An Algorithmic Representation of the Syntax Diagram of a Computer Programming Language

Anichebe Gregory Emeka

University of Nigeria, Nsukka, Enugu State, Nigeria

Abstract

Every programming language has its own syntax rules. Such rules can be represented with either the Backus-Naur form (BNF) notation or with a Syntax diagram (also called a Railroad diagram). BNF uses text-based mathematical notations for defining those rules, while a Syntax diagram employs a graphical approach. Converting any of the two techniques to an algorithm or computer program is somewhat difficult for students due to the recursive expressions used by each of the techniques in defining the syntactic rules of a grammar. The aim of this work is therefore to showcase how an algorithm for one of such techniques (namely, a syntax diagram) can be written for easy understanding and implementation with a computer. A Finite State automata (FSA) approach was adopted by the researcher for modelling any given grammatical rule of a programming language for easy implementation with a computer. The grammatical rules for generating an integer number was arbitrarily selected by the researcher, amongst other rules, for formulating the required algorithm. The algorithm (which is a pseudocode) was written to be in tandem with the FSA model for easier understanding and programming. Results showed that when the pseudocode is implemented with a computer with some trial data, every data that conformed with the grammatical rules for generating integer numbers was accepted as ”valid integer”, while other incoherent ones were declared “invalid integer”. This helps in smoothening the understanding of students of any rigorous or recursive problem for easy implementation with a computer.

Keywords

Backus-Naur form (BNF), Syntax diagram, Algorithm, Computer program, Finite State automata (FSA), Recursion

1. Introduction

A Backus-Naur form is a notation used in the field of Computer Science to express the syntax of a programming language [1]. The expression contains a list of all the rules that defines a particular grammar of a programming language. The three basic symbols used by the BNF are:

::= (which means, “is defined as”)
| (which means, “Or”)
< > (angular brackets that contain a category name)

For instance, to use the BNF to define the grammatical rules for generating an unsigned integer number, we have the following structure:

<pinteger> ::= <no>
<no> ::= <digit><no> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 1: The BNF definition for unsigned integer number

In plain English, the first line of figure 1 states that an unsigned integer, **<pinteger>**, is defined as a number, **<no>**. The second line contains two rules which state that: (i) **<no>** is defined as **<digit>** followed by **<no>** , or (ii) **<no>** is defined as **<digit>**. The third line contains 10 rules which state that **<digit>** is defined as ‘0’ or ‘1’ or ‘2’ or ‘3’ or ‘4’ or ‘5’ or ‘6’ or ‘7’ or ‘8’ ‘9’. The above grammar contains a total of 13 rules. Each rule has a **left part** and a **right part**. The ‘left part’ defines the ‘right part’. Any symbol appearing on the left part (or both parts) of a rule is called a **non-terminal symbol**, while any symbol that appears *only* on the right part (but not on both parts) of a rule is called a **terminal symbol**. In other words, the terminal symbols are the *sentences* (or *string*) that can be derived from a grammar. The first non-terminal symbol is called the **start** symbol. All generation of sentences of a grammar commences from the

ISIC’21: International Semantic Intelligence Conference,
February 25-27, 2021, Delhi, India

EMAIL: gregory.anichebe@unn.edu.ng (A.G)
ORCID: 0000-0003-4057-6277 (A.G)

start symbol. In figure1, the start symbol is <pinteger>. According to [2],

> The Backus-Naur Form is a way of defining syntax. It consists of
> - a set of terminal symbols
> - a set of non-terminal symbols
> - a set of production rules of the form, *Left-Hand-Side ::= Right-Hand-Side* where the LHS is a non-terminal symbol and the RHS is a sequence of symbols (terminals or non-terminals).

In figure1, the set of non-terminal symbols are, {<pinteger>, <no>, <digit>}, while the set of terminal symbols are, {0,1,2,3,4,5,6,7,8,9}. The grammar contains a recursive definition of how to generate an unsigned integer number, as shown in the second line of the grammar. Such recursive expression appears somewhat difficult for an ordinary person to easily understand, not to talk of converting it to an algorithm or computer program. According to William [3], recursion is often regarded as a deep mystery by novices in mathematics or computing, and so aught to be reserved for more advanced courses. The same recursive scenario occurs when the grammar of figure 1 is represented with a syntax diagram, as shown in figure 2.

