Annales Mathematicae et Informaticae
57 (2023) pp. 65–77
DOI: https://doi.org/10.33039/ami.2023.04.002
URL: https://ami.uni-eszterhazy.hu

Logical conditions in programming languages: review, discussion and generalization*

Benedek Nagy[ab*], **Khaled Abuhmaidan**[c], **Monther Aldwairi**[d]

[a]Department of Mathematics, Faculty of Arts and Sciences,
Eastern Mediterranean University, Famagusta,
North Cyprus, via Mersin-10, Turkey
corresponding author: nbenedek.inf@gmail.com

[b]Department of Computer Science,
Institute of Mathematics and Informatics,
Eszterházy Károly Catholic University,
Eger, Hungary

[c]Faculty of Computing and Information Technology,
Sohar University, Oman
khmaidan@su.edu.om

[d]College of Technological Innovation, Zayed University,
144534 Abu Dhabi, United Arab Emirates
monther.aldwairi@zu.ac.ae

Abstract. Boolean logic is widely used in almost every discipline including linguistics, philosophy, mathematics, computer science and engineering. Boolean logic is characterized by the two possible truth values, and various logical connectives/operations allow us to make compound statements, conditions. Most of the programming languages, if not all, have some of the logic operations: conjunction, disjunction and negation. Actually, since the set of these three operations form a basis, any logical statement can be formed by them. However, on the one hand, there are smaller bases as well, i.e., one of the conjunction or disjunction is already superfluous. Moreover, there are bases with only one operation, e.g., by NAND. On the other hand, one may allow other operations helping the programmer/user to define the conditions of conditional statements and loops in a simpler manner. In this paper we

*This research was supported by Zayed University, Research Office, Research Incentive Fund Award #R20089.

Submitted: April 8, 2022
Accepted: April 20, 2023
Published online: April 28, 2023

discuss these issues, including some practical points, implementation issues and short cut evaluations for various operations.

Keywords: High level programming languages, conditional statements, loop conditions, logical connectives, short cut evaluation, formal logic

AMS Subject Classification: 03B70, 03B05, 68N15, 68N20, 97P40

1. Introduction

Classical Boolean logic is a well-known and widely used basis of various mathematical disciplines and also of computer sciences, including hardware and software related fields. In this paper, we are, first, analysing how it occurs in high-level programming languages, e.g., conditional statements (see Section 2). Then overviewing classical logic we give reasons why the logical operations conjunction, disjunction and negation are used, and not a smaller basis of the operations. However, there are other widely used logical operations which may help our lives to be easier, e.g., implication and equivalence, if they are also allowed to be used in our formulations. But then, one may ask the question, why the programmers should translate all their conditions to a form that may include only that three connectives that are built in the programming languages, and why the others are not like that. This is the motivation of our paper. We show how the other usual logical operations can be used in our programs. We give some thoughts also on possible implementations, e.g., based on preprocessing or a functional representation that is close to the so-called Polish notation introduced by Łukasiewicz [12].

2. Conditions in programs

In high-level programming languages both the branching and the loop structures occur frequently. The branching is usually done by conditional statements. The usual form of a conditional statement (although it may be slightly varied in various programming languages) is either

`if` (condition) `then` statement

or

`if` (condition) `then` statement(1) `else` statement(2)

For details about the syntax used in various languages that are not explained here, the readers are referred to textbooks [8–11, 22, 25, 28]. Note that some of the languages are case sensitive and some of them are not.

In this paper we are interested and concentrate on that part which includes logic: The (condition) in the above structures means logical condition, i.e., a logical formula that is evaluated by the computer, and if it evaluates to true then the

statement (or statement(1) in the second case) is executed, otherwise they are not executed at this time (but statement(2) is executed in the second case) and the program then continues by the next statement.

The logical condition can be a simple condition, e.g., $(x < 15)$ or a compound statement. In most of the programming languages the statement can also be compound, i.e., a block built up by a sequence of statements, for syntax see the mentioned textbooks.

