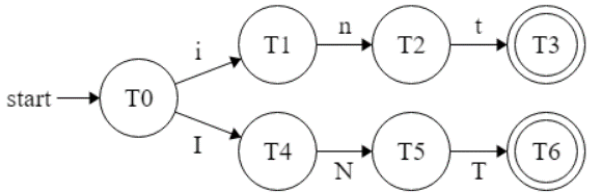
Léo Paol – 50181573
Christophe Mei – 50181587

II. NFA & DFA

INT

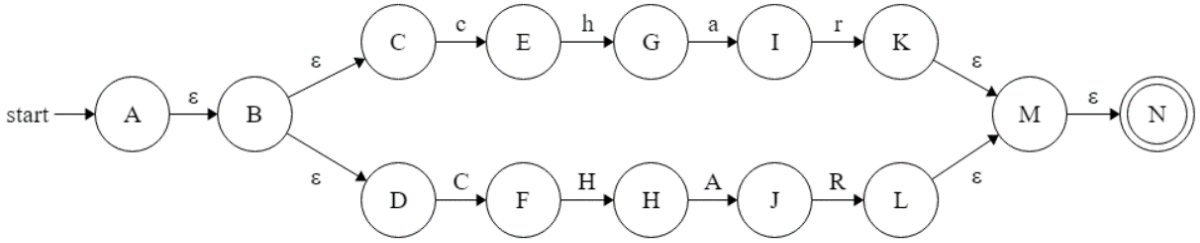


	i	n	t	I	N	T
T0	T1			T4		
T1		T2				
T2			T3			
T3						
T4					T5	
T5						T6
T6						

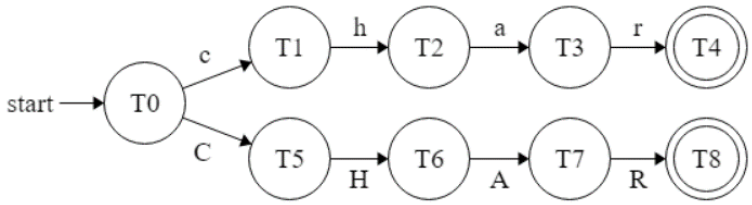


Léo Paol – 50181573
Christophe Mei – 50181587

CHAR

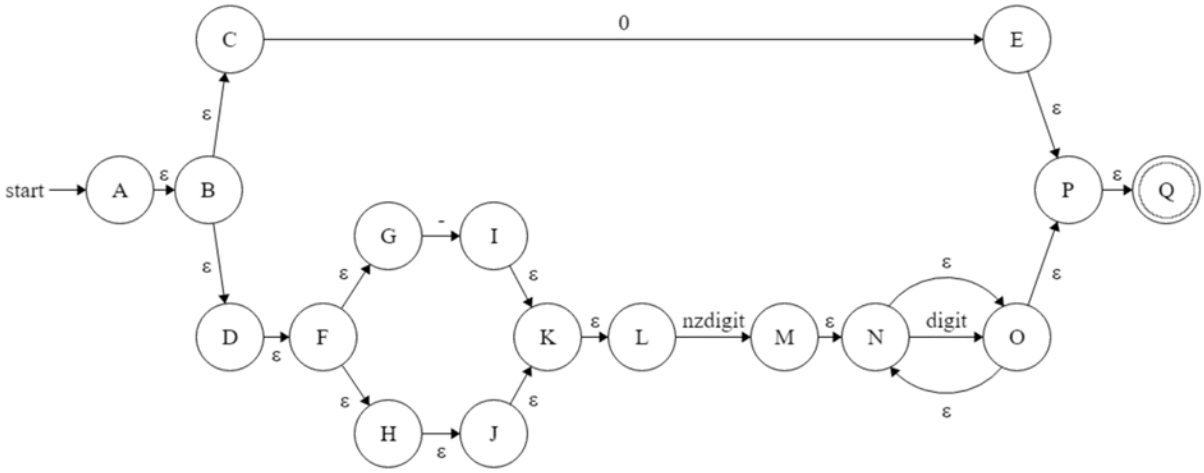


	c	h	a	r	C	H	A	R
T0	T1				T5			
T1		T2						
T2			T3					
T3				T4				
T4								
T5						T6		
T6							T7	
T7								T8
T8								