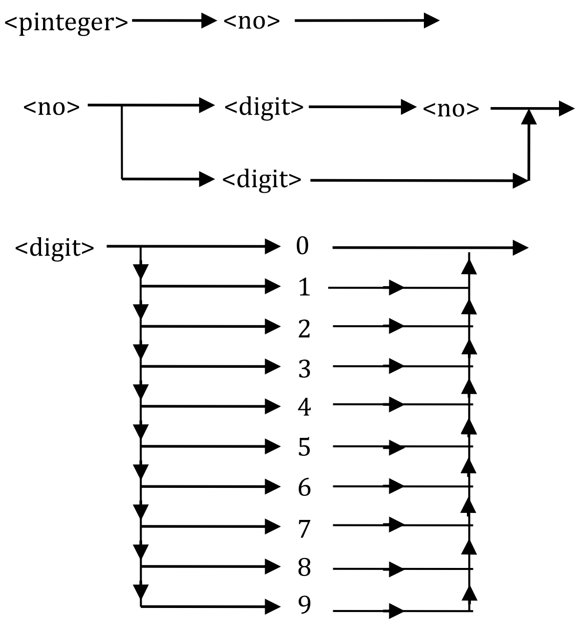


Figure 2: The syntax diagram definition for an unsigned integer

A 'parse tree' shows how valid (or invalid) sentences can be derived (or non derivable) from a grammar, as shown in figure 3 and figure 4, respectively for the derivation of the string, **614** and **2T** from the grammar defined in figure 1 (using BNF) or figure 2 (using syntax diagram).

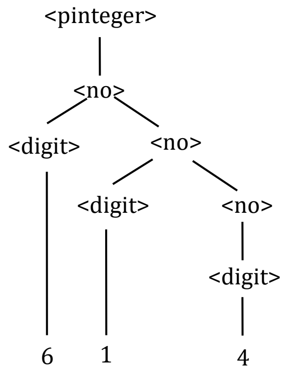


Figure 3: The string, **614** is *derivable* from the grammar, and is therefore a **valid integer**

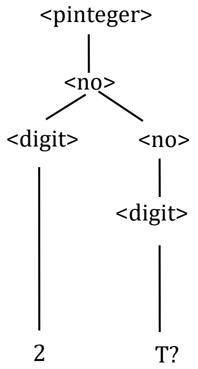


Figure 4: The string, **2T** is *not derivable* from the grammar because the symbol, T is not defined; therefore the string is an **invalid integer**

Figures 3 and 4 show the various recursive steps required for validating a sentence of a grammar. The question now is, "How can such steps be easily implemented programmatically with a computer?" This forms the basic research question for this work, and of which a solution to it is expatiated in section 3 under "Methodology".

2. Literature Review

In Computer Science, recursion is typically used for solving problems that involve *recursive relations* (such as the generation of integer numbers shown in figure 1 and figure 2 of section1). According to [3], "The concept of recursion comes from mathematics where we often encounter recursive relations". For example, consider the problem of raising a real number, X, to an integer power, N. The problem can be solved recursively by stepwise refinement (i.e. breaking a problem down into simpler computational parts), as shown in table 1.

Table 1: Evaluation of X^N by recursion

Problem	Recursive process	Step
X^N	$X \, . \, (X^{N-1})$	1
	$X \, . \, X \, . \, (X^{N-2})$	2
	$X \, . \, X \, . \, X \, . \, (X^{N-3})$	3
	etc.	etc.

Thus, each recursive step is a simpler version of the initial problem. The process continues until a termination point is reached (i.e. $X^0 = 1$ in the above example) where no further recursive process occurs, and the final result then determined by backward substitution. The above problem can be solved programmatically using a Java function as follows:

```
public static double power(double X, int N)
    {
    if(N = = 0)
    {
      return 1.0;
    }
    else
    {
      return X * power(X, N-1);
    }
}
```

Figure 5: A Java function code for evaluating the function, X^N, recursively

Thus, we can see that, recursion is a process of breaking a computation down in such a way that a simpler computation of the same kind with the previous problem is derived, and the decomposition process continues until a trivial stage is reached. Simply put, [4] explains that, "Recursion is a process whereby a function calls itself inside its body for the execution of a task until a base case is reached". The beauty of recursion is that it presents a clearer, intuitive, and simpler solution to a problem which would have been very difficult or too clumsy to solve through other means [5]. For instance, a popular mathematical puzzle called 'The towers of Hanoi' can be easily solved with recursion, as elegantly illustrated by [5], [3], and [6], to mention a few. The puzzle would have been too clumsy or nasty to solve through other means such as *Iteration*.