The loops usually have heads with conditions and bodies with statements. The condition in the head is also a logical condition that is evaluated to be either true or false and based on that either the statements of the body of the loop are executed (once more) or the program continues by the statement after the body of the loop. There are various types of loops that can be used (depending also on the used language). Most of the languages have `for` loops, however, in some of the languages that type of loop does not have a formal logical condition, but the body should be executed for some specific values of the loop variable (e.g., in Pascal and Python). In some other languages, including C, C++, Java and also Javascript, the for loop is very similar to the `while` loops that we are explaining shortly in the sequel. In `while` loops, after the word `while` a condition is written (for syntactic details the author is referred to the textbooks mentioned before), then the body with its statement(s) is written. In the programming languages in which the for loop is similar, after the word `for` in brackets, first an initialisation statement (that is executed only once before checking the loop condition at the first time), then in the middle, the loop condition itself that is very similar to the loop condition of `while` loops, finally, the third part in the bracket is the increment statement, that is a statement which is executed after the body each time when the body is executed, right before the loop condition is (re)checked. Finally, there is also another type of loop, where the condition is after the body. In Pascal it is written as follows: between `repeat` and `until` the statement(s) and right after `until` the logical condition (to decide if the statement(s) in the body are executed once more). In other languages, e.g., in C, C++, Java and Javascript, these are written as `do - while` loops having the body between these words and the logical condition after the second. For more details about its syntax in various languages we recommend to check the mentioned textbooks. We note that any algorithms can be implemented without this type of loop.

Here, we are concentrating on the logical conditions. Thus, let us see how one can build complex conditions. The usual classical connectives to make compound statement are the conjunction, disjunction and negation, see Table 1. In some high level languages there is a simple datatype for Boolean values and there are also built in constant values for both the truth-values.

Although in the programming language C, there is no specific type for Boolean values, integers and pointers can be used also for this purpose in such a way that the value 0 or $NULL$ is understood as false, while any other values are understood as true. In various other languages the values 0 and 1 also play the role of false and true, for details the reader is referred to the above listed (text)books.

Table 1. The logical operations and constants in various high-level programming languages.

Programming Language	conjunction	disjunction	negation	Boolean/logical type	true,	false
Pascal	`and`	`or`	`not`	boolean	`true`	`false`
C	`&&`	`||`	`!`	-		
C++	`&&`	`||`	`!`	bool	`true`	`false`
Java	`&&`	`||`	`!`	boolean	`true`	`false`
JavaScript	`&&`	`||`	`!`	boolean	`true`	`false`
Python	`and`	`or`	`not`	bool	`true`	`false`

3. Bases of Boolean logic

As we have already seen, in Boolean logic there are two truth-values: true and false. Usually they are represented by 1 and 0, respectively. Boolean variables, denoted by, e.g., A, may take one of these values at interpretation and evaluation. As usual in almost every part of mathematics, the operators are unary and binary ones. The only unary operator, the negation is usually (syntactically) written as $\sim A$ or $\neg A$ in formal logic (with a formula A). In Boolean algebra, the notation $\bar{A}$ is used, while various programming languages use various notations as we have already shown them in Table 1. The (semantical) result of this operation is that the compound statement having this operation as the main operation has the opposite truth value than the subformula without this main operation (i.e., the truth value of formula A).

Table 2 shows all possible Boolean operations. Note that the first and last rows are not functions of any of the variables A and B, but always false and true (they are in fact the Boolean constants and can be seen as functions with zero arguments). As we have already mentioned in many programming languages the programmer can use these constants as well, e.g., to have a (theoretically) infinite loop with a condition that is always true. There are two other rows, which are in fact the copies of the values of A and B, respectively. These rows do not define logical connectives, neither.

Further, two of the rows are showing the negation of the variables A and B, respectively, thus the unary operation negation occurs twice in the table.

The remaining ten rows define the ten binary operations, and there are no more [7, 18]. On the one hand, obviously, the conjunction (logical and) and disjunction (logical or) are among them. On the other hand, most of the other operations have also their own names, as they play important roles, either in natural languages (implication, equivalence, exclusive or), in logical deductions (implication) or in the hardware industry (NAND, NOR).

As we have seen there are eleven Boolean operations. How it can happen then, to use only three of them in programming languages? The answer is in the properties of the Boolean algebra [1, 15], i.e., the concept of base set of operations.

