Abstract Syntax Tree (AST) and Control Flow Graph (CFG) Construction of Notasi Algoritmik

Irfan Sofyana Putra
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: 13517078@std.stei.itb.ac.id

Satrio Adi Rukmono
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: sar@itb.ac.id

Riza Satria Perdana
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: riza@informatika.org

***Abstract*—Abstract Syntax Tree (AST) and Control Flow Graph (CFG) are program code representations widely used for static analysis. One of the uses of static analysis is for automated grading programming exercises. Notasi Algoritmik is a notation used in our institution for learning programming, including those related to the evaluation of programming exercises. However, the current condition in our institution is that the grading process of programming exercises using Notasi Algoritmik is still done manually, which has an error-prone risk. Before our work, there was no research about how AST and CFG are constructed from Notasi Algoritmik, even though it is possible to develop an automated grader to solve manual assessment in the evaluation of programming learning. Therefore, in this paper, research was conducted to construct the AST and CFG from Notasi Algoritmik, to make developing an automated grader for programming assignments that uses Notasi Algoritmik possible in the future. The first thing that we need to do is to define the grammar of Notasi Algoritmik. After the grammar is defined, we need to do lexical and syntax analysis to construct the AST. AST itself will be used to represent the Notasi Algoritmik in abstract form to construct the CFG. An algorithm created by Tim Teitelbaum (2008) alongside some modifications can be used as a reference to construct the CFG. The research shows that AST is well constructed after two-phase lexical analysis and syntax analysis. Meanwhile, CFG can be constructed using the recursive technique that traverses nodes in the AST. With the result of this research, the development of an automated grader for Notasi Algoritmik can be tried in the future.**

***Index Terms*—Notasi Algoritmik, Lexical Analysis, Syntax Analysis, Abstract Syntax Tree (AST), Control Flow Graph (CFG)**

I. Introduction

Learning programming is an interactive process between learners and educators to learn how to read, write, and test a computer program. Usually, an educator use assignments as a part of the evaluation in the learning process. Our institution adopts a specific notation used in many courses for learning programming. The notation is called Notasi Algoritmik (lit. Algorithmic Notation). This notation is used as a medium of communication between lecturers and students. Notasi Algoritmik is also used in some assignment activities like quizzes, mid-term tests, and final tests.

The assessment process of assignments that used Notasi Algoritmik is carried out manually by lecturers. The process that is still done manually will become a problem if the number of students increases. Therefore, the lecturers need to work harder to do an assessment, but at the same time, it is also error-prone, and in the worst case, has a reduced level of objectivity.

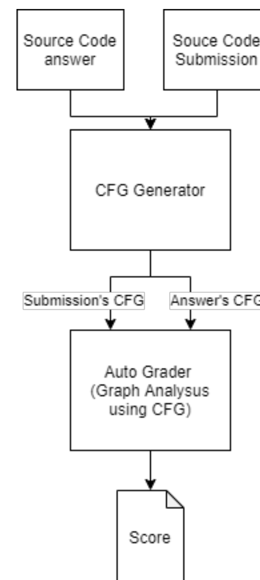

Fig. 1. High Level Overview of Auto Grader Based on CFG

There are already many kinds of research and projects to solve a similar problem. One of the approaches is to develop an automatic grader based on static analysis. The usage of static analysis gives two advantages: (1) it assists educators in assessing a large number of assignments; (2) static analysis can provide feedback and instructions for learners from an expected solution without providing a complete solution to the problem at hand.

Some methods can be used to develop automated graders based on static analysis. One of them is using the graph analysis technique. This method will convert the source code into a CFG then the grader analyses the CFG and produces the score. Figure 1 shows the high-level overview of the automatic grader using CFG [1]. Unfortunately, those methods could not be directly implemented to Notasi Algoritmik because the automatic grader needs an intermediate representation of Notasi Algoritmik. As it stands, no existing method or tool

978-1-6654-9453-3/21/.00 ©2021 IEEE

```
PROGRAM odd_event

KAMUS
  n: integer

ALGORITMA
  input(n)
  if (n mod 2 = 1) then
    output("odd")
  else
    output("even")
```

Fig. 2. Example of Notasi Algoritmik

can be used to represent it.