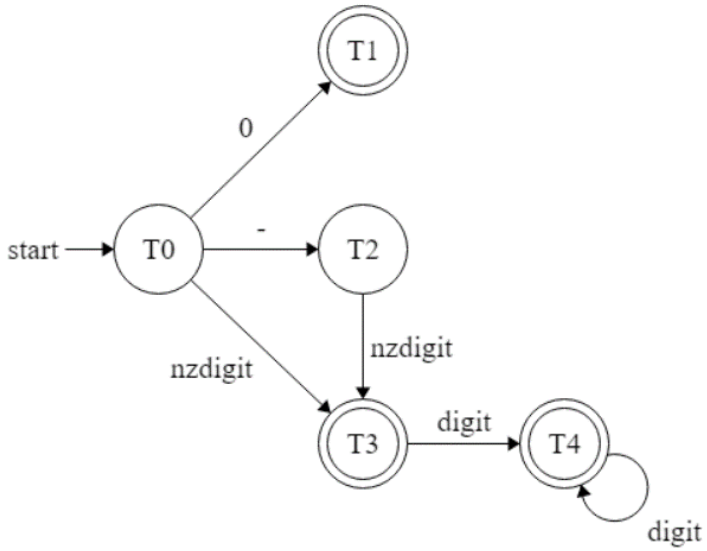


Léo Paol – 50181573
Christophe Mei – 50181587

INTVAL

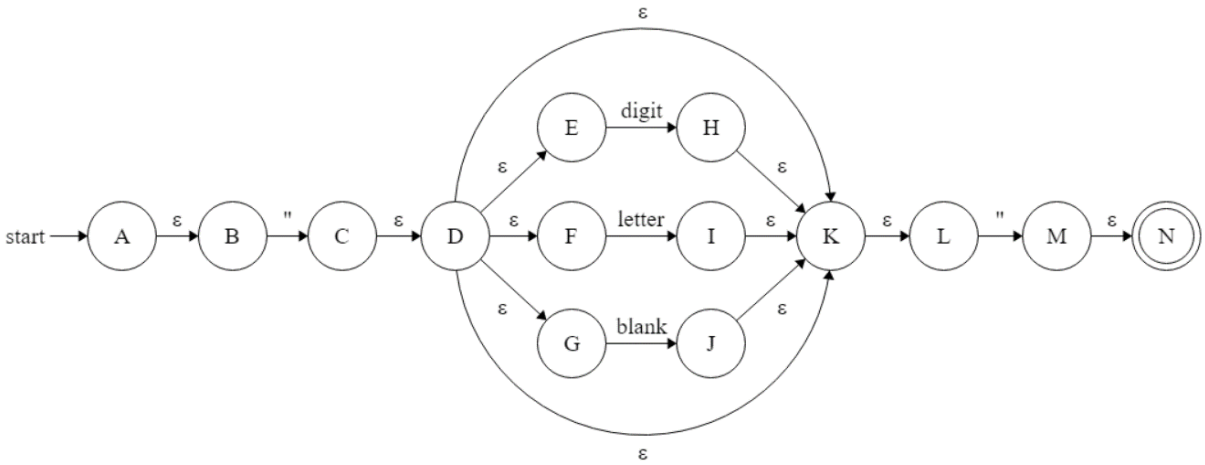


	0	-	nzdigit	digit
T0	T1	T2	T3	
T1				
T2			T3	
T3				T4
T4				T4

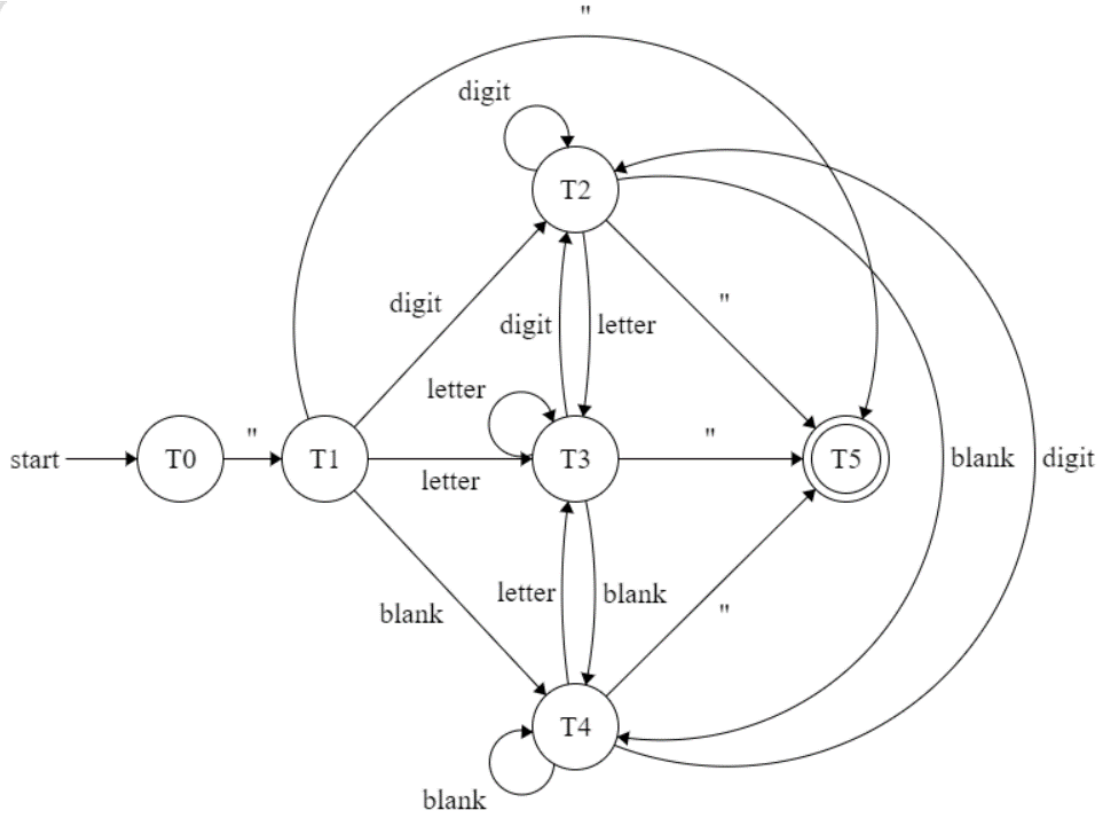