The BNF notation, according to [7], was named after the two inventors: John Backus of the United States of America, and Peter Naur of Denmark. The notation makes extensive use of recursion in defining the syntax of a programming language very succinctly. The mystery behind recursion can be demystified by understanding it as a stepwise refinement of the initial problem (by divide-and-conquer technique) until a trivial case (or terminal point) is reached that requires no further simplification. There are [now] many variants and extensions of BNF, generally either for the sake of simplicity and succinctness, or to adapt it to a specific application, [8]. Typical examples of these variants are given by [9] as, "Extended BNF (EBNF) notation", "Augmented BNF (ABNF) notation", and "Regular extensions to BNF notation". The EBNF notation is almost a superset of BNF, and is now the most commonly used structure for describing the grammar of a language. On the other hand, the ABNF notation is commonly used for describing the Internet Engineering Task Force (IETF) protocols; while the Regular extensions to BNF notation are popularly used by the Python programming language for its lexical specifications. Table 2 shows some of the basic differences amongst these notations.

Table 2: Some basic differences amongst the BNF, EBNF, ABNF, and Regular extensions notations

					Examples			
Description of symbol	**BNF**	**EBNF**	**ABNF**	**Regular extension**	**BNF**	**EBNF**	**ABNF**	**Regular extension**
non-terminal symbol	uses angular bracket, < >	appears plain	appears plain	appears plain	<answer>::= <reply>	answer = reply;	answer= reply	answer::= reply
definition of rule	::=	=	=	::=	<age> ::= <integer>	age = integer;	age = integer	age ::= integer
alternative rule or choice	|	|	/	|	<reply>::= "yes" | "no"	reply= "yes" | "no";	reply= "yes" / "no"	reply::= "yes" | "no"

End of a rule	whitespace	;	whitespace	whitespace	<result> ::= <score>	result = score;	result= score	result ::= score
concatenation symbol	the symbols appear next to each other	,	same as BNF	same as BNF	<CGPA>::= <digit>.<digit >	CGPA = digit, "." ,digit;	CGPA = digit "." digit	CGPA:: = digit "." digit
optional rule	defined as a separate rule	[...]	[...]	[...]	<name> ::= <title> <next> | <next>	name =[title,] next;	name =[title] next	name::=[title] next
repeating an expression **0** or **1** times	defined as a separate rule	{...}	* (used as prefix)	? (used as postfix)	<title> ::= <"">|<"mr"> | <"mrs"> | <others>	title = {"mr" | "mrs" | others};	title = *("mr"/ "mrs"/ others}	title :: = ("mr" | "mrs" | others)?
repeating an expression **1** or **more** times	defined as a separate rule	{...}	n* (used as prefix)	+	<age>::= <digit><no>| <digit>	age = digit{digit};	age =1*digit;	age::= digit+
repeating an expression **n** to **m** times (where n ≥ 1, and m > n)	defined as a separate rule	{...}	n*m (used as prefix)	n*m (used as postfix)	<age>::= <digit> | <digit><digit >| <digit><digit ><digit>	age = digit{2*digit };	age = 1*3digit;	age :: = digit1*3

The syntax diagram is used to represent the BNF notation pictorially for easier understanding, whereby each of the non-terminal symbols in a grammar is represented as a block or module that performs a given task, as earlier illustrated in figure 2. By the words of [10], "A syntax diagram is a pictorial method of representing the format of components in a programming language, and the direction of the arrowed lines indicates the order in which the diagram is to be read or followed". This is where the use of Finite State Automata (FSA) becomes very necessary in showing pictorially, as well as in a more compact form, how the sentences (or terminal symbols) of a grammar can be generated for easy implementation with a computer. The use of FSA defines the syntax of a grammar as well as its recursive processes very clearly so that valid or invalid sentences can easily be identified. According to [11],

> A finite state automata (FSA) is a simple idealized machine that recognizes patterns from the input [which consists of finite string of symbols] taken from some character set (or alphabet). The job of an FSA is to *accept* or *reject* an input depending on whether the pattern defined by the FSA occurs in the input

According to [12], an FSA (simply denoted as, M) consists of 5 components: M = (Q, ∑, q_0, δ, F), where,

- Q is a finite set of states
- ∑ is the input alphabet
- q_0 ϵ Q is the start state
- F ϵ Q is the set of final or accepting states
- δ is a transition function that takes up a pair of state and input symbol (say, <q,a>) in order to determine the next state (say q′) for the machine; i.e., δ(q,a) = q′

The above representation conforms perfectly with the generation of sentences using the syntax diagram or BNF notation, as shown in table 3.