Meanwhile, AST and CFG are two representations usually used in automatic graders based on static analysis. AST is the abstract representation of the code, and it can be used to derive some valuable metrics and representations, including CFG [2]. However, the absence of methods and tools that can represent Notasi Algoritmik led to an intermediate problem to be solved before developing the automatic grader. This paper explains how to construct AST and CFG from Notasi Algoritmik so that automatic grader development that uses Notasi Algoritmik can be carried out in the future.

II. Notasi Algoritmik

Notasi Algoritmik is a specific standard or notation applied in our institution for some programming courses like programming fundamentals and data structure and algorithm courses. Notasi Algoritmik is adapted from a book called "Schemas Algorithmiques Fondementaux" created by Scholl P.C and Peyrin, J.P in 1988. This notation is mainly used for learning purposes. The notation is required to bridge the diversity and complexity of programming language to focus more on algorithm design than coding. Notasi Algoritmik is used for learning purposes in assignments, quizzes, mid-term tests, and final tests [3].

For this paper, we use Notasi Algoritmik as defined by the book *Draft Diktat Kuliah Dasar Pemrograman*, a textbook for a programming fundamentals course created by Inggriani Liem in 2007. In general, we can say that Notasi Algoritmik is not considered a programming language because it does not have a compiler. However, in the *Diktat*, Notasi Algoritmik has a structure that has many similarities with some programming languages, notably Pascal and Python. Figure 2 shows an example of Notasi Algoritmik, and from there, we can see that Notasi Algoritmik has similarities with some programming languages.

Notasi Algoritmik has core components like common programming languages. In general, Notasi Algoritmik consists of the "KAMUS" part, where we declare elements such as variable, type, and function. There is also the "ALGORITMA" part which is consists of statements that comprise an algorithm. Statements recognised in Notasi Algoritmik include simple statements, branching statements, loop statements, and subprograms (function and procedure). In Notasi Algoritmik, the recursive concept is also recognised. Some statements like branching statements and loop statements can be achieved in several ways. For example, the "if" statement and the "depend-on" statement are two ways to declare branching statements in Notasi Algoritmik. As shown in Figure 2, Notasi Algoritmik also uses the indentation concept to convey the program structure similar to Python Programming Language.

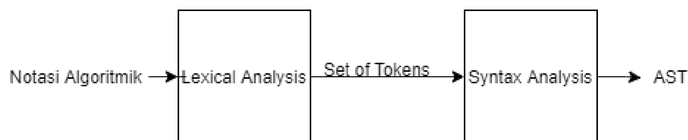

Fig. 3. High level process of AST Construction for Notasi Algoritmik

As previously mentioned, Notasi Algoritmik does not have a compiler. It means that right now, Notasi Algoritmik does not have a formal syntax, although some rules are defined in *Diktat*. This situation requires us to define a standardised syntax of Notasi Algoritmik, or formally, we need to define the context-free grammar of Notasi Algoritmik. This is one of the challenges that we faced to construct AST and CFG of Notasi Algoritmik. Context-free Grammar is likewise commonly abbreviated as CFG, like the control flow graph, therefore to avoid confusion, we will use the term "grammar" to replace "context-free grammar" in the rest of our paper.

To define the grammar of Notasi Algoritmik, first, we need to define what components from Notasi Algoritmik that we will cover. In this paper, we use components related to simple statements (assignment and mathematics statement), all statements related to branching statements, all statements related to loop statements, subprogram (procedure and function), subprogram calling, and recursive statements. We are also defining how to declare constants, data types, and variables. After the components that will be used are defined, for each component, we need to define some possible rules of how we accept the component; this process is also known as finding the production rule of the grammar. We also need to make the rule as small as possible. This is very challenging, and sometimes the resulting grammar is ambiguous. In that case, we need to refine the grammar until it is unambiguous.

III. AST Construction for Notasi Algoritmik

The high-level process of AST construction for Notasi Algoritmik can be seen in Figure 3. Each process will be explained in two following subsections.

A. Lexical Analysis

The first process that needs to be conducted to construct AST is lexical analysis. Lexical analysis is the first phase of the compilation process of a source code. This process involves scanning the source code and breaking it down into a set of tokens. A token is the smallest unit in a language that has a meaningful value. In a programming language, tokens can be grouped into symbols, reserved words, literal values, and identifiers [4].