Léo Paol – 50181573
Christophe Mei – 50181587

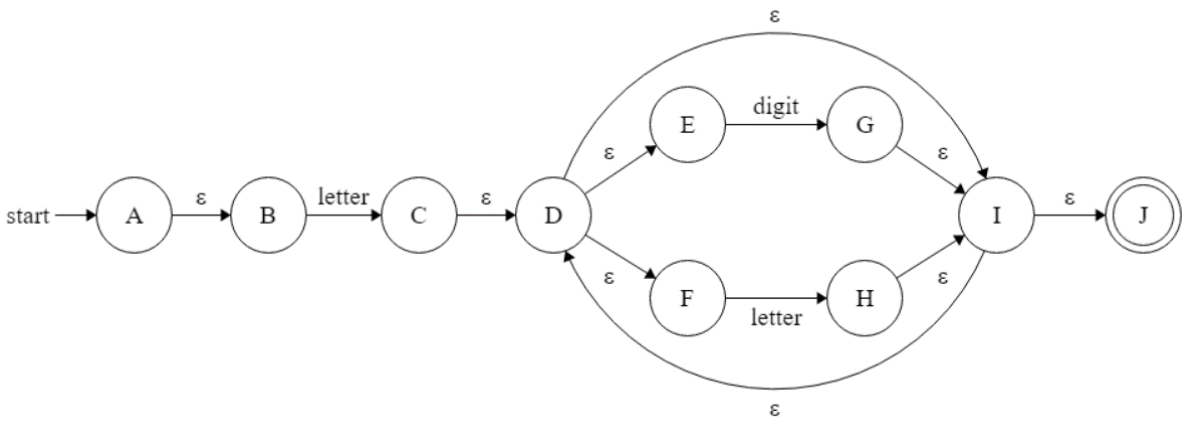
CHARVAL



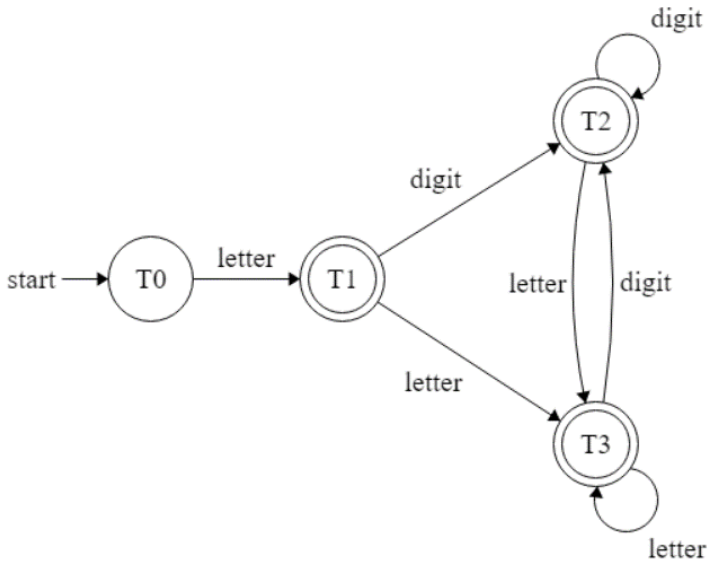
	"	digit	letter	blank
T0	T1			
T1	T5	T2	T3	T4
T2	T5	T2	T3	T4
T3	T5	T2	T3	T4
T4	T5	T2	T3	T4
T5				