Table 3: The similarity between syntax diagram/BNF notation and Finite state automata

Syntax diagram / BNF notation	**Finite state automata equivalence**
terminal symbols	Q (finite set of states)
supplied input string	∑ (input alphabet)
start symbol	q_0 (start state)
valid sentences	F (set of final or accepting states)
Production rule	δ (transition function)

This work therefore showcases how the FSA can be used to model a syntax diagram/BNF notation for easy conversion to a computer program.

3. Methodology

The syntax for generating signed or unsigned integer numbers using the syntax diagram or the BNF notation was used by the researcher to illustrate how such rules can be implemented programmatically. Figure 6 shows the syntax for generating signed or unsigned integer numbers using the BNF notation.

<int> ::= + <no> | – <no> | <no>
<no> ::= <digit><no> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 6: The BNF definition of signed and unsigned integers

The syntax diagram representation of figure 6 is shown in the following figure 7.

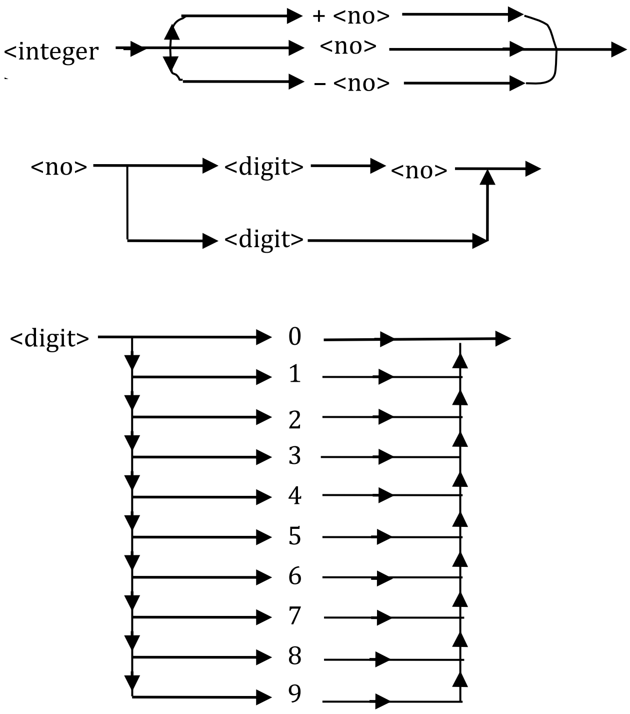
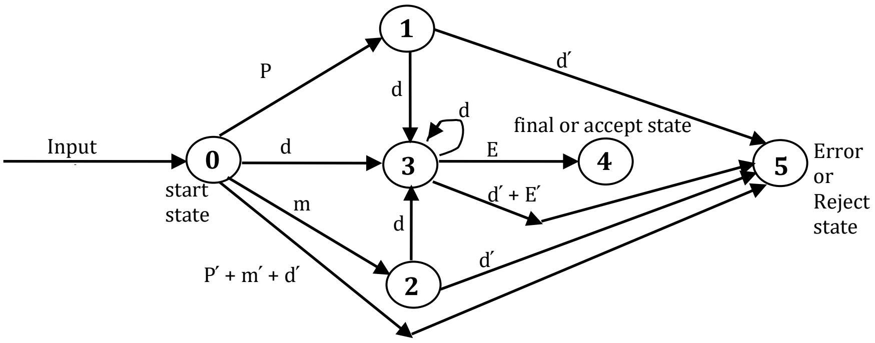


Figure 7: The syntax diagram definition of signed and unsigned integers

We can use a finite state automata to model any of the two techniques, as shown in figure 8.



Variable description	
P = plus (+) sign	P′ = not a plus sign
m = minus (-) sign	m′ = not a minus sign
d = digit (0,1,2,3,4,5,6,7,8,9)	d′ = not a digit
E = 'Enter key' character	E′ = not 'Enter key' character