Before the lexical analysis can be carried out, the tokens used in Notasi Algoritmik need to be defined. We divide the tokens into four categories: (1) symbol or character; (2) reserved word; (3) literal value; and (4) identifier. These tokens can be obtained by looking at and analysing some essential aspects in Notasi Algoritmik. For example, in Notasi Algoritmik, a block called "ALGORITMA" contains a collection of statements representing the algorithm. Hence, the word "ALGORITMA" is defined as a token. The other way to find the tokens is by looking at the statements available in Notasi Algoritmik. For example, in Notasi Algoritmik, there is a "depend-on" statement. Therefore, the word "depend" and "on" needs to be defined as a token.

In Notasi Algoritmik, there are some non-ASCII characters used. For example, the character "←" is used in an assignment statement. To handle those cases, the character encoding used in the lexical analysis is UTF-8. One of the interesting facts in Notasi Algoritmik is the usage of indentation as in the Python programming language. This becomes a challenge in the lexical analysis process. We separate indentations into two tokens representing the algorithm block's start and end to solve this problem. Again, this is adapted from other programming languages. For examples in Pascal, the reserved word "begin" and "end" is used.

As explained before, the lexical analysis process is divided into two phases to classify the indentation into two tokens. The first phase recognises all tokens in Notasi Algoritmik, where an indentation is simply a whitespace token. Then, in the second phase, all those whitespace tokens will be classified into two types of tokens called "INDENT" (similar to "begin" in Pascal) and "DEDENT" (similar to "end" in pascal). The main idea to classify the whitespace token is to keep track of the state of indentation. The state itself stores the number of spaces of the whitespace token. When the whitespace token is read, there are three possible meanings. The first case is when the number of spaces in the token is greater than the last state; then, the read token is an "INDENT" token. The second case is when the number of spaces in the read token equals the last state. In such a case, the token is ignored because it is in the same indentation level as the last state. The third case is when the number of spaces in the read token is less than the last state, then the token is considered a "DEDENT" token, and the state must be adjusted to the number of spaces being read so that there may be an addition of a "DEDENT" token in the final token.

B. Syntax Analysis

Syntax analysis is an advanced lexical analysis phase in compiler development to check the syntax and build a data structure (commonly referred to as a parse tree, abstract syntax tree, or other hierarchical structure) implicit in the input token [5]. After a set of tokens is retrieved from the lexical analysis, the tokens are checked whether the input structure of the Notasi Algoritmik meets the specified grammar or not. If it follows the grammar, then this process produces an AST.

As we already have the grammar from the previous section, the next step that needs to be done is to design how the AST will be constructed. AST design is done for each production rule that exists in the grammar. At this stage, terminal symbols that exist in the production rules but have no significant meaning will be ignored and not included in the AST representation. Each AST constructed from a production rule has one root node and zero or more nodes as child nodes. If the right-hand side of the production rule contains only terminal symbols, then the root node stores information for those terminal symbols. However, if there is at least one non-terminal symbol, then the root node of the AST stores information in the form of a string representing the identity of the grammar rule. In contrast, its child nodes represent the AST of the non-terminal symbol of the production rule.

IV. CFG Construction for Notasi Algoritmik

A control-flow graph (CFG) is a directed graph in which each node represents a basic block, and each edge represents the flow of control between basic blocks. A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. CFG is primarily used in static analysis and compiler applications, as they can accurately represent the flow inside a program unit [6].

Tim Teitelbaum created an algorithm in 2008 to construct CFGs. The algorithm uses the recursive technique by decomposing a statement into a tinier statement until it cannot be decomposed. The resulting CFG is the result of combining short statements that constitute it with a specific rule. Meanwhile, statements that cannot be decomposed have a CFG representation with one node with the same entry and exit blocks, namely the node itself. With this algorithm, AST can be used as a data structure that stores the program input [7].

Every statement in Notasi Algoritmik has a different CFG representation according to the control flow it represents. Therefore, one of the CFG construction tasks is to think and design the graph representation of each statement in Notasi Algoritmik. One of the challenges in CFG construction is handling cases of subprogram calls, in which the AST of each subprogram needs to be saved. Meanwhile, calling a subprogram in Notasi Algoritmik has two possibilities: (1) function is called in an expression. Therefore, whenever the CFG of a statement is constructed, there is a process to check whether there is a function call on any expression contained in that statement. If there is a function call, then the CFG node of the expression is connected to the CFG of the called function. (2) procedure is called in a statement. Therefore, there is a check whether a statement is a procedure call or not. If a statement is a procedure call, then the CFG node of that statement will be connected to the CFG of the called procedure.

