AUTO GENERATION OF C CODE

A MINI PROJECT REPORT

18CSC305J - ARTIFICIAL INTELLIGENCE

Submitted by

ANUSHKA LONDHE(RA2011003010463)

JOGESWAR PANIGRAHI(RA2011003010470)

Under the guidance of

DR. R Jeya

(Assistant Professor, Department of Computing Technologies)

in partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR-603203

MAY 2023

COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

S.R.M.NAGAR, KATTANKULATHUR – 603203

BONAFIDE CERTIFICATE

Certified that this project report **"Auto Generation of C Code"** is the Bonafide work of **"ANUSHKA LONDHE (RA2011003010463) and JOGESWAR PANIGRAHI (RA2011003010470)"** of III Year/VI Sem B. Tech (CSE) who carried out the mini project work under my supervision for the course **18CSC304J- Compiler Design** in SRM Institute of Science and Technology during the academic year 2022-2023(Even Sem).

(Signature)	**(Head Of Department)**
Dr.R.Jeya	Dr. M. Pushpalatha
Assistant Professor	Head and Professor
Dept of Computing Technologies	Dept of Computing Technologies
SRM Institute of Science and Technology	SRM Institute of Science and Technology

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honorable **Vice Chancellor, Dr. C. Muthamizhchelvan,** for being the beacon in all our endeavors.

We would like to express my warmth of gratitude to our **Registrar, Dr. S. Ponnusamy,** for his encouragement.

We express our profound gratitude to our **Dean (College of Engineering and Technology), Dr. T. V. Gopal**, for bringing out novelty in all executions.

We would like to express my heartfelt thanks to the **Chairperson, School of Computing, Dr. Revathi Venkataraman**, for imparting confidence to complete my course project.

We extend my gratitude to our **HoD, Dr. M. Pushpalatha, Head Professor, Department of Computing Technologies.**

We are highly thankful to our **course project faculty, Dr.R.Jeya, Assistant Professor, Department of Computing Technologies** for her assistance, timely suggestion and guidance throughout the duration of this course project and my Departmental colleagues for their support.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

Aim: To implement a system for the Auto-generation of C Code.

ABSTRACT

The project aims to develop a system for automatic C code generation from pseudocode. Pseudocode is a widely used technique for describing algorithms in a high-level language that is easy for humans to understand. However, converting pseudocode into actual code can be a time-consuming and error-prone task, especially for complex algorithms.

The proposed system will use natural language processing techniques to parse pseudocode and generate corresponding code in a target programming language. The system will be trained on a large corpus of pseudocode examples and corresponding code to learn patterns and syntax rules that can be used to generate accurate and efficient code.

The system will support multiple programming languages and will be designed to generate code that adheres to best practices and coding standards. The system will also provide feedback to users to help them refine their pseudocode and improve the quality of the generated code.

The proposed system has the potential to significantly reduce the time and effort required to convert pseudocode into actual code and to improve the accuracy and efficiency of the resulting code.

TABLE OF CONTENT

EXISTING SYSTEMS AND THEIR DRAWBACKS

Auto code generators are software tools that automatically generate code based on a higher-level representation of a program, such as pseudocode. Here is some information on existing auto-code generators from pseudocode systems and their drawbacks:

1. **Matlab's "codegen" tool**: Matlab's codegen tool is a widely-used auto code generator that allows users to generate C, C++, or MEX code from their Matlab functions. However, the generated code can be hard to read and understand, making it difficult to modify or debug.

2. **Simulink's "Embedded Coder":** Simulink's Embedded Coder is a popular auto code generator that allows users to generate C code from their Simulink models. However, it can be challenging to configure and may not always produce the most optimized code.

3. **Modelio's "Modelio 3.8.1":** Modelio is a UML modeling tool that can automatically generate code in a variety of languages, including Java, C++, and Python. However, its code generation capabilities can be limited compared to other tools, and it may not always produce the most efficient code.

4. **Rational Software Architect's "Code Generation Framework":** Rational Software Architect's Code Generation Framework is a powerful auto code generator that can generate code in a variety of languages, including Java and C++. However, it can be challenging to set up and configure, and it may require a significant amount of customization to generate code that meets specific requirements.