ID

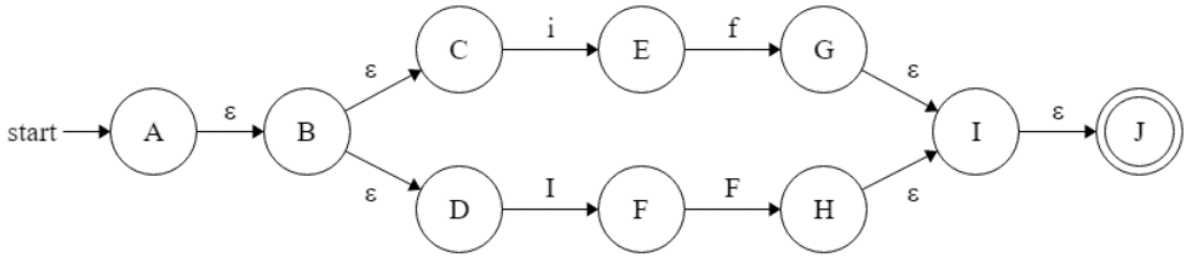


	letter	digit
T0	T1	
T1	T3	T2
T2	T3	T2
T3	T3	T2

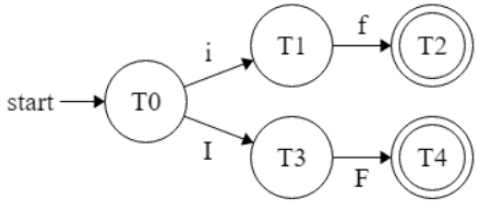


Léo Paol – 50181573
Christophe Mei – 50181587

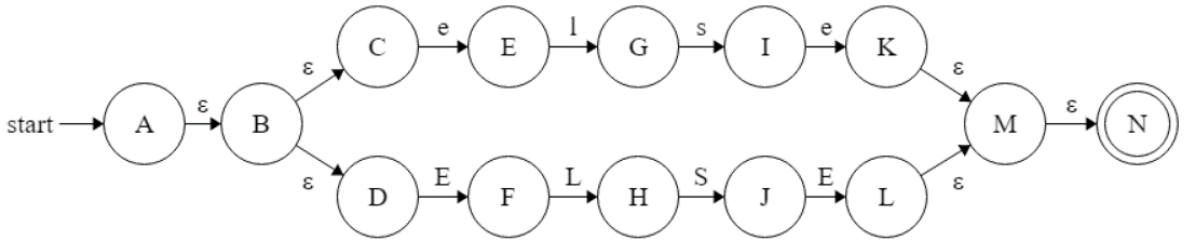
IF



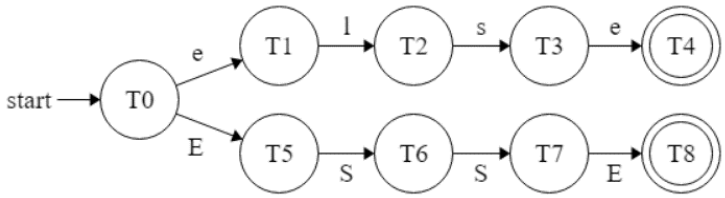
	i	f	I	F
T0	T1		T3	
T1		T2		
T2				
T3				T4
T4				



ELSE

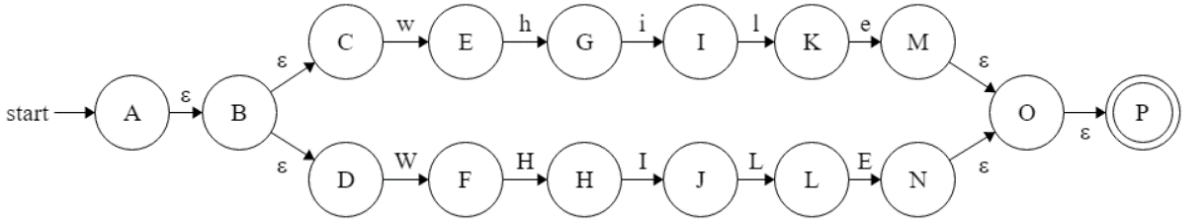


	e	l	s	e	E	L	S	E
T0	T1				T5			
T1		T2						
T2			T3					
T3				T4				
T4								
T5						T6		
T6							T7	
T7								T8
T8								