Table 2. The Boolean operations (based on all possible binary Boolean functions).

formula	name	values			
A		1	1	0	0
B		1	0	1	0
0	-	0	0	0	0
$A \nmid B$	NAND	0	0	0	1
$A \not\subset B$		0	0	1	0
$\neg A$	negation	0	0	1	1
$A \not\supset B$		0	1	0	0
$\neg B$	negation	0	1	0	1
$A \bigoplus B$	eXclusive OR	0	1	1	0
$A \mid B$	NOR, Sheffer stroke	0	1	1	1
$A \wedge B$	and	1	0	0	0
$A \equiv B$	EQUivalence	1	0	0	1
B	-	1	0	1	0
$A \supset B$	IMPlication	1	0	1	1
A	-	1	1	0	0
$A \subset B$	reverse implication	1	1	0	1
$A \vee B$	or	1	1	1	0
1	-	1	1	1	1

We say that a subset S of the operations listed in Table 2 is a base, if for any number of Boolean variables, one can write equivalent formula L' using the variables and the operations of S to any logical formula (or Boolean function) L of the same variables. A base is also called a functionally complete set of operations, and by Post, it is known that the set must contain at least one operation without each of the following five properties:

- monotonic (by "increasing" the input, i.e., by changing the value of any of the arguments from 0 to 1, the value of the result cannot decrease, i.e., cannot switch from 1 to 0, e.g., $\wedge$ and $\vee$ are monotonic),
- linear/counting (those rows of Table 2 are referred here that have even number of 1's, and thus, also even number of 0's),
- self-dual (those operations are counted here for which by flipping – i.e., changing from 0 to 1 or vice versa – the values of all the arguments, the result must also change to the opposite),
- truth-preserving (if all the arguments have a value truth, then also the result is true) and
- false-preserving (if all the arguments have a value false, then also the result is false),

see more details, e.g., in [19].

It is well-known (see, e.g., [13, 18]) that $S = \{\neg, \wedge, \vee\}$ is a base. On one hand, this is the most used base, since in Boolean algebraic description usually, exactly these operations are used. Further, these operations are used to define the conjunctive and disjunctive normal form expressions, and it is well-known that for any Boolean formula there is an equivalent one in conjunctive/disjunctive normal form. On the other hand this is not a minimal base, as we recall in the next subsection.

3.1. Why not smaller bases?

Although, $S = \{\neg, \wedge, \vee\}$ is a base, it is not minimal: by the use of the De-Morgan identities (and the law of double negation), i.e., $(A \wedge B)$ is equivalent to $\neg(\neg A \vee \neg B)$ and $(A \vee B)$ is equivalent to $\neg(\neg A \wedge \neg B)$, thus one may exclude either $\vee$ or $\wedge$.

Then, in a minimalist language (in terms of defining as less things as possible to make a high level programming language), one may use one of the sets $S_\wedge = \{\neg, \wedge\}$ and $S_\vee = \{\neg, \vee\}$ and keep only negation and one of the binary connectives of S.

Moreover, there are even smaller bases: Sheffer has already shown that one operation is enough to express any Boolean formulae/function [24], he has used the operation named after him, as Sheffer stroke (and it is known also as NOR, i.e., Negated OR, in Boolean algebra and logic gates). On the other hand, the operation NAND also has this property: any logical formulae can be expressed also by using the sole operator NAND. These operations do not have any of the five properties described by Post.

Now, we may ask the question that is given in the head of the subsection: if we have smaller bases, why do we use three connectives and not less in the programming languages.

The answer is obvious: these three connectives are so natural and it is very easy to connect them to natural languages:

- the negation refers for the negative statements, when (usually) the verb part is negated, e.g., "John does not go to the school today." Sometimes another verb meaning the negation of the other exists in the language, e.g., 'remember' could play similar role as 'do not forget'.
- the conjunction refers to connect two (independent) statements by the connective 'and', e.g., "John goes to the school and Bob plays football today." In the language we may use various connectives, e.g., 'and', 'but', or maybe only a semi-colon to connect the two statements and form a compound statement in this way.
- the disjunction usually refers to sentences compounded by the connective 'or', e.g., "Rick likes the taste of the coffee or he likes the hot drinks."

However, to formalize conditions, one needs some special care, as there are various differences in formal logic that is used in mathematics and programming and the 'logic' used in natural languages.

- The connective 'and' may have the meaning 'and then', e.g., "Bob went to the supermarket and he has bought some drinks." In this sense, this connective is not always commutative. It is also dangerous to abbreviate the statements and put the 'and' between some parts of the sentences without repeating the other parts of the statement. Ambiguity occurs, e.g., "You are allowed to distribute the softwares I wrote and I licensed." (It may not be clear if it is only about the softwares which I have both written and licensed or also those which I only have written or only have licensed...)