Overall, while auto code generators can be useful for simplifying the development process, they may also have drawbacks such as producing less readable or less optimized code. It's important to carefully evaluate the capabilities and limitations of each tool before selecting one for a particular project.

PROPOSED METHODOLOGY

The methodology for building an auto code generator in C from given pseudocode can be broken down into several steps:

1. **Define the pseudocode language:** The first step is to define the language of the pseudocode that will be input into the system. This involves defining the syntax and grammar rules for the language, as well as the semantics of the language constructs.

2. **Create a lexical analyzer:** Once the language has been defined, a lexical analyzer needs to be created to tokenize the input pseudocode. This can be done using a tool such as Lex, which generates a lexical analyzer based on a set of regular expressions that define the language's tokens.

3. **Implement a parser:** After the input has been tokenized, a parser is needed to analyze the input and create a representation of the program structure. The parser should use the language grammar rules to construct an abstract syntax tree (AST) that represents the structure of the program.

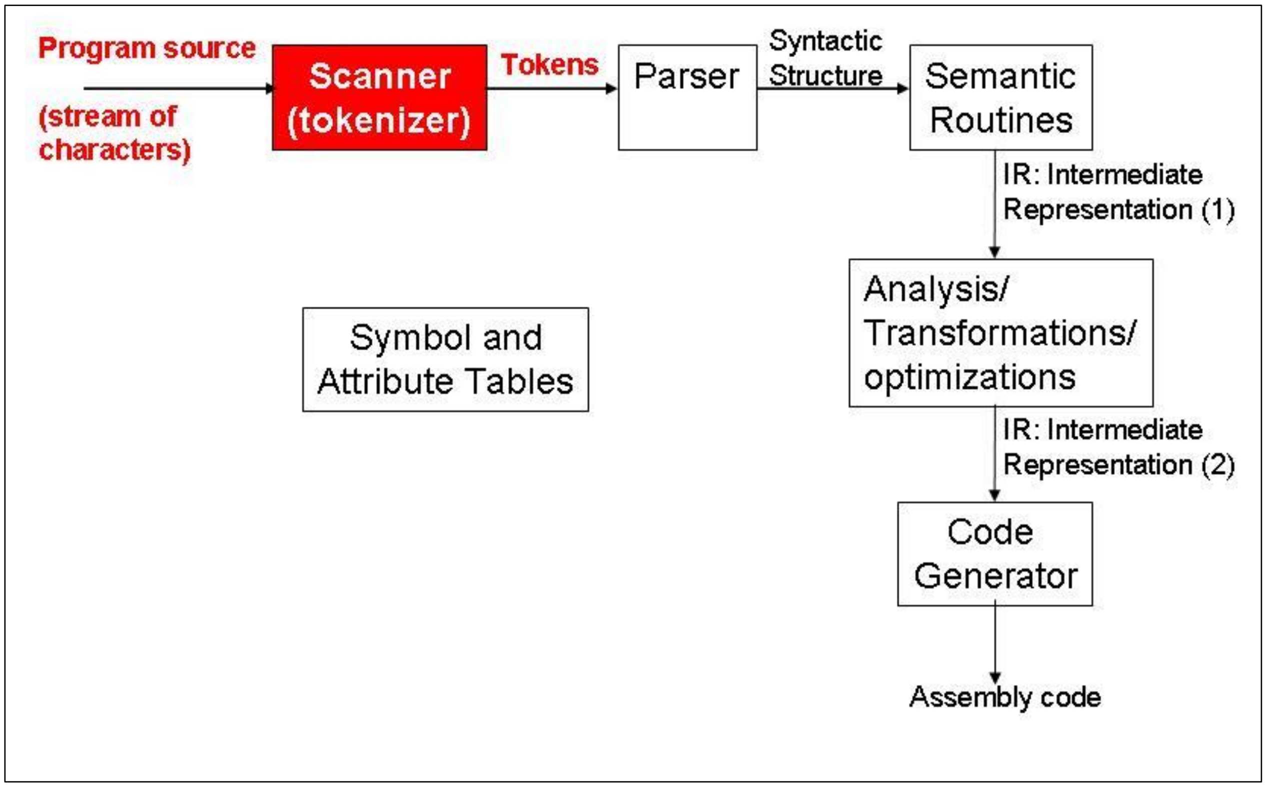
4. **Implement a code generator:** Once an AST has been constructed, a code generator is needed to translate the AST into C code. The code generator should traverse the AST and emit C code for each node in the tree. The code generator may also need to perform some optimizations or transformations to the code to ensure that it is efficient and correct.