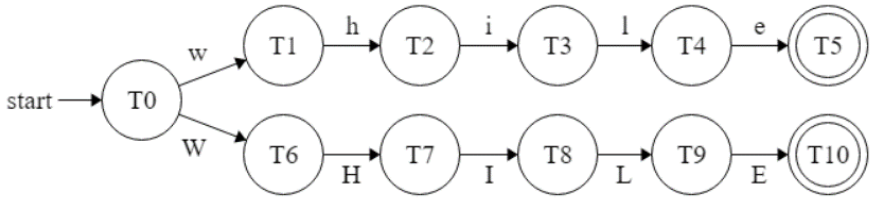


Léo Paol – 50181573
Christophe Mei – 50181587

WHILE

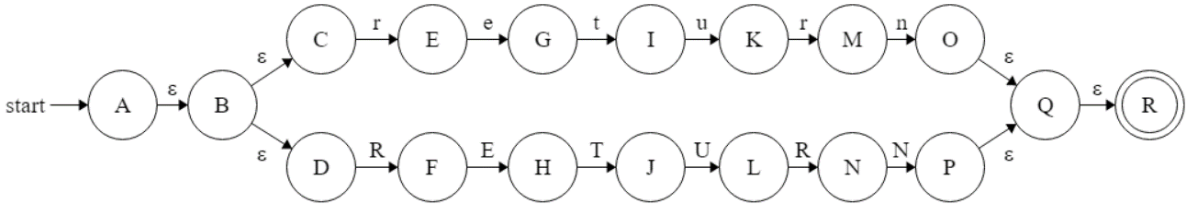


	w	h	i	l	e	W	H	I	L	E
T0	T1					T6				
T1		T2								
T2			T3							
T3				T4						
T4					T5					
T5										
T6							T7			
T7								T8		
T8									T9	
T9										T10
T10										