- The connective 'or' frequently has 'xor' meaning, in the sense that we want to allow only one of the options, e.g., "Jack is drinking a coffee or he is drinking a tea"; "Bob will do his homework or he will fail in the course." Thus in some cases instead of simply writing 'or', in some text 'and/or' is written in the usual meaning of the disjunction. (Sometimes to highlight the 'xor' feature, the format, "either he will do the homework or he will fail" is used, but the usage of 'either' is optional in the language, even if the meaning of the sentences should be the same.)

Now, we turn to give some notes on the usage of a base with only one operator. The options would be to use only NAND or only NOR. However, this would make the writing of the conditions long and hardly understandable causing extra care and possible faults and bugs in coding. Although mathematically and theoretically, these would be options, in practice it would be not a good way to use the high level programming languages in any of these ways. As computers become widespread, to allow more and more people to learn programming and write their own codes it would be very difficult to learn.

Although, in the hardware industry, it could be a good decision, as the connection of the logic gates can be easily checked by simulations and also computers support the design of the circuits, the programmers have not been trained to convert any types of Boolean logical conditions/statements to formulae using only one connective.

4. Expanded Logic

Now, since we have seen why it is not a good idea to use to small number of logical connectives, we show how the set of used connectives can be expanded to allow some more of the usual connectives.

As we have shown in Table 2, the following well-known and widely used operations, i.e., connectives have three true and one false values in their truth table: disjunction, implication, NAND. The connectives conjunction and NOR are defined in the opposite way, i.e., by only one true and three false values. The other two well-known operations, the equivalence and the XOR have two-two true and false values. These facts turn to be important in the sequel.

4.1. Practical implementation

A preprocessor or a macro may substitute the formulae containing any of the IMP, NAND, NOR, EQU, XOR operations to formulae with only conjunction, disjunction and negation. In this way, the code is first transformed to an equivalent code in the traditional/usual programming language, and thus, the programmer may use these larger set of connectives to write a more intuitive and simpler condition, but, in fact, the computer will execute the code after preprocessed in the traditional manner. On the other hand, since programming languages are focusing on performance when making the executable code, and the logical operations on the machine code include e.g., XOR, this operation can be compiled and executed directly.

Nevertheless, as a possible solution to include these new logical operations, we give possible substitution(s) that a preprocessor may do for each of the five operations. Let `a` and `b` abbreviate the two (simple or compound) conditions that are connected by the given operator, their (truth-)value can be 0 and 1.

- `a IMP b` can be written as `NOT a OR b`
- `a NAND b` can be written as `NOT a OR NOT b`
- `a NOR b` can be written as `NOT a AND NOT b`
- `a EQU b` can be written as `(a AND b) OR (NOT a AND NOT b)` or `(a AND b) OR NOT (a OR b)`
- `a XOR b` can be written as `(NOT a AND b) OR (a AND NOT b)`

We may need to use fully bracketed formulae to make their meaning clear. Especially, the substitution of the last two operators, the EQU and XOR seem long. Now some tricks are shown which may be used in the programming language C, where instead of the Boolean type, integers are used. Let `a` and `b` abbreviate the two conditions (which may have values 0 or 1), as before, then we have the following. We start with the operators AND and OR, which do not need to be substituted or interpreted in other ways, as they are built in, however, for the sake of completeness and to feel what types of ideas are behind the scene, we start with those.

- `a AND b` can be written as `a * b`
- `a OR b` can be written as `a + b`
- `a IMP b` can be written as `a <= b`
- `a NAND b` can be written as `!(a * b)`
- `a NOR b` can be written as `!(a + b)`
- `a EQU b` can be written as `a == b`

- `a XOR b` can be written as `a != b` or `!(a == b)`

As here, we are showing rewriting that is close to the style of the programming language C, we used the sign `!` for negation. At the disjunction, as the sum may produce a value that is outside of the targeted set $\{0, 1\}$, in some cases one may need to use it in the form `!!(a+b)` that transforms the value based on the double negation law to the desired set of values. (In some other languages, e.g., in C++, explicit type conversions can also be used to transform the resulted value back to the set of official truth-values.) The sign '`<=`' may seem to be an arrow $\Leftarrow$ or a horseshoe $\subset$, but in fact none of them is used to represent implication in this orientation. On the other hand, the '==' can be seen as a built in equivalence operator, although it is not highlighted in this way. Moreover, it can be used not only in C, but in many other high-level programming languages, as the equality operation is defined usually also on Boolean values. Thus, in this way, we may be happy that, although it is not underlined in the textbooks and courses, in a usual high-level programming language the programmers may use not only negation, conjunction and disjunction, but also the equivalence operator (usually with the lowest priority, therefore bracketing may be needed if one uses it for this purpose).

