SOFTWARE TOOL FOR TRANSLATING PSEUDOCODE TO A PROGRAMMING LANGUAGE

Amal M R , Jamsheedh C V and Linda Sara Mathew

Department of Computer Science and Engineering, M.A College of Engineering, Kothamangalam, Kerala, India

ABSTRACT

Pseudocode is an artificial and informal language that helps programmers to develop algorithms. In this paper a software tool is described, for translating the pseudocode into a particular programming language. This tool takes the pseudocode as input, compiles it and translates it to a concrete programming language. The scope of the tool is very much wide as we can extend it to a universal programming tool which produces any of the specified programming language from a given pseudocode. Here we present the solution for translating the pseudocode to a programming language by implementing the stages of a compiler.

KEYWORDS

Compiler, Pseudocode to Source code, Pseudocode Compiler, c, c++

1. INTRODUCTION

Generally a compiler is treated as a single unit that maps a source code into a semantically equivalent target program [1]. If we are analysing a little, we see that there are mainly two phases in this mapping: analysis and synthesis. The analysis phase breaks up the source code into constituent parts and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source code. If the analysis phase detects that the source code is either syntactically weak or semantically unsound, then it must provide informative messages. The analysis phase also collects information about the source code and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis phase. The synthesis phase constructs the target program from the intermediate representation and the information in the symbol table [2], [3]. The analysis phase is often called the front end of the compiler; the synthesis phase is the back end.

Compilation process operates as a sequence of phases, each of which transforms one representation of the source program to another. Compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation; so that the backend can produce a better target program than it would have otherwise produced from an un-optimized intermediate representation.

In this paper, it is intended to produce a user specified programming language from pseudocode. This tool requires a single pseudocode and it can produce the programming language that is specified by the user. Its significance is that it can be extended to a universal programming tool that can produce any specified programming language from pseudocode.

2. COMPILING PSEUDOCODE

The process of compiling the pseudocode consists of certain analysis and operations that has to be performed on it.

2.1. LEXICAL ANALYSER

The Lexical Analyser module analyses the pseudocode submitted by the user by using the transition analysis. The keywords, Identifiers and tokens are identified from the given input.

2.2. SYNTAX ANALYSER

The Syntax Analyser module creates the Context Free Grammar from the pseudocode submitted .The resultant grammar is then used for creation of the parse tree. Then pre order traversal is done to obtain the meaning of the syntax.

2.3. SEMANTIC ANALYSER

The semantic Analyser uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information.

2.4. INTERMEDIATE CODE GENERATOR

In this module, an intermediate code is generated from the input pseudocode. The generated intermediate code is that code which is used for the conversion to any other languages.

2.5. INTERMEDIATE CODE OPTIMIZER

In this module, code optimization is applied on the intermediate code generated. The generated optimized intermediate code is that code which is used for mapping to the concrete languages.

2.6. CODE GENERATOR

The optimized intermediate code is converted into the required programming language in this module. The result might be obtained in the language selected by the user.

2.7. LIBRARY FILE MANAGER

In this module the administrator manages the library files of the target language and also manipulates files in the library package.

3. PROBLEM STATEMENT

3.1. INTERPRET THE PSEUDOCODE

The main task is to identify and interpret the pseudocode given by the user. Each user has his own style of presentation, variation in using keywords or terms (E.g.: - Sum, Add, etc. to find sum of two numbers), structure of sentence, etc. So initial task is make the tool capable of develop the tool is to interpret and identify the right meaning of each line of the pseudocode.

3.2. Translate The Pseudocode Into Programming Language

The second step involves the translation of the interpreted pseudocode into programming language. User can specify output in any of the available programming languages. So the tool must be able to support all the available programming language features (in this paper we are concentrating on C and C++ only). That is it must support the object oriented concepts, forms and so on.

4. Methodology

4.1. Lexical Analysis

The first phase of the software tool is called lexical analysis or scanning. The lexical analyser reads the stream of characters making up the pseudocode and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyser produces as output a token of the form {token- name, attribute-value} that it passes on to the subsequent phase, syntax analysis. In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry 'is needed for semantic analysis and code generation. For example, suppose a source program contains the declare statement[1],[2].