5. **Generate C code:** Once the code generator has completed its work, the resulting C code is output to the user. This could be done by generating a text file containing the C code or by displaying it in the user interface.

6. **Integrate with a C compiler:** Finally, the generated C code can be compiled and executed using a C compiler such as GCC or Clang. The auto code generation tool could be designed to integrate with these compilers, automatically invoking them to compile the generated code and produce an executable binary.

Throughout the development process, it is important to test the system with a variety of input pseudocode programs to ensure that it is working correctly. The testing process should involve a combination of unit tests and integration tests to verify that each component of the system is functioning as expected, and that the system as a whole is capable of generating efficient and correct C code from pseudocode inputs.

ARCHITECTURE AND BLOCK DIAGRAM


Program source
(stream of characters)
Scanner (tokenizer)
Tokens
Parser
Syntactic Structure
Semantic Routines
IR: Intermediate Representation (1)
Symbol and Attribute Tables
Analysis/ Transformations/ optimizations
IR: Intermediate Representation (2)
Code Generator
Assembly code

1. **User interface:** The user interface is the part of the system that the user interacts with. It might consist of a graphical user interface (GUI) or a command-line interface (CLI). The user inputs the pseudocode that they want to convert into C code through this interface.

2. **Pseudocode parser:** The pseudocode parser is responsible for analyzing the input pseudocode and generating an internal representation of the program structure. This could be done using a parser generator tool such as Lex, which generates a lexical analyzer that tokenizes the input, and a parser that

constructs an abstract syntax tree (AST) based on the grammar rules defined for the pseudocode language.

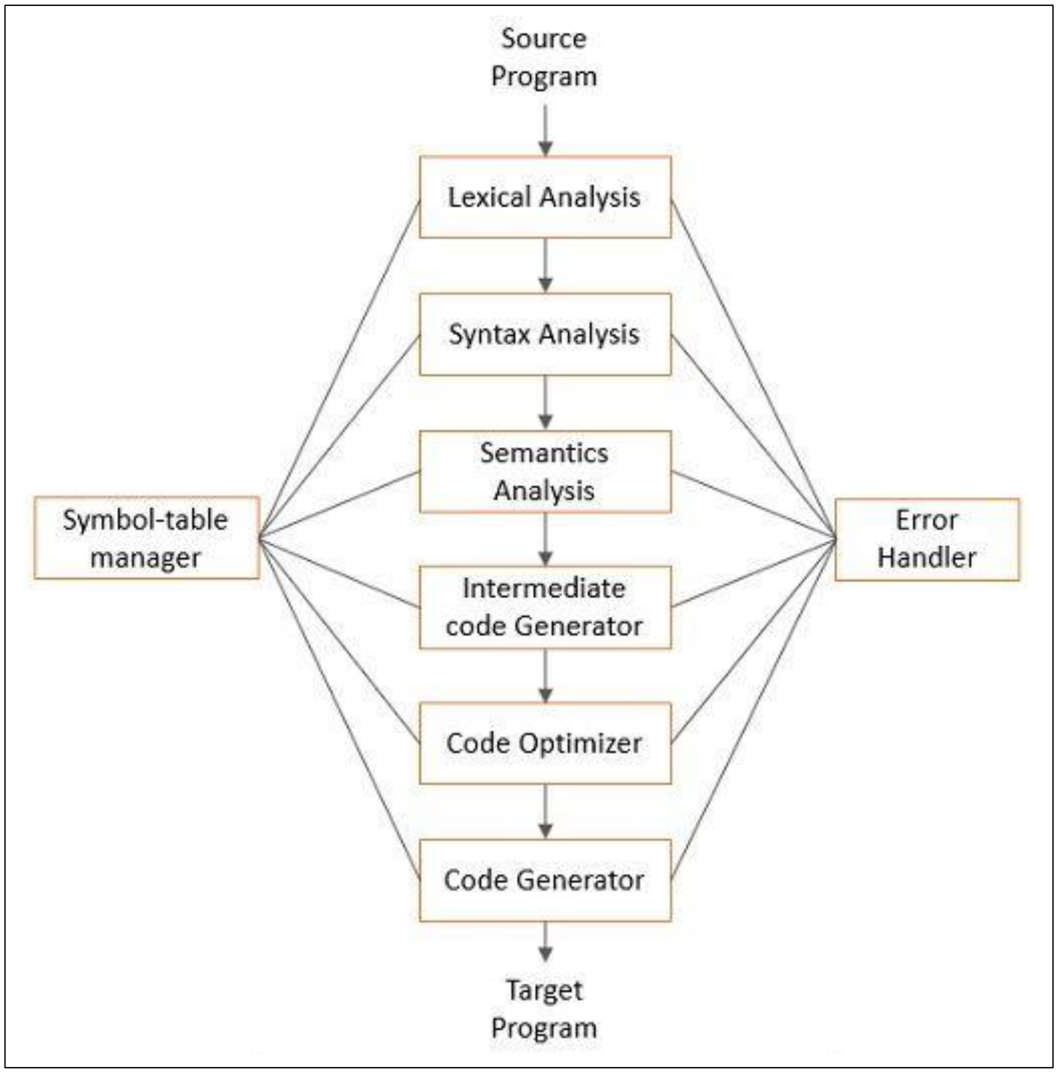
3. **Code generator:** The code generator is responsible for taking the internal representation of the pseudocode and translating it into C code. This is done by traversing the AST and emitting corresponding C code for each node in the tree. The code generator may also need to perform some optimizations or transformations to the code to ensure that it is efficient and correct.