Another challenge in CFG construction is handling recursive cases. For that, the first node of the CFG constructed by a subprogram must be saved. Then, every time the CFG of a

```
PROGRAM DETERMINE_NUMBER

KAMUS
  x : integer

ALGORITMA
  input(x)
  if (x > 0) then
    output(x, " is positive number")
  else
    if (x < 0) then
      output(x, " is negative number")
    else
      output(x, " is zero")
```

Fig. 4. Notasi Algoritmik Input for the First Example

subprogram is constructed, there is a first check whether the CFG of the subprogram has been constructed before. If it has previously been constructed, the expression or statement that calls it will be connected to the previously saved node. Meanwhile, if it has never been constructed, the CFG construction process from the subprogram will be carried out, and the first node of the constructed CFG will be saved.

V. Examples

This section will explain some examples of how to generate the AST and CFG from Notasi Algoritmik. Two examples will be used.

A. First Example

The input for the first example can be seen in Figure 4. This example examines how the if-else statement in Notasi Algoritmik is represented in the AST and CFG. The constructed AST and CFG can be seen in Figures 5 and 6, respectively.

As shown in Figure 5, the constructed AST can show each part from the given input properly. The constructed CFG also shows the algorithm part correctly. That can be seen from the fact that there is an edge from the "if $(x > 0)$" node into the "output(x, "is positive number")" node that represents the true condition of the statement. Next, there is also edge from the "if $(x > 0)$" node into the "if $(x < 0)$" node that represents the false condition of the statement. From the given input, we can see that there is also an if-else statement. Therefore, there is an edge from the "if $(x < 0)$" node into the "output(x, " is negative number")" node and the "output(x, " is zero")" node that represents true and false conditions respectively.

One interesting fact about Notasi Algoritmik, especially about the CFG representation of the "if-else" statement, is that there are always two children from the conditional node. Those two represent the true and false conditions. Therefore, we cannot have a chained if-else statement like the "if...elif...elif..." statement in Python, which can have more than two children from the conditional node.

B. Second Example

The input for the second example can be seen in Figure 7. This example examines how some of the loop statements in Notasi Algoritmik (repeat_times and while statement) is represented in the AST and CFG. The constructed AST and CFG can be seen in Figures 8 and 9, respectively.

As shown in Figure 8, the constructed AST can properly show each part that exists in the given input. We can see each element declaration and the statements from the given input in the constructed AST. Meanwhile, the CFG is also appropriately constructed. We can see that all the statements from the given input are available in the CFG.

In this example, we can see there are two loop statements. The first statement is a "repeat-n-times" statement, and the second is a "while" statement. The representation of these two loop statements in the CFG is correct. This is proven for "repeat-n-times" statement because there is an edge from the node into the "$j < - 1$" node that represents the entry point of the loop, and there is an edge from the "$i < - i + 1$" node back into "repeat" node that represents the loop. For the "while" statement, the representation in the CFG is correct proven by there is an edge from the "while $(j <= i)$" node into the "$sum_square < - sum_square + i$" node that represent the entry point of the loop and there is an edge from the "$j < - j + 1$" node going back into "while" node.

VI. Discussion

This research shows that AST can be constructed after going through lexical and syntax analyses. Earlier, several things had to be defined, such as tokens and the grammar of Notasi Algoritmik. The tokens used in the final project were obtained from studies on several aspects, such as the structure of Notasi Algoritmik and the statements contained in Notasi Algoritmik. The lexical analysis process in this research consists of two stages to overcome the problem caused by indentations in Notasi Algoritmik.

In this research, we also define the grammar of Notasi Algoritmik. The grammar of Notasi Algoritmik is quite similar to the Pascal language because there are quite a lot of syntactic similarities between the two. Therefore, when we want to define the grammar for a language, it is helpful to look for language references that are pretty similar to the language being created. The grammar of the reference language can be used as a reference to build the grammar of the language that is being created. Nevertheless, it is still a challenge to define the grammar of Notasi Algoritmik because we still need additional rules or modify the existing rules from a programming language. For example, in Notasi Algoritmik, we need to define the grammar to handle the indentation, which does not exist in Pascal.