Declare an integer variable called sum#   (1.1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyser:

1. *Declare a***,** is a lexeme that would be mapped into a token (Declare, 59), where Declare is a keyword and 59 points to the symbol table entry for position.
2. *Integer,* is a lexeme that would be mapped into a token (Integer, 112), where Integer is a keyword and 112 points to the symbol table entry for position.
3. *Variable,* is a lexeme that would be mapped into a token (Variable, 179), where Variable is a keyword and 179 points to the symbol table entry for position.
4. *Called,* is a lexeme that would be mapped into a token (Called, 340), where Called is a keyword and 340 points to the symbol table entry for position.
5. *Sum,*is a lexeme that would be mapped into a token (sum, 740), where sum is an identifier and 740 points to the symbol table entry for position.
   (Blanks separating the lexemes would be discarded by the lexical analyser.)

4.2. Syntax Analysis

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyser to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation [12], [13]. The syntax of programming language constructs can be specified by context-free grammars or BNF (Backus-Naur Form) notation; Grammars offer significant benefits for both language designers and compiler writers.A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language. From certain classes of grammars, we can construct automatically an efficient parser that determines the syntactic structure of a source program. As a side benefit, the parser-construction process can reveal syntactic ambiguity and trouble spots that might have slipped through the initial design phase of a language. The structure imparted to a language by a properly designed grammar is useful for translating source programs into correct object code and for detecting errors. A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. These new constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

4.3. CONTEXT-FREE GRAMMARS

Grammars were introduced to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable 'Stmt' to denote statements and variable 'expr' to denote expressions, In particular, the notion of derivations is very helpful for discussing the order in which productions are applied during parsing. The Formal Definition of a Context-Free Grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyser.
2. Non terminals are syntactic variables that denote sets of strings. The set of string denoted by non-terminals helps to define the language generated by the grammar. Non terminals impose a hierarchical structure on the language that is the key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. Conventionally, the productions for the start symbol are listed first.
4. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of:

a) A nonterminal called the head or left side of the production; this production defines some of the strings denoted by the head.
b) The symbol --+. Sometimes:: = has been used in place of the arrow.
c) A body or right side consisting of zero or more terminals and non-terminals.

The Context Free Grammar generated from 1.1 by the Software tool is

Stmt -> declare_an <DataType> variable Called <Identifier>   (1.2)
DataType->integer
Identifier->sum

4.4. SEMANTIC ANALYSIS

The semantic analyser uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array .The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number. The Parse Tree generated from 1.2 by the Software tool is by array representation as follows,

The pre order traversal:

Stmt-> declare_an DataType integer variable Called Identifier sum (1.3)

4.5. Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine[10],[11]. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine. In our software tool intermediate code is generated to convert the code to various languages from single pseudocode.

The intermediate code for 1.3 is as follows:*149 i780 300o (1.4)*

4.6. Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine Code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

The resultant program code for 1.4 is as follows:*int sum; (1.5)*

5. Schema Description

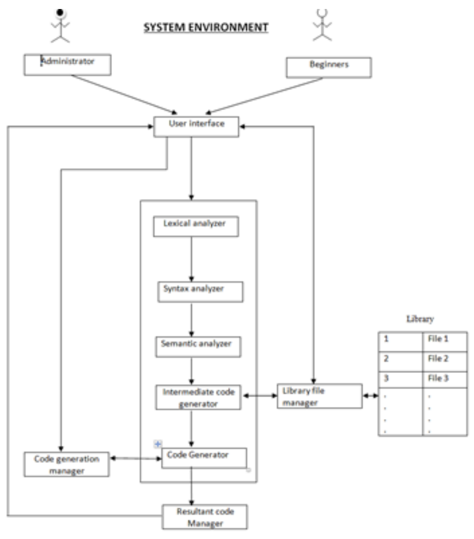

Fig:-1 System environment of the proposed software tool

The proposed software tool consists of several modules which are used to process the input

4.1. INPUT DESIGN

This is a process of converting user inputs into computer based formats. The data is fed into system using simple interactive forms. The forms have been supplied with messages so that user can enter data without facing any difficulty. The data is validated wherever it requires in the project. This ensures that only the correct data have been incorporated into the system. It also includes determining the recording media methods of input, speed of capture and entry into the system. The input design or user interface design is very important for any application. The interface design defines how the software communicates with in itself, to system that interpreted with it and with humans who use.

The main objectives that guide input design are as follows:

User friendly code editor- Providing line numbers and shaded graphical rows to easily identify each line of code.