Another idea could be to use the connectives in functional form, e.g., to write the logical expressions in prefix form (with or without brackets).

Without using any brackets, the so-called Polish notation of formulae, also called prefix form, can be used. This form is invented by Łukasiewicz to avoid brackets and have a unique way to read and evaluate the formulae [12]. In this writing the operators precede their operands, as we show below.

Examples could be:
EQU AND A IMP B, C XOR B NOT A is representing the formula
$((A \wedge (B \supset C)) \equiv (B \bigoplus \neg A))$.
AND OR NOT A, B IMP A, C is representing $(\neg A \vee B) \wedge (A \supset C)$.

With brackets, the connectives can be interpreted as functions, and in this way, they can be programmed in the programming language itself and they can easily be put to a new logical library as well to include them and allow them to be used by the programmers.

Our last example written in this form is:
AND (OR (NOT (A), B), IMP (A, C))

There is also an advantage of this form over the other without brackets, namely, for the associative operations, e.g., for AND, OR, XOR the programmer may use more than two parameters without any problem, misunderstanding or misinterpretation.

We note that although this type of prefix notation is not widely used in logic and mathematics, it is used in computer science, e.g., in the programming language LISP [27]. The reverse Polish notation, the postfix notation, in which all operands precede the operator is also used in computer science, especially in stack-based programming [20, 26].

4.2. Shortcut for the new connectives

On the one hand, the EUQ and XOR operations are of the form of 2-2 (remember to Table 2 and discussion on the number of the occurrences of the truth values). In this way, we cannot do any pruning or short-cut evaluation technique. By knowing the value of the first part one cannot involve the knowledge of the truth-value of the whole formula in any case, the second part must also be evaluated. This phenomenon is related to the fact that, e.g., in Gentzen sequent calculus by working with a formula having main connective as EQU or XOR, the deduction process is branching and in both branch we need to write and use both immediate subformulae instead of the original formula.

Now, on the other hand, let us take a look on our other five operations (including the original AND and OR) we deal with. All of these are 1-3 or 3-1 forms, meaning that by knowing the truth-value of the first part of the expression, we may know the truth-value of the whole expression (in the fortunate case). These cases are listed below.

At operator	if the first part is	then the whole formula is
AND:	false,	false.
OR:	true,	true.
IMP:	false,	true.
NAND:	false,	true.
NOR:	true,	false.

It is clear to see how the short-cut evaluation works for AND and OR, and in fact, these short-cut evaluations are built in features of the programming languages. On the other hand we can also use the new short-cuts listed in the last three rows of the previous table.

We would like to highlight and discuss one interesting issue here: we have listed 5 operators, but there are only 4 possibilities (to have the truth-value of the first part given and to infer from this to the truth-value of the whole formula). Seemingly, in the table IMP and NAND are similar. Actually, if we can use the short-cut, i.e., the evaluation can be done earlier than all parts of the formula are evaluated, then yes, definitely, they work on the same way. However, in case the short-cut evaluation cannot be used, i.e., the first part is true and we need to evaluate the second part, then their difference appears: if the second part is false, then IMP gives false and NAND gives true. Alternatively, if the second part is also true, then IMP gives true and NAND gives a false value to the whole expression.

We note here again the analogy of the possibilities of the usage of the short-cut techniques and the theorem proving methods Gentzen sequent-calculus and Smullyan tableaux. These methods make a branching at some formulae, and if only the first immediate subformula occurs in a branch, then we can make a cut, e.g., at implication, the whole formula evaluates to true, if the first part is false (and we do not need to check the second subformula).

Finally, we highlight that short-cut evaluations are not only used to make the evaluation faster, but they have safety features as well by allowing to shorten some

parts of the code.