4. **C code output**: Once the code generator has completed its work, the resulting C code is output to the user. This could be done by generating a text file containing the C code, or by displaying it in the user interface.

5. **Compiler integration:** Finally, the generated C code can be compiled and executed using a C compiler such as GCC or Clang. The auto code generation tool could be designed to integrate with these compilers, automatically invoking them to compile the generated code and produce an executable binary.

Overall, the architecture for an auto code generation tool in C involves several components working together to convert pseudocode into executable C code, while providing a user-friendly interface for developers to create efficient and correct software.

PHASES OF A COMPILER



1. **Lexical Analysis:** The first phase of a compiler is lexical analysis, also known as scanning. This phase reads the source code and breaks it into a stream of tokens, which are the basic units of the programming language. The tokens are then passed on to the next phase for further processing.

2. **Syntax Analysis:** The second phase of a compiler is syntax analysis, also known as parsing. This phase takes the stream of tokens generated by the lexical analysis phase and checks whether they conform to the grammar of the programming language. The output of this phase is usually an Abstract Syntax Tree (AST).

3. **Semantic Analysis:** The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, i.e., whether it conforms to the language's type system and other semantic rules.

4. **Intermediate Code Generation**: The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily translated into machine code.

5. **Optimization:** The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.

6. **Code Generation:** The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

MODULES DESCRIPTION

1. Understanding the Pseudo Code

The first step in converting pseudo code to C code is to fully understand the logic and structure of the pseudo-code. This involves breaking down the pseudo code into smaller parts and analyzing each part separately. It is important to identify the key variables, functions, and control structures used in the pseudo-code.

Once you have a clear understanding of the pseudo-code, you can begin to translate it into C code. This involves mapping each part of the pseudo-code to its corresponding C code construct. For example, if the pseudo code uses a loop to iterate over a set of data, you would use a for loop in the C code to achieve the same result.

2. Translating the Pseudo Code to C Code

Translating the pseudo code to C code requires a deep understanding of the syntax and structure of the C programming language. You must also be familiar with the various data types, operators, and control structures used in C code. It is important to pay attention to details such as variable names, function calls, and indentation, as these can affect the functionality of the code.

One effective technique for translating pseudo code to C code is to write out the C code in small chunks, testing each chunk as you go. This allows you to catch errors early on and make adjustments as needed. It is also important to document your code as you go, using comments and descriptive variable names to make the code more readable and maintainable.