Arise that lead to processing delays- Input is designed so that it does not lead to bottlenecks and thus avoid processing delays.

Dynamic check for errors in data-errors in data can lead to delays. Input design should be such that the data being entered should be free from error to the maximum possible limit.

Avoided extra steps-more the number of steps more is the chance of an error. Thus the number of steps is kept to a minimum possible.

Kept the process simple-the process should be kept as simple as possible to avoid errors.

4.2. OUTPUT DESIGN

A quality output is one, which meets the requirements of the end user and presents the information clearly. In the output design, it is determined how the information is to be displayed for immediate need and also the hard copy output. The output design should be understandable to the user and it must offer great convenience. The output of the proposed software tool is designed as opening the text file containing the translated code.

The main objectives that guide the output design are as follows:

a. User can copy the code and run it in any of the IDE available.
b. Since the output is written in a standard text file, the user can directly call the file in the host program.
c. User can add some more code to the existing output and edit it easily.

4.3. DATA STRUCTURES USED

4.3.1. DATA BANK, TOKEN AND TOKEN ID

Here a table with all the tokens and there ID codes used in lexical analysis are tabulated. These tokens and there IDs are stored using Hash Table while implementing.

4.3.2. Library File, for a particular Programming language:

This is the data in the library file stored in the Library in the software tool. The file consists of all the keywords and header files in the language.
For example:-

Library File:For 'C':-

It includes the data in the library file stored in the Library of the software tool for all the keywords and header files in the language 'C'.

6. Experiment Analysis

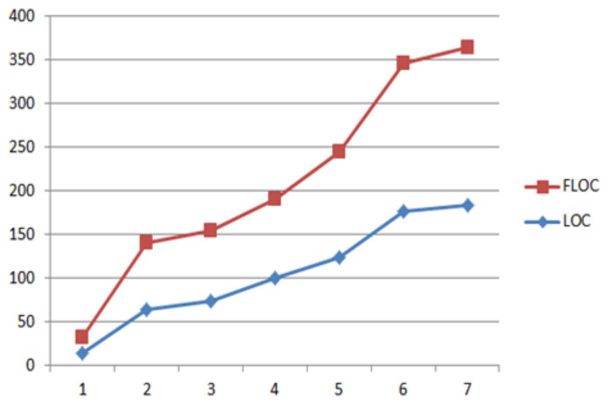

Figure 1. Comparison of Final Lines of Code(FLOC)and Lines of code(LOC)

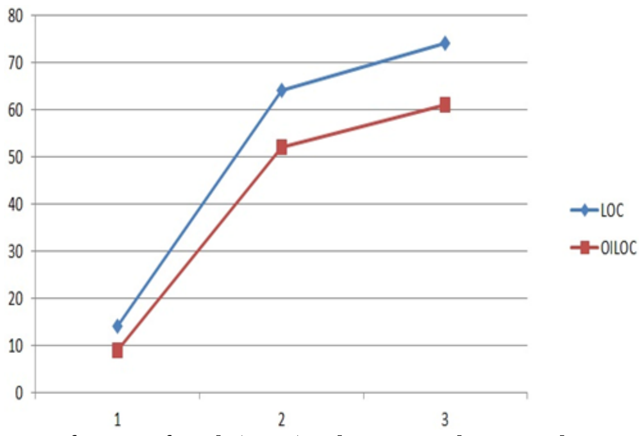

Figure 2. Comparison of Lines of Code(LOC)and Optimized Intermediate Lines of code(LOC)

***Analysis*:** The graphs are plotted with the Lines of Code (LOC) against the number of experiments. From the plots, it is clear that the initial LOC of the pseudocode given by the user is reduced proportionally in the optimized intermediate generated codes (OILOC).Then the final

LOC of the generated code is comparatively larger in proportion of the LOC of the pseudocode (see fig. 1).This measures indicates the efficiency of the tool in the generation of the code of the specified programming language. This measures depends on the efficiency and compatibility of the new developed tool.