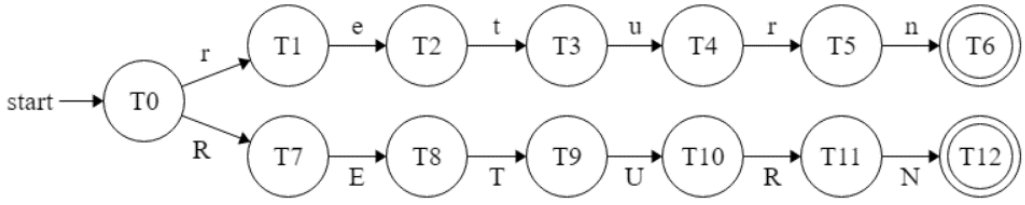


Léo Paol – 50181573
Christophe Mei – 50181587

RETURN

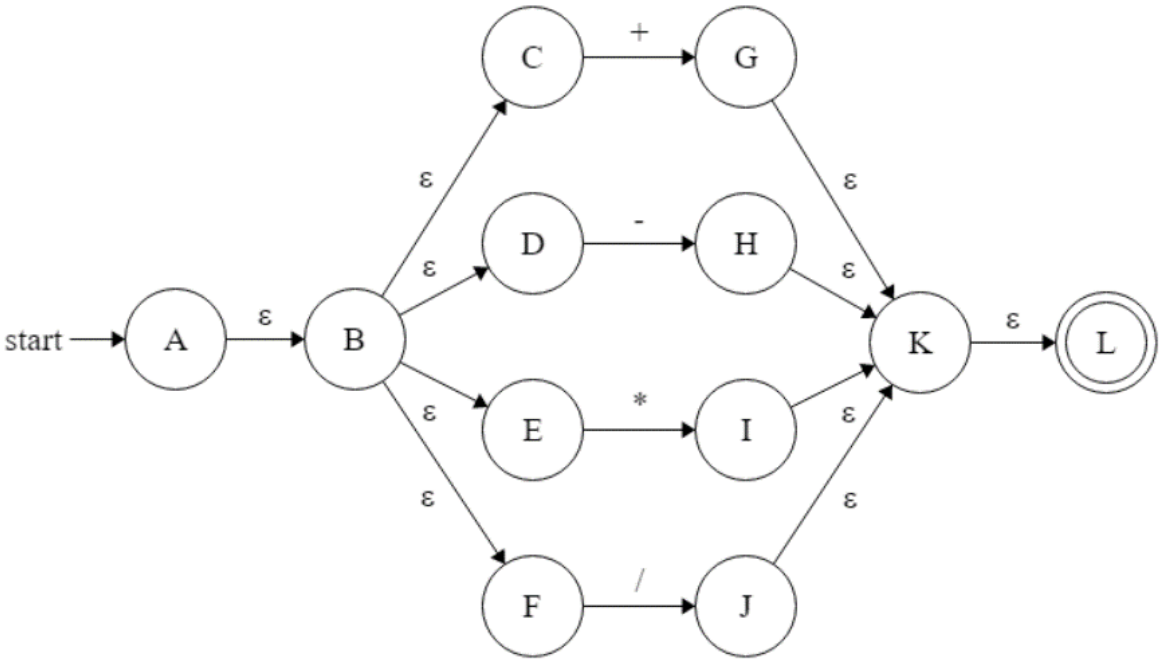


	r	e	t	u	r	n	R	E	T	U	R	N
T0	T1						T7					
T1		T2										
T2			T3									
T3				T4								
T4					T5							
T5						T6						
T6												
T7								T8				
T8									T9			
T9										T10		
T10											T11	
T11												T12
T12												

