From Algorithmic Notation to Program: Using Regular Expression to convert ITB Algorithmic Notation into basic C++ code

Aloisius Adrian Stevan Gunawan - 13523054
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Bandung Institute of Technology, Rd. Ganesha 10 Bandung
E-mail: kremix6767@gmail.com , 13523054@std.stei.itb.ac.id

***Abstract*—Learning ITB's algorithmic notation is not an easy feat, especially for students who has never learned anything about programming. This paper introduces a new way to learn about algorithmic notation, where students could also learn by doing, testing their understanding about it. Due to the large amount of syntax, the implementation would be restricted to just the basic of C++ syntax. This however, serves as a proof of concept that the continuation of this idea is possible, with the aim to create a new way for students to learn.**

Keywords—ITB's algorithmic notation ; learning algorithmic notation

I. Introduction

The ability to design algorithm proves to be fundamental in the world of computer science. The solution to a problem is often written in algorithmic notation, before being implemented into code. This allows programmers to think of a solution without worrying about the syntax. In ITB, students are taught their version of algorithmic notation, and this notation has logic that is similar to C/C++.

This paper explores a new way to learn notation algorithmic by converting it into C/C++ code. To achieve this, string matching techniques, especially regular expressions, is used to design a code that could convert an algorithmic notation into a C/C++ code.

Due to the large amount of syntax, the scope of the converter for this paper is limited to just the basic syntax, such as variable declaration, conditionals, and loops. This paper contains experimental results and could be used as a stepping stone for the other.

II. Theoretical Basis

A. Algorithmic Notation

Algorithmic notation could differ based on where it is taught. In this paper, the algorithmic notation that will be used is the ITB's algorithmic notation. In this section, we will present the basic algorithmic notation that would be implemented for this paper.

1) Variable Declaration

At the start of every algorithmic notation, if the programmer were to use any kind of variables, then they must declare it beforehand, just like in C++.

```
KAMUS
var : <varType>
…
ALGORITMA
… (codes)
```

<varType> is the variable type that would be used. It can only be declared once, and could not be re-declared. This means that a variable cannot hold more than one type. The current available variable types right now are: string, integer, char, bool, and float.

2) Input/Output

Interacting with the user would prove to be difficult without input/output system. This syntax is used to ask for user input, or to output that variable or string into the terminal.

```
input(var)
```

```
output(var)

output("stringText")
```

3) Variable Assignment

Assigning a value to a variable does not require user interaction. This would be done inside the program, so the user could not tamper with it.

```
Var <- <value>
```

The value that would be assigned must be compatible with the variable type that has been declared before.

4) Conditional Statement

Conditional statement is fundamental to programming.

```
if (condition) then
    trueCode
else
    falseCode
```

5) Iterative(Looping) Statement

In ITB's algorithmic notation, there is an iterative statement that is very similar to C++'s for statement. Other than that, there are also three types of looping statement. Those are while-do, do-while, and repeat-until. Each type serves similar yet different purposes. While-do statement would repeat the code written inside, until the condition is no longer fulfilled. Do-while condition is similar to while-do statement, but it ensures the code to execute at least once. On the other hand, repeat-until statement is a little different. It repeats the code until the condition to stop is met.

```
var traversal[start..end]
    code
```

```
while (trueCondition) do:
    code
```

```
do:
    code
while (trueCondition)
```

```
repeat:
    code
until (stopCondition)
```

Considering the purpose of the paper, these algorithmic notation is enough to serve as a proof of the concept. This allows a more focused analysis on the conversion method, rather than making the converter into C++ itself.

B. String Matching

String Matching is a fundamental problem to find a pattern in a text. This would be the core part of the program. To do this, we can use algorithm to find pattern within a text. However, most of them uses brute-force approach to solve string matching problem. This would make it not efficient, especially for very similar pattern that could occur in a text.