7. CONCLUSIONS

This paper is focused on providing a user friendly environment for the beginners in programming. They can easily build a code in specified language from a pseudocode without considering the factor of knowledge about the syntax of the particular language. A beginner level programmer familiar with writing pseudocode can implement his logic in any particular language, simply by using this tool. The main advantage of this tool is that, user can build program code in any language from a single pseudocode. For a beginner in programming, it is difficult to learn the syntax of a particular language at the very first time. The user can implement his logic in a pseudocode and the pseudocode required for this software tool requires simple syntax. A formulated pseudocode is simple to be generated by a beginner. Then this pseudocode is simply submitted to the text area in our tool. Then specify the language of the output required. Then after processing he will get the resultant programming code in a file, which is much simpler with user friendly interface. Then the resultant code can be executed in its programming platform. The library files in the software tool can be manipulated to add more syntaxes into the database. Future versions can be built with giving support to more languages. We can develop this software tool to a universal programming tool, which can be used to build programming code in any of the programming language, from simple, single pseudocode generated by the user. It reduces the user's overhead to be known about the syntax of various languages.

REFERENCES

[1] G Alfred V.Aho,Monica S.Lam,Ravi Sethi,Jeffrey D.Ullman,Compilers Principles,Techniques and Tools, Second edition 2007

[2] Allen Holub, "Compiler Design in C", Prentice Hall of India, 1993.

[3] Kenneth C Louden, "Compiler Construction Principles and Practice",Cenage Learning Indian Edition..

[4] V Raghavan, "Priniples of Compiler Design",Tata McGraw Hill,India, 2010

[5] Arthur B. Pyster, "Compiler design and construction: tools and techniques with C and Pascal", 2nd Edition, Van Nostrand Reinhold Co. New York, NY, USA.

[6] D M Dhamdhare, System programming and operating system, Tata McGraw Hill & Company

[7] Tremblay and Sorenson, The Theory and Practice of Compiler Writing - Tata McGraw Hill & Company.

[8] Steven S. Muchnick, "Advanced Compiler Design & plementation", Morgan Kaufmann Pulishers, 2000.

[9] Dhamdhere, "System Programming & Operating Systems", 2nd edition, Tata McGraw Hill, India.

[10] John Hopcroft, Rajeev Motwani & Jeffry Ullman: Introduction to Automata Theory Languages & Computation , Pearson Edn.

[11] Raymond Greenlaw,H. James Hoover, Fundamentals of Theory of Computation,Elsevier,Gurgaon,Haryana,2009

[12] John C Martin, Introducing to languages and The Theory of Computation, 3rd Edition, Tata McGraw Hill,New Delhi,2010

[13] Kamala Krithivasan, Rama R, Introduction to Formal Languages,Automata Theory and Computation, Pearson Education Asia,2009.

[14] Rajesh K. Shukla, Theory of Computation, Cengage Learning, New Delhi,2009.

[15] K V N Sunitha, N Kalyani: Formal Languages and Automata Theory, Tata McGraw Hill,New Delhi,2010.

[16] S. P. Eugene Xavier, Theory of Automata Formal Language &Computation,New Age International, New Delhi ,2004.

[17] K.L.P. Mishra, N. Chandrashekharan , Theory of Computer Science , Prentice Hall of India.

[18] Michael Sipser, Introduction to the Theory of Computation, Cengage Learning,New Delhi,2007.
[19] Harry R Lewis, Christos H Papadimitriou, Elements of the theory of computation, Pearson Education Asia.
[20] Bernard M Moret: The Theory of Computation, Pearson Education.
[21] Rajendra Kumar,Theory of Automata Language & Computation,Tata McGraw Hill,New Delhi,2010.
[22] Wayne Goddard, Introducing Theory of Computation, Jones & Bartlett India,New Delhi2010.

AUTHORS

Amal M R is currently pursuing M.Tech in Computer Science and Engineering Mar Athanasius College of Engineering, Kothamangalam. He completed his B.Tech from Lourdes Matha College of Science and Technology Thiruvananthapuram. His areas of research are Compiler and Cloud Computing.

Jamsheedh C V is currently pursuing M.Tech in Computer Science and Engineering in Mar Athanasius College of Engineering, Kothamangalam. He completed his B.Tech from Govt. Engineering College Idukki. His areas of research are Networking and Cloud Computing

Linda Sara Mathew received her B.Tech degree in Computer Science and Engineering from Mar Athanasius College of Engineering ,Kothamangaam,Kerala in 2002 and ME degree in Computer Science And Engineering Coimbatore in 2011. She is currently, working as Assistant Professor, with Department of Computer Science and Engineering in Mar Athanasius College of Engineering, Kothamangalam and has a teaching experience of 8 years. Her area of interests include digital signal processing, Image Processing and Soft Computing.