Meanwhile, CFG can be constructed using a recursive technique by performing a traversal process on the nodes in the CFG, as suggested by Teitelbaum(2008). However, the algorithm does not explain handling the CFG construction case with a subprogram call and a recursive case. This becomes one of the challenges in the research to modify the algorithm so that subprogram calling cases and recursive cases can be handled. Furthermore, when implementing CFG generation, it was not easy to find literature studies that discussed how CFG

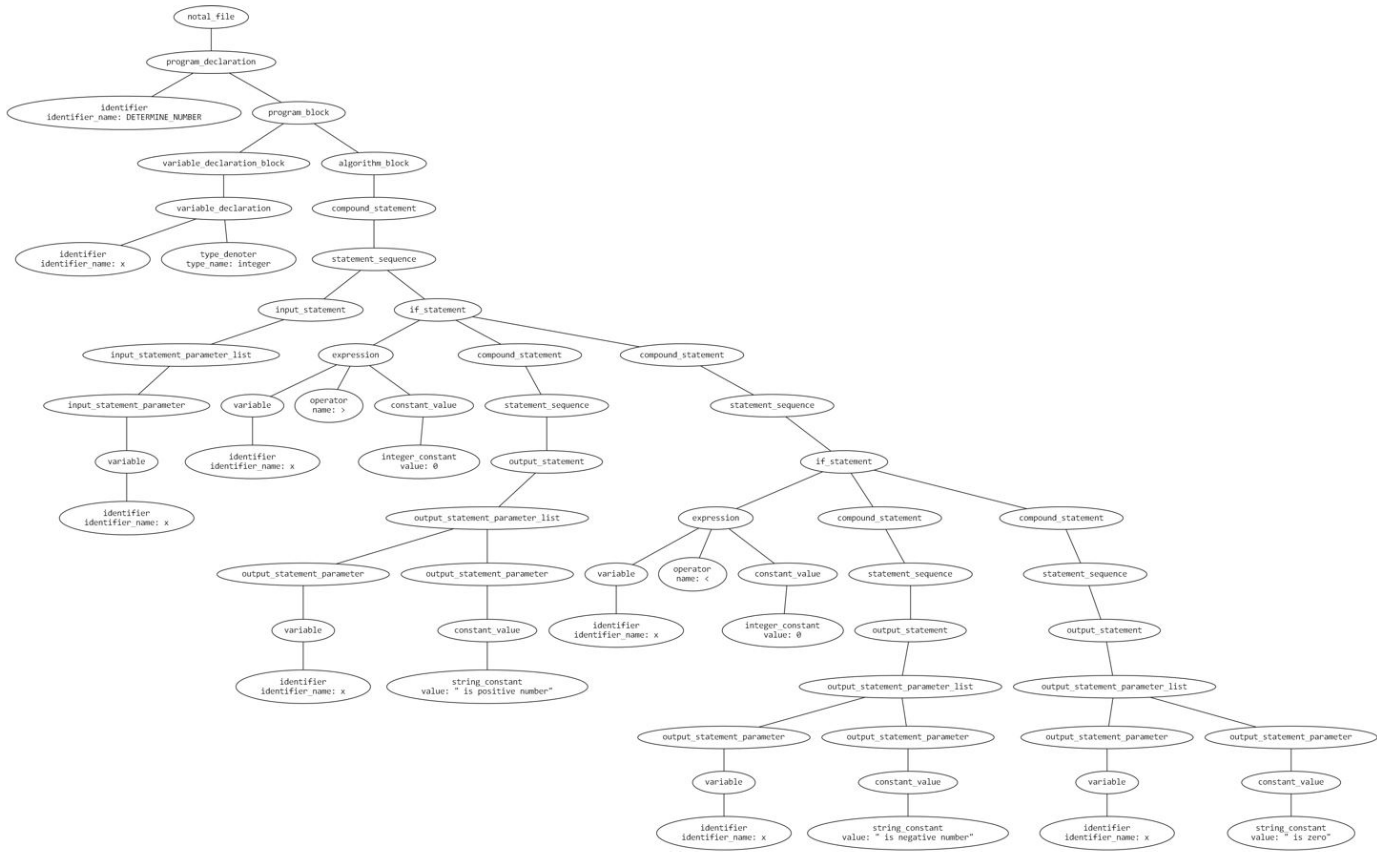

Fig. 5. The Constructed AST for the First Example

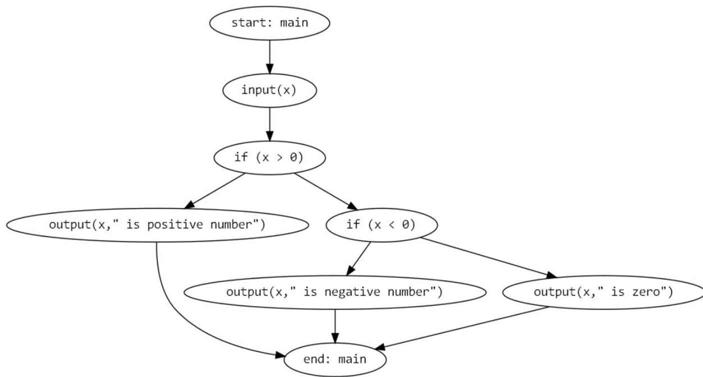

Fig. 6. The Constructed CFG for the First Example

```
PROGRAM SUM_SQUARE
{Compute 1^2 + 2^2 + ... + n^2}

KAMUS
  sum_square: integer
  n, i, j: integer

ALGORITMA
  input(n)
  sum_square <- 0
  i <- 1
  repeat n times
    j <- 1
    while (j <= i) do
      sum_square <- sum_square + i
      j <- j + 1
    i <- i + 1
  output(sum_square)
```

Fig. 7. Notasi Algoritmik Input for the Second Example

generation was carried out. Therefore, the author also hopes that this research can be helpful for future research or projects requiring algorithms to generate CFG.

The following improvements can be made from this research: i) add parts of Notasi Algoritmik that we left undefined in our research to construct the AST and CFG for statements such as sequential file processing, ii) perform semantic analysis after the syntax analysis process to detect semantic error from the Notasi Algoritmik, and iii) include additional helpful information within the CFG node, such as the line number of the statement.

The general strategies to construct the AST and CFG provided in this research also can be tried in other programming languages or even pseudocode as long as we have the rule or the grammar of the language. However, in this research, we use Notasi Algoritmik because it is used in our institution, and we have a guidebook on how to use it. Other than that, we use Notasi Algoritmik to ensure that the result of this research