C. Regular Expressions (Regex)

Regular expression is a sequence of characters that is used as a search pattern. Although similar to string matching, it serves a different purpose and could also be used to catch text with similar pattern, but not exactly the same,

One of the most important feature in regex is the capturing group. This allows future use of the caught pattern that matches the capture group expression. The portion of the text that were caught could be processed again, and this allows nesting codes to be possible.

Although it might be a little confusing at first, regex is still a powerful tool to consider to use. The following table are the definition for the widely-known regex syntax[1].

TABLE I. REGEX SYNTAX DEFINITION

No	Regex-Syntax definition	
	Syntax	***Definition***
1.	.	Any character except '\n'
2.	^	Start of string
3.	$	End of string
4.	\d, \w, \s	A digit, a word character, or a whitespace
5.	\D, \W, \S	Not a digit, not a word character, or not a whitespace
6.	[abc]	Character a or b or c
7.	[a-z]	Character between(inclusive) a and z
8.	[^abc]	Not a nor b nor c
9.	a|b	a or b
10.	?	Zero or one of the preceding element
11.	*	Zero or more of the preceding element(s)
12.	+	One or more of the preceding element(s)
13.	(expr)	Catches expression
14.	(?:expr)	Does not catch expression
15.	\., *, \?, \+, \(, \), \[, \], \\	Literal character (\. Is dot)

Using Table I., capturing texts in-between specific pattern would be much easier. This would be the foundation to make the converter program from algorithmic notation into C++.

III. METHODOLOGY

This section offers a more detailed explanation on how the program works, and serve as a foundation before implementation and testing. This explanation focuses more on the regex pattern that would be used, and the workflow of the program that allows it to work.

A. General Design

The program works by converting text inside a .notal file into a C++ syntax, then inserting it into output.cpp file. Inside the program, it will take a text-based algorithmic notation, that would then be converted into a C++ syntax file by using several regex notation.

B. Regex Pattern Design

This section will discuss about the regex pattern that would be used to capture algorithmic notation.

1) Variable Declaration

The regex used to catch variable declaration needs to make sure of all variable types that are going to be implemented.

```
(\w+)\s*:\s*(integer|string|char|bool|float)
```

2) Input/Output

Input/Output regex only needs to make sure that it starts with “input” or “output”, followed by brackets.

```
input\((.+)\)
```

```
output\((.+)\)
```

3) Variable Assignment

Assigning a value to a variable might be simple, but there are also case where programmer needs to use ‘+’ or ‘-‘ to do looping or conditional statement.

```
(\w+)\s+<-\s+(.+)
```

4) Conditional Statement

Conditional statement only needs the if and then to be caught. The else would be handled separately.

```
if\s*\((.+)\)\s*then
```

5) Iterative(Looping) Statement

These are the four different types of iterative statement.

```
(\w+)\s+traversal\s+\[(.+)\.\.(.+)\]:
```

```
while\s*\((.+)\)\s*do:
```

```
^do:$

^while\s*\((.+)\)$
```

```
^repeat:$

^until\s*\((.+)\)$
```

IV. Implementation and Testing

A. Testing Environment

The program is made to be able to handle both Windows and Linux environment. This approach aims to make it easier for beginner that wants to learn from windows or linux.

For testing purposes, the program would be run on Windows-Subsystem-Linux(WSL), using Visual Studio Code.

B. Test cases

1) Variable Declaration

TABLE II. Variable Declaration Testcase

Algorithmic Notation	C++
KAMUS var : string var1 : integer var2 : string var3 : char var4 : bool var5 : float var6 : integer ALGORITMA	#include <iostream> #include <string> using *namespace* std; *int* main() { std::string var; *int* var1; std::string var2; *char* var3; *bool* var4; *float* var5; *int* var6; return 0; }

2) Input/Output

TABLE III. Input/Output Testcase

Algorithmic Notation	C++

KAMUS var : string ALGORITMA input(var) output(var)	#include <iostream> #include <string> using *namespace* std; *int* main() { std::string var; *std::cin >> var;* std::cout << var << std::endl; return 0; }