Figure 8: The finite state automata for generating signed and unsigned integers

Figure 8 is a *directed tree* that contains all the 'terminal symbols' of the grammar (which are represented as **nodes** in the diagram), and 'production rules' of the grammar (which are represented as **arrows**). The reason for using only 'terminal symbols' and 'production rules' for such representation is because every grammar of a programming language boils down to the generation of *sentences* (i.e. concatenation of *terminal symbols*) from the grammar, as well as *how* the sentences are derived (i.e. the use of *production rules* to derive the sentences). Thus, Figure 8 consists of 6 states (i.e., states 0,1,2,3,4, and 5). State(0) is the 'start state' at which the automata machine receives an input string for parsing. State(1) is the state for receiving '+' sign. State(2) is the state for receiving '–' sign. State(3) is the state for receiving digits(0,1,2,...,9). State(4) is the 'accept state' for an input string that successfully moves from state(0) to state(4). Finally, state(5) is the 'reject state' for an input string that moves from state(0) to state(5). The arrows in the figure indicate how the terminal symbols are concatenated to generate a valid integer number. Thus, the automata machine changes state (or transition) according to the characters in the input string in order to determine valid or invalid integer. For instance, the input string, **+621** undergoes the following states: 0 => 1 => 3 => 3 => 3 => 4. It is therefore accepted as a valid integer number.

4. Data Analysis

Table 4 shows in detail how the automata machine of figure 8 can be used to parse a sample of some input string such as 725, – 6A31, and +9.

Table 4: An illustration of input string parsing with the automata machine of figure 8

Input string	**Start state**	**character by character parsing**	**Next state**	**Result**
725<enter key>	0	7	3	
		2	3	
		5	3	
		<enter key>	4	Valid integer
–6A31 <enter key>	0	-	2	
		6	3	
		A	5	Error (invalid integer). The symbol, 'A' is illegal
+9<enter key>	0	+	1	
		9	3	
		<enter key>	4	Valid integer

- The input string, *725<enter key>*, is parsed as follows: at state(0), the finite state automata (FSA) receives the input string supplied by the user; the 1st character of the string is digit(7) which belongs to state(3); the FSA therefore moves from state(0) to state(3). the 2nd character of the string is digit(2) which also belongs to state(3); the FSA therefore moves from state(3) to state(3) – a recursive movement or transition. The 3rd character of the string is digit(5) which again belongs to state(3); the FSA therefore makes another recursive transition from state(3) to state(3). The last character of the string is the <enter key> which belongs to state(4); the FSA therefore moves from state(3) to state(4) – the 'Accept state'. The string, *725*, is subsequently accepted as VALID integer.
- Similarly, the second input string, –6A31<enter key>, is parsed as follows: at state(0), the finite state automata (FSA) receives the input string supplied by the user; the 1st character of the string is a minus(–) sign which belongs to state(2); the FSA therefore moves from state(0) to state(2). the 2nd character of the string is digit(6) which belongs to state(3); the FSA therefore moves from state(2) to state(3). The 3rd character of the string is letter(A) which belongs to state(5) – 'Error or Reject state' because the received character is d′ (i.e. not a digit); the FSA therefore moves from state(3) to state(5). Subsequently, the string, *–6A31*, is rejected as INVALID integer (without parsing the remaining characters, '31', in the string).
- Lastly, the input string, +9<enter key>, is parsed as follows: at state(0), the finite state automata (FSA) receives the input string supplied by the user; the 1st character of the string is a plus(+) sign which belongs to state(1); the FSA therefore moves from state(0) to state(1). The 2nd character of the string is digit(9) which belongs to state(3); the FSA therefore moves from state(1) to state(3). The 3rd character of the string is the <enter key> which belongs to state(4); the FSA therefore moves from state(3) to state(4) – the 'Accept state'. The string, *+9*, is subsequently accepted as VALID integer.
- The pseudocode for implementing the finite state automata of figure 8 is shown in figure 9 that follows.

```
1.  input any integer number as string
          1.1  input strgintno
2.  determine the number of characters, N, of the string, strgintno
          2.1  N = length(strgintno)
3.  determine the first character, fchar, of the string, strgintno
          3.1  if  fchar = ‘+’ then
                    process state1()
               else if  fchar = ‘–‘ then
                    process state2()
               else if  fchar = digit(‘0’ or ‘1’ or ‘2’ or ‘3’ or ‘4’ or ‘5’ or ‘6’ or ‘7’ or ‘8’ or ‘9’) then
                    process state3()
               else
                    process state5()     //error routine
               end if
4.  //definition of functions
          4.1  state1()
                    Determine the second character, schar, of the string, strgintno
                    if  schar = digit(‘0’ or ‘1’ or ‘2’ or ‘3’ or ‘4’ or ‘5’ or ‘6’ or ‘7’ or ‘8’ or ‘9’) then
                              process state3()
                    else
                              process state5()     //error routine
                    end if
          4.2  state2()
                    Determine the second character, schar, of the string, strgintno
                    If  schar = digit(‘0’ or ‘1’ or ‘2’ or ‘3’ or ‘4’ or ‘5’ or ‘6’ or ‘7’ or ‘8’ or ‘9’) then
                              Process state3()
                    Else
                              Process state5()     //error routine
                    End if
          4.3  state3()
                    //determine the subsequent characters of the string, strgintno, as follows:
                    for charcount = 1 to N
                        if strgintno(charcount) = digit(‘0’ or ‘1’ or ‘2’ or ‘3’ or ‘4’ or ‘5’ or ‘6’ or ‘7’ or ‘8’ or ‘9’) then
                              Continue
                        else
                              process state5()     //error routine
                        end if
                    next charcount
                    //display the status of the data supplied
                    Print  “valid integer”
                    Stop
          4.4  state5()
                    //display the status of the data supplied
                    Print “invalid integer
                    Stop
```