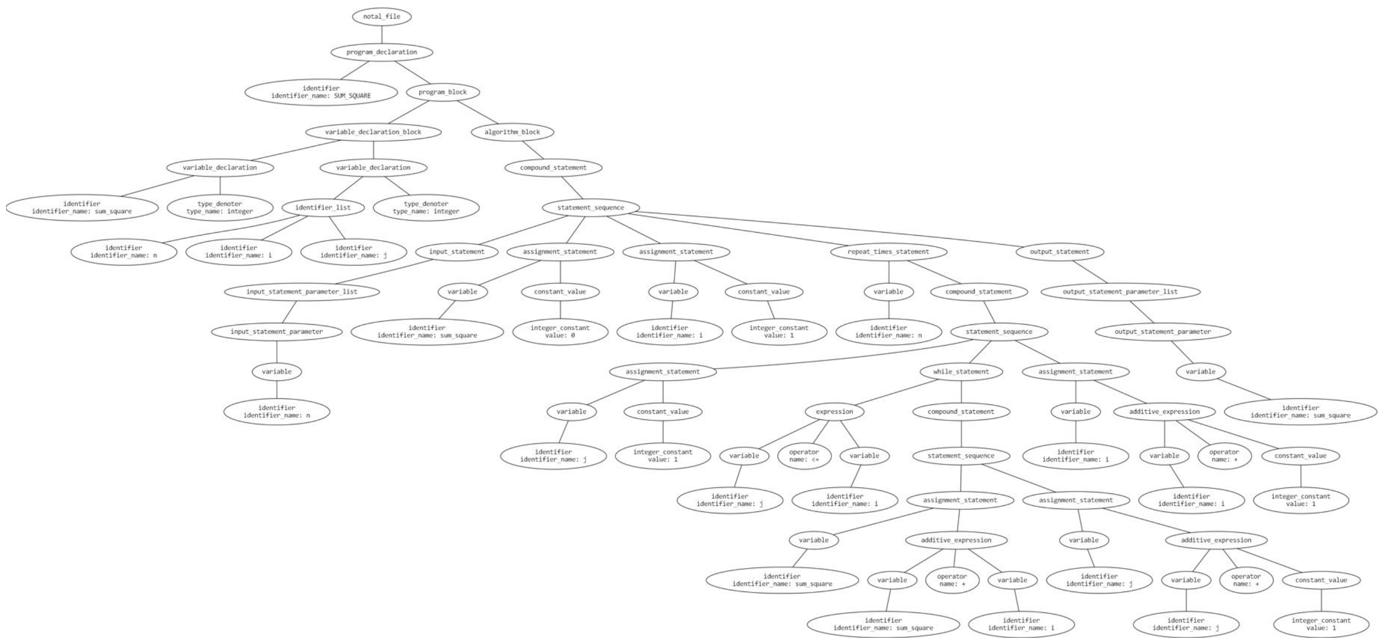

Fig. 8. The Constructed AST for the Second Example

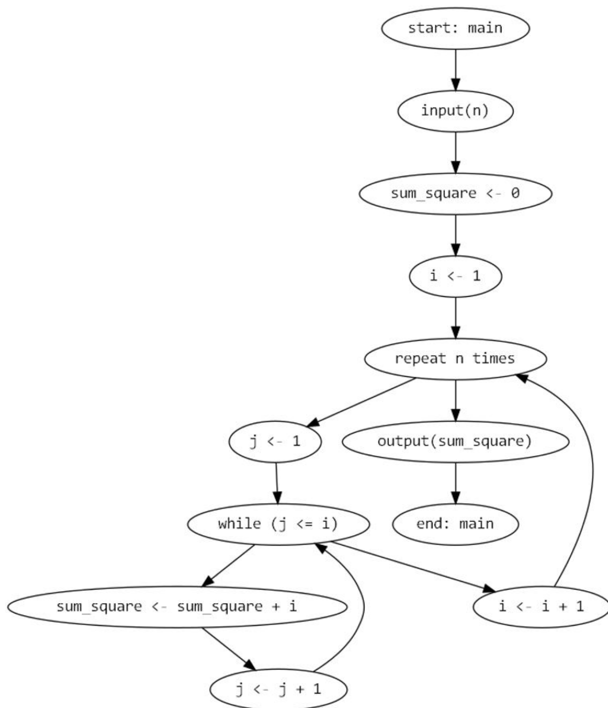

Fig. 9. The Constructed CFG for the Second Example

can be used as a baseline to develop an automated grader for programming exercises that used Notasi Algoritmik in our institution.

VII. Conclusion

From this research, we can conclude that the AST of Notasi Algoritmik can be constructed through two stages of lexical analysis and syntax analysis. The first stage of the lexical analysis process reads the indent as a whitespace token. Then in the second stage, the whitespace token is classified as an ”INDENT” token if it is an indent that represents the beginning of a block or a ”DEDENT” token if it is an indent that represents the end of a block. In this research, the grammar of Notasi Algoritmik is also defined, which has quite many similarities with the grammar of the Pascal programming language.

Meanwhile, the CFG of Notasi Algoritmik can be constructed with Teitelbaum's algorithm and with some modifications to handle the subprogram and recursive calling. This technique performs the traversal process on the nodes in the AST that has been constructed before. The results obtained from this research can be a foundation to start automatic grader development that uses Notasi Algoritmik.

References

[1] Sendjaja, Kevin. (2021). Evaluasi Tugas Pemrograman dengan Control Flow Graph. Skripsi. Bandung: Institut Teknologi Bandung.

[2] Fischer, G., lusiardi, J., & Gudenberg, J. W. (2007). Abstract Syntax Trees and their Role in Model Driven Software. International Conference on Software Engineering Advances (ICSEA 2007). Cap Esterel: IEEE

[3] Liem, I. (2007). Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural). Bandung: Kelompok Keahlian Rekayasa Perangkat Lunak dan Data STEI ITB.

[4] Farhanaaz, & V, S. (2016). An exploration on lexical analysis. International Conference on Electrical, Electronics, and Optimisation Techniques (ICEEOT). Chennai: IEEE.

[5] Mulik, S., Shinde, S., & Kapase, S. (2011). Comparison of Parsing Techniques For Formal Languages. International Journal on Computer Science and Engineering (IJCSE).

[6] Harrold, M. J., Rothermel, G., & Orso, A. (2003). Representation and analysis of software. Symposium on Night-and Shiftwork

[7] Teitelbaum, T. (2008). Introduction to Compilers. New York: Cornell University.