Consider the following conditional statement with operator IMP and with variables `num`, `divisor`:

```
if ( IMP( divisor > 0 , num/divisor > 5 ) ) return 1 else return 0
```

It will return 1, if the actual value of the `divisor` is negative or in the case when `divisor` has value 0 , without checking the fraction in the second part. Further, it returns also 1 if `divisor` is positive and `num/divisor` is larger than 5. Finally, it returns 0 only if `divisor` is positive and `num/divisor` is at most 5. The second part of the implication, including the division by `divisor` is checked only if the first part was evaluated to true, i.e., the value of the `divisor` is not 0, but it is positive. In this way, the possible error of division by zero is avoided by the short cut evaluation technique. Statements of this type are related to the nature of the material implication widely used in formal logic, namely, if the condition part, the first part of the statement has a false truth-value, then does not matter how strange and weird is the second part, the whole statement is evaluated to be true.

In this way it is very similar to the very usual compound condition with integer variables `num`, `divisor` and `result`.

The double, nested condition

```
if ( divisor !=0 ) if ( num/divisor > 5 ) result = num/divisor
```

can be abbreviated to a sole, but compound condition as

```
if ( divisor != 0 && num/divisor > 5 ) result = num/divisor
```

Note that in the programming language C, the first part can be simplified and the condition (that is in the bracket) can be written as

```
(divisor && num/divisor > 5)
```

The fact that we can write the double nested condition in one compound condition without any risk is related to the fact that, for instance in the programming language C (and in other languages), the logic is not exactly the classical Boolean logic, but a kind of 3-valued not commutative logic (see in [16]). As the condition written in

```
( num/divisor > 5 && divisor != 0 )
```

causes a runtime error in case the value of `divisor` is 0, this is not equivalent to our original form. This already leads to us to the next section.

5. Discussion, conclusion and related works

This study can be seen as a follow up study about logic in programming languages, which we have started in [16]. There we have concentrated on how the logical values are computed and what type of ideas and processes are behind the scene.

In this paper, we give some thoughts about which and how many logical operations can and should be used in a high-level programming language. We argued and give hints on how is possible to include not only the widely used three operations of Boolean algebra, but some other well-known and frequently used operations, like the exclusive or, the equivalence and the implication in our programs to make compound logical conditions. They may allow (beginner) programmers to write their conditions in a simpler way, or in the way that is more closely reflected by the condition stated in natural language. We have also studied the possibilities of short-cut evaluations, which can also be seen, on one hand the generalizations of the very closely related alpha and beta pruning techniques of game theory [21, 23] that are also generalised to games with chance nodes (i.e., with some random events) [14] and for other types of operations [2, 3]. Related works are also done by using and analysing similar techniques in the three most-known and most used fuzzy and many-valued logic systems, in the Gödel type logic [5], in the product logic [6] and in the Łukasiewicz-type logics [4, 17].

Acknowledgements. Comments of the anonymous reviewer are gratefully acknowledged.

References

[1] B. H. Arnold: *Logic and Boolean Algebra*, Dover Publications, 2011, p. 144, isbn: 978-0486483856.

[2] R. Basbous, B. Nagy: *Generalized Game Trees and their Evaluation*, in: CogInfoCom 2014: 5th IEEE International Conference on Cognitive Infocommunications, Vietri sul Mare, Italy, IEEE, 2014, pp. 55–60, doi: https://doi.org/10.1109/CogInfoCom.2014.7020518.

[3] R. Basbous, B. Nagy: *Strategies to Fast Evaluation of Tree Networks*, Acta Polytechnica Hungarica 12.6 (2015), pp. 127–148, doi: https://doi.org/10.12700/APH.12.6.2015.6.8, url: http://acta.uni-obuda.hu/Basbous_Nagy_62.pdf.

[4] R. Basbous, B. Nagy, T. Tajti: *Pruning Techniques in Łukasiewicz Logics*, Acta Polytechnica Hungarica v.n (2022), doi: https://doi.org/10.12700/APH..

[5] R. Basbous, B. Nagy, T. Tajti: *Short Circuit Evaluations in Gödel Type Logic*, in: Ravi V., Panigrahi B., Das S., Suganthan P. (eds) Proceedings of the Fifth International Conference on Fuzzy and Neuro Computing (FANCCO - 2015), vol. 415, Advances in Intelligent Systems and Computing (AISC), Springer, Cham., 2015, pp. 119–138, doi: https://doi.org/10.1007/978-3-319-27212-2_10.