3. Debugging the C Code

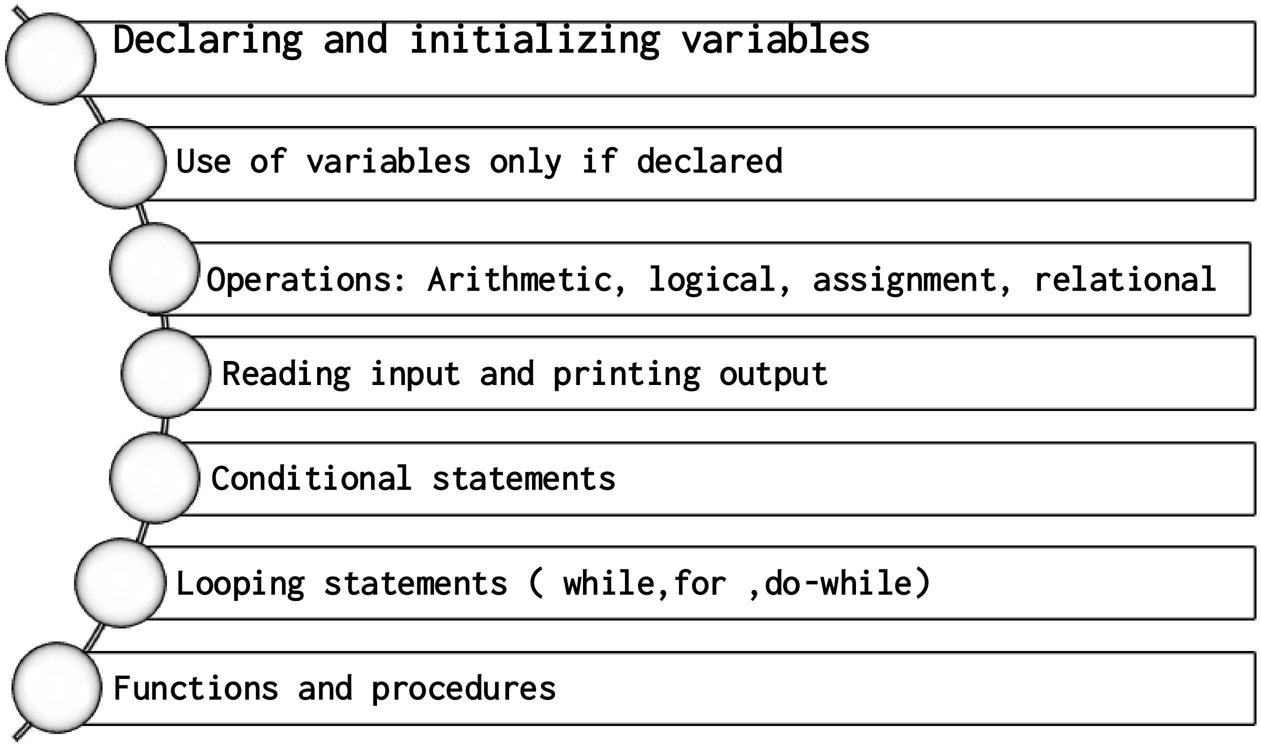
Once you have translated the pseudo code to C code, it is important to test and debug the code to ensure that it functions correctly. This involves running the code through a series of tests and identifying any errors or bugs that may arise. Common debugging techniques include using print statements, stepping through the code line by line, and using a debugger tool.

It is also important to consider edge cases and potential errors that may occur during runtime. This includes handling input validation, error messages, and unexpected behavior. By thoroughly testing and debugging your C code, you can ensure that it is reliable and functional.

4. Optimizing the C Code

After you have tested and debugged your C code, you may want to optimize it for performance or efficiency. This involves analyzing the code for areas where it can be improved, such as reducing unnecessary calculations or memory usage. It is important to balance optimization with readability and maintainability, as overly complex code can be difficult to understand and modify.

Common optimization techniques include using efficient algorithms, minimizing memory usage, and reducing the number of function calls. By optimizing your C code, you can improve its speed and efficiency, making it more useful for large-scale applications.