Figure 9: The pseudocode for the implementation of the automata machine of figure 8

5. Results and Discussion

Table 5 shows the output of the program of figure 9 when some input data are supplied to the computer during its execution.

Table 5: A sample output of the program of figure 9

Input string	Output
5A	WRONG integer
C32	WRONG integer
-946890	VALID integer
+73.6	WRONG integer
+152	VALID integer
-600	VALID integer
496+	WRONG integer
8112	VALID integer
16 – 4	WRONG integer
2	VALID integer

008	VALID integer
1,500	WRONG integer

The use of automata machine to model the generation of integer numbers (which are defined recursively by a syntax diagram or BNF notation) makes the programming aspect of its implementation very easy to write and understand. The **6** states in the machine were well represented in the program (as ***functions*** that perform specific tasks) so that any character that goes contrary to the rules of the grammar is reported by state(5) as an error, and the parsing process is terminated immediately without processing the remaining part of the input string (if any). For instance, the input string, '+73.6' in the sample output of table 4 is 'WRONG integer' because the decimal point(.) is not defined in the grammar being represented by the automata machine. Again, the input string, '496+' is 'WRONG integer' because the **order** of arrangement of the characters does not agree with the syntax of the grammar. On the other hand, any input string that transits successfully from state(0) to state (4) is accepted as VALID integer because it agrees with the syntax of the grammar in (i) order of arrangement, and (ii) in symbols.

6. Conclusion

Recursion is a powerful mathematical technique used ubiquitously in Computer science. The BNF notation or syntax diagram employs it extensively in defining the grammatical rules of a programming language. The use of Finite State Automata (FSA) in transforming the 'terminals' and 'production rules' of a grammar into a *directed graph* helps immensely in modelling the grammar into a very compact form for easier understanding and programming.

References

[1] "Theory of Computation: Backus-Naur form", 2020. URL: https://en.M.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Theory_of_computation/Backus-naur-form

[2] "What is BNF?", 2020. URL: http://www.cs.umsl.edu/~janikow/cs4280/bnf.pdf

[3] William, I.S., "Structures and Abstractions – an introduction to Computer Science with Turbo Pascal", Richard Irwin inc., 1992, p.409, p.415, p.428

[4] "What is Recursion? How is it helpful, and where it is used?", 2020. URL: www.equationanswers.com/c/c-recursion.php

[5] Liang, Y.D., "Introduction to Java Programming", 3rd ed., Prentice-Hall Inc., Upper Saddle River, New Jersey, 2001, p.123

[6] Deitel, P. & Deitel, H., "Java-How to Program", 8th ed., Pearson Education Inc., Upper Saddle River, New Jersey, 2010, p.790

[7] Daniel, D.M. and Edwin, D.R., "Backus-Naur form (BNF)", 2020. URL: https://www.researchgate.net/publication/262254296_Backus-Naur_form_NBF

[8] Wikipedia, "Backus-Naur form", 2020. URL: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

[9] "Grammar_The language of languages(BNF, EBNF, ABNF, and more)", 2020. URL: http://matt.might.net/articles/grammars-bnf-ebnf/

[10] Holmes, B.J., "Pascal Programming", 2nd ed., The Guernsey Press Co Ltd., Channel Islands, 1990, p.30

[11] "Finite Automata", 2020. URL: https://www.cs.rochester.edu/u/nelson/courses/csc_173/fa/fa.html

[12] Carol, C. & David, J.E., "Finite-State Automata", 2020. URL: https://eng.libretexts.org/Bookshelves/Computer_Science/Book%3A_Foundations_of_Computation_(Critchlow_and_Eck)/03%3A_Regular_Expressions_and_FSA's/3.04%3A_Finite-State_Automata