3) Variable Assignment

TABLE IV. INPUT/OUTPUT TESTCASE

Algorithmic Notation	C++
KAMUS var : string var2 : integer ALGORITMA var <- "3" var2 <- 3	#include <iostream> #include <string> using namespace std; int main() { std::string var; int var2; var = "3"; var2 = 3; return 0; }

4) Conditional Statement

TABLE V. IF-ELSE TESTCASE

Algorithmic Notation	C++
KAMUS var : integer ALGORITMA input(var) if (var < 3) then output(var) else var <- var-1 output(var)	#include <iostream> #include <string> using *namespace* std; *int* main() { int var; *std::cin >> var;* if (var < 3) { std::cout << var << std::endl; } else { var = var – 1 std::cout << var << std::endl; } return 0; }

TABLE VI. NESTED IF-ELSE TESTCASE

Algorithmic Notation	C++
KAMUS bensin : integer status : string ALGORITMA input(bensin) if (bensin>0) then bensin <- bensin-1 status <- "berjalan" else if (bensin == 0) then status <- "bensin habis" else status <- "input bensin tidak valid!" output(status)	#include <iostream> #include <string> using namespace std; int main() { int bensin; std::string status; std::cin >> bensin; if (bensin>0) { bensin = bensin-1; status = "berjalan"; } else { if (bensin == 0) { status = "bensin habis"; } else { status = "input bensin tidak valid!"; } } std::cout << status << std::endl; return 0; }

5) Iterative(Looping) Statement

TABLE VII. WHILE-DO TESTCASE

Algorithmic Notation	C++
KAMUS num : integer ALGORITMA input(num) while (num>0) do: num <- num - 1 Output(“Halo”)	#include <iostream> #include <string> using namespace std; int main() { int num; std::cin >> num; while (num>0) { num = num - 1; std::cout << "Halo" << std::endl; } return 0; }

TABLE VIII. DO-WHILE TESTCASE

Algorithmic Notation	C++
KAMUS num : integer ALGORITMA input(num) do: output("Halo") num <- num - 1 while (num>0)	#include <iostream> #include <string> using namespace std; int main() { int num; std::cin >> num; do { std::cout << "Halo" << std::endl; num = num - 1; } while (num>0); return 0; }

TABLE IX. REPEAT-UNTIL TESTCASE

Algorithmic Notation	C++
KAMUS num : integer ALGORITMA input(num) repeat: output("Halo") num <- num - 1 until (num==0)	// Generated C++ code #include <iostream> #include <string> using namespace std; int main() { int num; std::cin >> num; do { std::cout << "Halo" << std::endl; num = num - 1; } while (!(num==0)); return 0; }

V. CONCLUSION

This paper has shown that it is possible to make a converter from ITB's algorithmic notation into C++ code. Using regular expression, this function could recognize algorithmic notation and then understands how to convert it according to the desired syntax.

Albeit functional, this program still has its own limitation and could always be improved. One of the most obvious improvement is that the current implementation is limited to only the basic notation. More advanced syntax like function, lambda, procedure, and more complex data types is not yet handled and thus could be another topic on its own. This paper is just a foundation to show that it is possible to achieve this feat.

APPENDIX

[1] Repository: https://github.com/mimiCrai/LangITB/tree/v1.0

REFERENCES

[1] Khodra, Masayu Leila, "String Matching dengan Regular Expression," Rinaldi Munir Website. [online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf. Accessed: 24-June-2025.

[2] Munir, R., "Pencocokan String (String/Pattern Matching)," Rinaldi Munir Website. [online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf. Accessed: 24-June-2025.

STATEMENT

I declare that this paper is my own work. I have not plagiarized, adapted, nor translated the work of others. If any violation were to be found in the future, I am prepared to accept sanction in accordance with the current applicable regulations.

Bandung, 24 June 2025

Aloisius Adrian Stevan Gunawan, 13523054