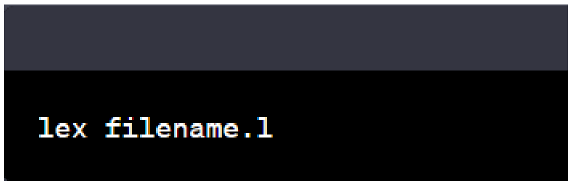

Declaring and initializing variables
Use of variables only if declared
Operations: Arithmetic, logical, assignment, relational
Reading input and printing output
Conditional statements
Looping statements (while,for ,do-while)
Functions and procedures

IMPLEMENTATION

The ‘lex’ command is used to generate a lexical analyzer or lexer from a ‘.l’ file that contains a set of regular expressions and corresponding actions. Here are the steps to use ‘lex’ to generate a lexer:

The `lex` command is used to generate a lexical analyzer or lexer from a `.l` file that contains a set of regular expressions and corresponding actions. Here are the steps to use `lex` to generate a lexer:

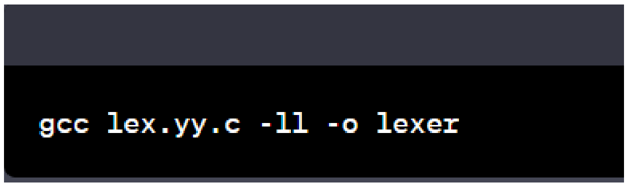
1. Write the regular expressions and corresponding actions in a `.l` file using the `lex` syntax.
2. Save the file with a `.l` extension, for example, `filename.l`.
3. Open the terminal and navigate to the directory where the `.l` file is located.
4. Type the following command in the terminal to generate the lexer:



```
lex filename.l
```

This will generate a `lex.yy.c` file in the same directory as the `.l` file.

5. Compile the generated `lex.yy.c` file using a C compiler. For example, if you have the `gcc` compiler installed, you can use the following command to compile the generated file:



```
gcc lex.yy.c -ll -o lexer
```

This will generate an executable file named `lexer`.

CODE FOR AUTO C CODE GENERATOR

```
alpha [a-zA-Z]
digit [0-9]
%%
"begin" return BEG;
"end" return END;
"int" { strcpy(yylval.code,yytext); return DATATYPE; }
"float" {strcpy(yylval.code,yytext);  return DATATYPE;}
"char" { strcpy(yylval.code,yytext); return DATATYPE; }
"double" { strcpy(yylval.code,yytext); return DATATYPE; }
"assign" return ASSIGN;
"to" return TO;
"print" return PRINT;
"scan" return SCAN;
"read" return READ;
"," return COMMA;
"(" return OPEN;
")" return CLOSE;
"if" return IF;
"then" return THEN;
"else" return ELSE;
"endif" return ENDIF;
"while" return WHILE;
"endwhile" return ENDWHILE;
```

```
"enddowhile" return ENDDOWHILE;
"do" return DO;
"for" return FOR;
"from" return FROM;
"repeat" return REPEAT;
"return" return RETURN;
"endfor" return ENDFOR;
"\"" return QUOTE;
"true" { strcpy(yylval.code,yytext); return BOOL; }
"false" { strcpy(yylval.code,yytext); return BOOL; }
"<=" { strcpy(yylval.code,yytext); return RELOP;}
">=" { strcpy(yylval.code,yytext); return RELOP; }
"==" { strcpy(yylval.code,yytext); return RELOP; }
"!=" { strcpy(yylval.code,yytext); return RELOP; }
"<" { strcpy(yylval.code,yytext); return RELOP; }
">" { strcpy(yylval.code,yytext); return RELOP; }
"&&" { strcpy(yylval.code,yytext); return LOGOP; }
"||" { strcpy(yylval.code,yytext); return LOGOP; }
"+" { strcpy(yylval.code,yytext); return AS; }
"-" { strcpy(yylval.code,yytext); return AS; }
"*" { strcpy(yylval.code,yytext); return MD; }
"/" { strcpy(yylval.code,yytext); return MD; }
"=" { strcpy(yylval.code,yytext); return Q; }
"start_procedure"    return START_PROCEDURE;
"end_procedure"    return END_FUNCTION;
```

```
{alpha}({alpha}|{digit})*   {strcpy(yylval.code,yytext); return VAR; }
{alpha}({alpha}|{digit})*/[(] { strcpy(yylval.code,yytext); return
NAME_PROCEDURE;}
[0-9]+ { strcpy(yylval.code,yytext); return NUM; }
%%
```