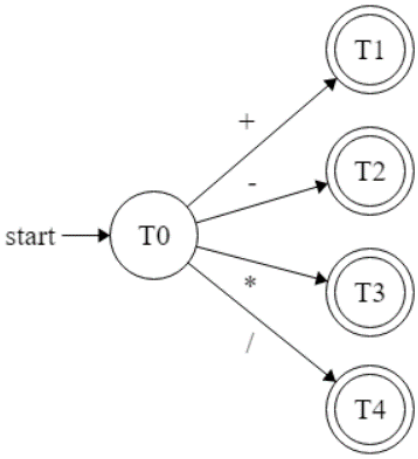


Léo Paol – 50181573
Christophe Mei – 50181587

OP

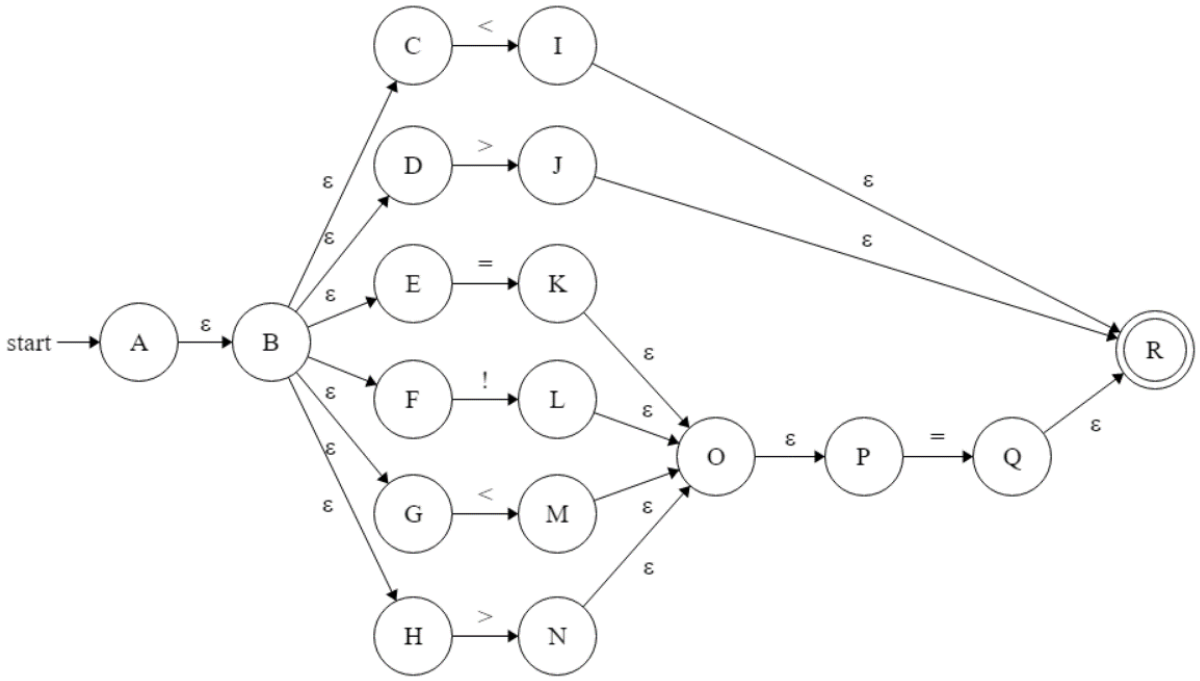


	+	-	*	/
T0	T1	T2	T3	T4
T1				
T2				
T3				
T4				

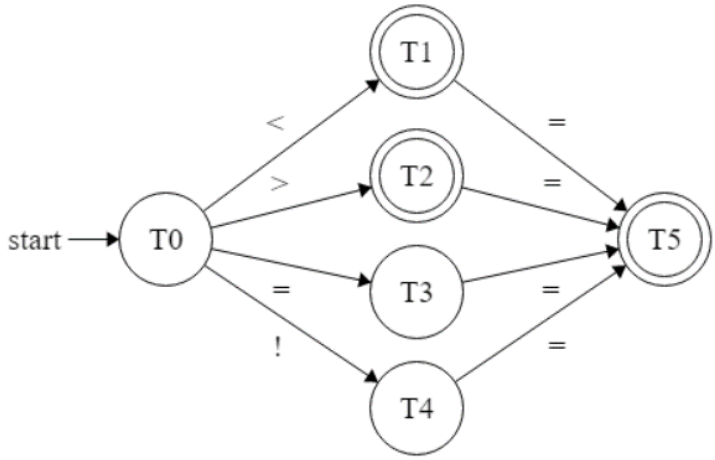


Léo Paol – 50181573
Christophe Mei – 50181587

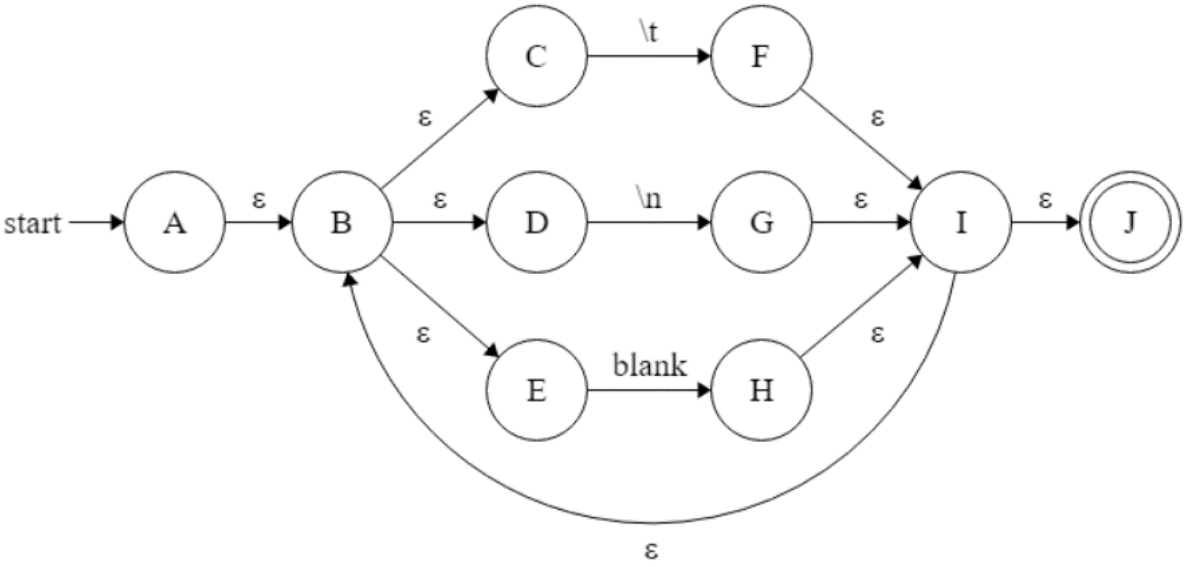
COMP



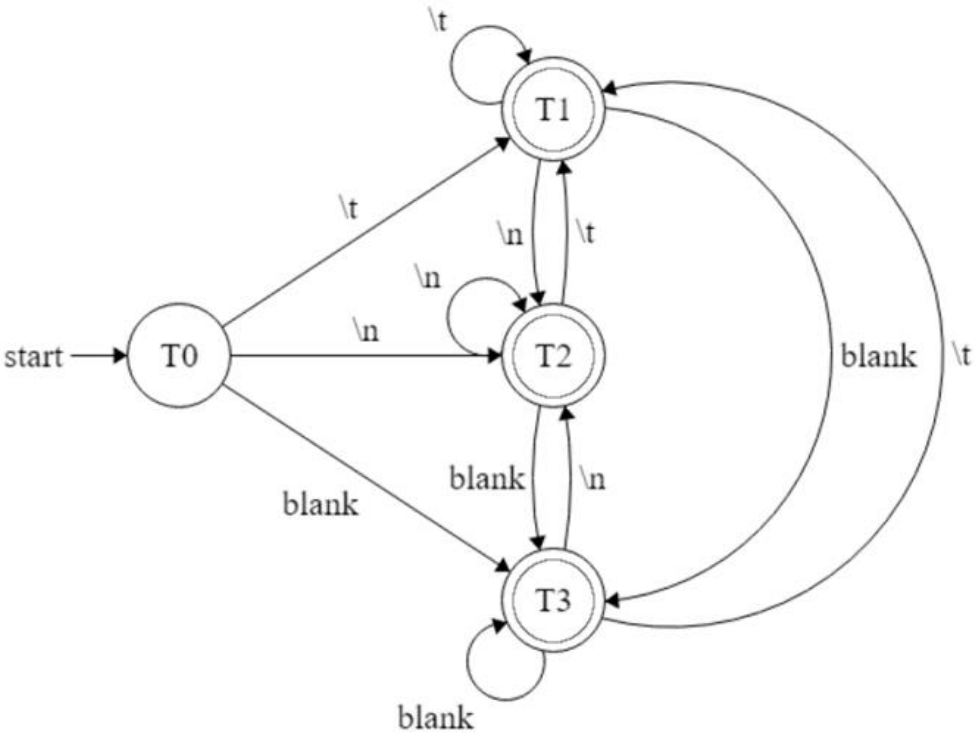
	<	>	=	!
T0	T1	T2	T3	T4
T1			T5	
T2			T5	
T3			T5	
T4			T5	
T5				