[6] R. Basbous, T. Tajti, B. Nagy: *Fast evaluations in product logic various pruning techniques*, in: 2016 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2016, Vancouver, BC, Canada, July 24-29, 2016, IEEE, 2016, pp. 140–147, doi: https://doi.org/10.1109/FUZZ-IEEE.2016.7737680.

[7] J. L. Bell, M. Machover: *A Course in Mathematical Logic*, North Holland, 1977, p. 599, isbn: 978-0080934747.

[8] D. Flanagan, G. M. Novak: *Java-Script: The Definitive Guide*, American Institute of Physics, 1998.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha: *The Java language specification*, Addison-Wesley Professional, 2000.

[10] E. Horowitz: *Fundamentals of Programming Languages*, Springer, Berlin, Heidelberg, 2012, isbn: 9783642967290.

[11] B. Kernighan, D. Ritchie, C. Tondo: *The C Programming Language*, Prentice-Hall software series, Prentice Hall, 1988, isbn: 9789688802052.

[12] J. Łukasiewicz: *Aristotle's Syllogistic from the Standpoint of Modern Formal Logic*, Oxford University Press, 1951, p. 141.

[13] R. J. McEliece, R. B. Ash, C. Ash: *Introduction to discrete mathematics*, English, New York etc.: Random House, 1989, pp. xv + 514, isbn: 0-394-35819-8.

[14] E. Melkó, B. Nagy: *Optimal strategy in games with chance nodes*, Acta Cybern. 18.2 (2007), pp. 171–192, url: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3712.

[15] E. Mendelson: *Theory and problems of Boolean algebra and switching circuits including 150 solved problems*, English, Schaum's Outline Series. New York etc.: McGraw-Hill Book Comp. 213 p. (1970). 1970.

[16] B. Nagy: *Many-valued Logics and the Logic of the C Programming Language*, in: ITI 2005, 27th International Conference on Information Technology Interfaces, Cavtat/Dubrovnik, Croatia, IEEE, 2005, pp. 657–662, doi: https://doi.org/10.1109/ITI.2005.1491200.

[17] B. Nagy, R. Basbous, T. Tajti: *Lazy evaluations in Łukasiewicz type fuzzy logic*, Fuzzy Sets Syst. 376 (2019), pp. 127–151, doi: https://doi.org/10.1016/j.fss.2018.11.014.

[18] K. Pásztor-Varga, M. Várterész: *A matematikai logika alaklmazásszerű tárgyalása (Mathematical logic from application point of view, in Hungarian, textbook)*, Budapest: Panem, 2003.

[19] F. J. Pelletier, N. M. Martin: *Post's Functional Completeness Theorem*, Notre Dame J. Formal Log. 31.3 (1990), pp. 462–475, doi: https://doi.org/10.1305/ndjfl/1093635508.

[20] A. Puntambekar: *Data Structures*, UNICORN Publishing Group, 2020, isbn: 9789333223911.

[21] E. Rich, K. Knight: *Artificial Intelligence*, Artificial Intelligence Series, McGraw-Hill, 1991, isbn: 9780070522633.

[22] G. van Rossum, The Python Development Team: *Python Tutorial (Release 3.6.6rcl)*. CreateSpace Independent Publishing Platform, 2018.

[23] S. Russell, P. Norvig: *Artificial Intelligence: A Modern Approach*, CreateSpace Independent Publishing Platform, 2016, isbn: 9781537600314.

[24] H. M. Sheffer: *A set of five independent postulates for Boolean Algebras, with application to logical constants*, Transactions of the American Mathematical Society 14 (1913), pp. 481–488.

[25] B. Stroustrup: *The C++ programming language*, Pearson Education India, 2000.

[26] M. A. Weiss: *Data Structures and Algorithm Analysis*, Redwood City, CA; Menlo Park, CA; Reading, Ma; New York; Amsterdam; Bonn; Sidney; Singapore; Tokyo; Madrid: The Benjamin/Cummings Publishing Company, Inc., 1995.

[27] P. H. Winston, B. K. P. Horn: *LISP*, United States: Pearson, Jan. 1989.

[28] N. Wirth: *Algorithms & data structures*, Prentice-Hall, Inc., 1985.