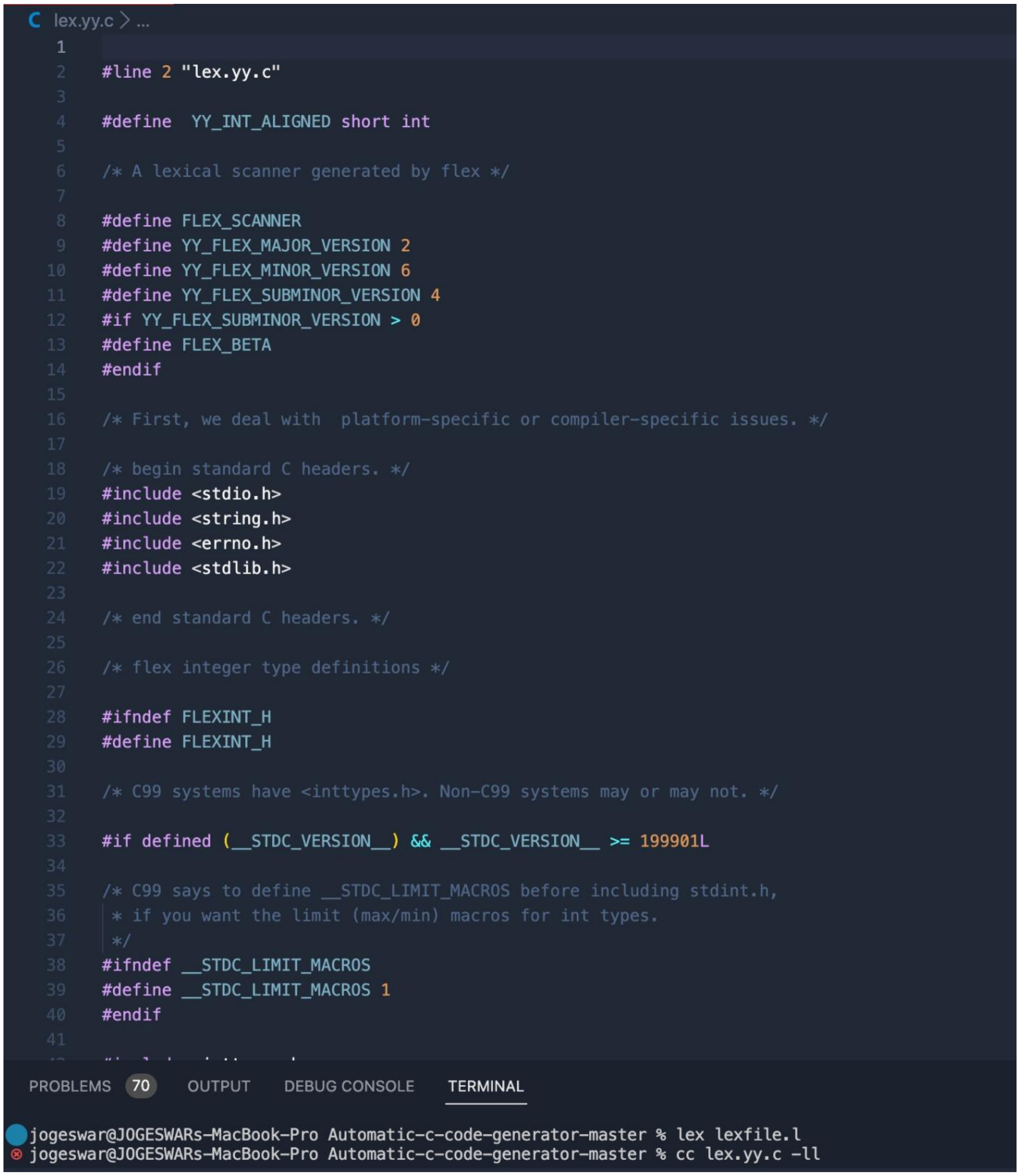
OUTPUT

#User-Inputs

```
assign ( b int )
assign ( c int )
assign ( d int )
assign a to 10
assign b to 15
print "Hello all"
while a+b > 20
then
if b > 15
then
c = a + b
endif
endwhile
print " The answer is int , c "
read int , d
start_procedure myfunction(a int)
print int , c
end_procedure
for a from 1 to 10
repeat
b = b + a
endfor
```

```
print int , b
print "Bye"
end
```