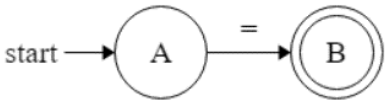
WSPACE



	\t	\n	blank
T0	T1	T2	T3
T1	T1	T2	T3
T2	T1	T2	T3
T3	T1	T2	T3

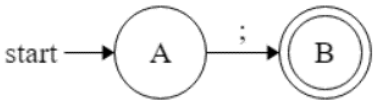


ASSIGN



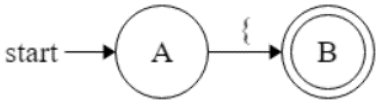
	=
T0	T1
T1	

TERM



	;
T0	T1
T1	

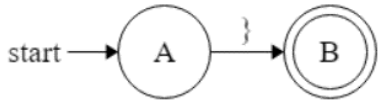
LSCOPE



	{
T0	T1
T1	

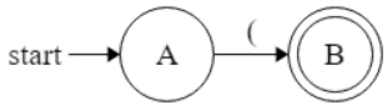
Léo Paol – 50181573
Christophe Mei – 50181587

RSCOPE



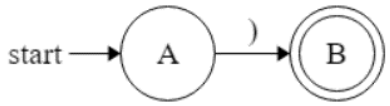
	}
T0	T1
T1	

LPAREN



	(
T0	T1
T1	

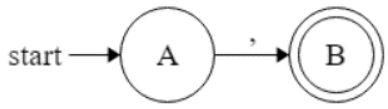
RPAREN



	)
T0	T1
T1	

Léo Paol – 50181573
Christophe Mei – 50181587

COMMA



	,
T0	T1
T1	

Léo Paol – 50181573
Christophe Mei – 50181587

Implementation of the lexical analyzer

Regular Expression Parser

token.py

The token.py file contains token enums and their DFA table.

For each token, the DFA table is an object with “key”, “value” and “final”.

“key” is the list of characters

“value” is the value of each case

“final” is the list of all final states

regexparser.py

The class Regex has a method isvalid() which have in argument the string to parse and the token enum to test.

We have a loop on the string and check for each character.

For each character, we check if the character is in the DFA table. If true, we check if there is a transition state from the current state. If true, the current state is replaced by the value from the table and we go on the next string character.

If the character is not in the table: return false

If there is no transition state from the current state: return false

If the current state is not a final state: return false

Main algorithm

Lexical-analyzer.py

In this file is written the main algorithm as well as the error handling.

The algorithm consists of reading the file content with a “window” that has a variable size and to check if the string contained in this window validate some regular expression using the regexparser.py. Once that we can’t validate any regular expression with what is contained anymore, we attribute a token name to the last state of the window.

For example a string “int;” will be first read with a window of size 1, “i”, and validate a regular expression for “ID”, then we increase the window size by one, “in” validate a regular expression for “ID”, then we increase the window size by one, “int” validate both “ID” and “INT”, then “int;” doesn’t validate any regular expression anymore so we take the last state of the window with the token name that as the most “weight”, here “INT” for “int”. Then the window will start again with a size of 1 on the character “;” and so on…

Things were not easy to implement for the “CHARVAL” regular expression with this logic as the character “ is recognized as an error when read alone.

Regarding the error system: it detects if there’s any issue with the program arguments, the file management (file doesn’t exist / impossible to write / …) and the errors contained in the file. For this last case it prints the line where the error comes from and the character(s) implied in the error.