#Results Generated



```

#line 2 "lex.yy.c"

#define  YY_INT_ALIGNED short int

/* A lexical scanner generated by flex */

#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 6
#define YY_FLEX_SUBMINOR_VERSION 4
#if YY_FLEX_SUBMINOR_VERSION > 0
#define FLEX_BETA
#endif

/* First, we deal with  platform-specific or compiler-specific issues. */

/* begin standard C headers. */
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>

/* end standard C headers. */

/* flex integer type definitions */

#ifndef FLEXINT_H
#define FLEXINT_H

/* C99 systems have <inttypes.h>. Non-C99 systems may or may not. */

#if defined (__STDC_VERSION__) && __STDC_VERSION__ >= 199901L

/* C99 says to define __STDC_LIMIT_MACROS before including stdint.h,
 * if you want the limit (max/min) macros for int types.
 */
#ifndef __STDC_LIMIT_MACROS
#define __STDC_LIMIT_MACROS 1
#endif

```

```
jogeswar@JOGESWARs-MacBook-Pro Automatic-c-code-generator-master % lex lexfile.l
jogeswar@JOGESWARs-MacBook-Pro Automatic-c-code-generator-master % cc lex.yy.c -ll
```

Result:

Thus, the programs were successfully executed and outputs were verified. The source code was successfully converted into a machine-level code and the Auto Code generating system was run successfully and has been verified.

FUTURE ENHANCEMENT

There are several potential future enhancements that could be made to an auto code generation system for pseudocode in C, including:

- **Support for additional programming languages:** While the system may be designed to generate C code, it could be extended to support additional programming languages such as Java, Python, or Ruby. This would make the system more versatile and useful for a wider range of developers.

- **Advanced optimization techniques:** While the code generator may perform some basic optimizations, such as loop unrolling or constant folding, more advanced optimization techniques could be implemented. This could include techniques such as code profiling, dynamic analysis, or machine learning-based optimization.

- **Integration with machine learning models:** The system could be extended to integrate with machine learning models to automatically optimize generated code. This could involve using neural networks or other machine learning models to analyze the generated code and suggest improvements.

- **Integration with version control systems:** The system could be integrated with version control systems such as Git to allow developers to track changes to generated code over time. This would make it easier to collaborate with other developers and maintain code quality.

- **Interactive debugging tools:** The system could be extended to provide interactive debugging tools that allow developers to step through generated code and identify errors. This would make it easier to debug and maintain generated code.

- **Integration with cloud services:** The system could be integrated with cloud services such as AWS or Google Cloud to provide additional functionality such as code compilation and deployment. This would make it easier to deploy generated code to production environments.

Overall, there are many potential enhancements that could be made to an auto code generation system for pseudocode in C. By continually improving and enhancing the system, developers can make it more useful and versatile for a wider range of applications.

CONCLUSION

In conclusion, the auto-generation of C code from pseudocode is a useful technique that can save programmers a significant amount of time and effort. By using specialized tools or programming languages, developers can easily convert pseudocode into efficient and reliable C code that can be used to build complex software applications. This process not only helps in reducing the development time but also ensures the accuracy of the code, as it eliminates the possibility of human errors during the manual conversion of pseudocode to C code.

REFERENCES

- Lex & Yacc: http://dinosaur.compilertools.net/
- "Compiler Construction" by Kenneth C. Louden: https://www.cs.sjsu.edu/~louden/cmptext/
- "Writing Compilers and Interpreters" by Ronald Mak: https://www.amazon.com/Writing-Compilers-Interpreters-Techniques-Software/dp/0470177071
- "Crafting Interpreters" by Bob Nystrom: https://craftinginterpreters.com/
- "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman: https://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811