MINI C COMPILER

Automata and Compiler Design (IT250) Project Report

Submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

By

MAYUR JINDE (201IT135)

SANKET HANAGANDI (201IT154)

KALPANA T (201IT228)

DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE - 575025

MAY, 2022

D E C L A R A T I O N

I hereby *declare* that the *Automata and Compiler Design (IT250) Project Report* entitled "**Mini C Compiler**" which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a ***bonafide report of the work carried out by me.*** The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Mayur Jinde – 201IT135	Sanket Hanagandi – 201IT154	Kalpana T – 201IT228
_ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _	_ _ _ _ _ _ _ _ _ _ _ _
Signature of the Student	Signature of the Student	Signature of the Student

Place: NITK, Surathkal
Date: 18th May, 2022

CERTIFICATE

This is to certify that the *Automata and Compiler Design (IT250) Project Report* entitled **"Mini C Compiler"** has been presented by ***Mayur Jinde – 201IT135, Sanket Hanagandi – 201IT154, Kalpana T – 201IT228***, students of IV semester B.Tech. (IT), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on 17th May, during the even semester of the academic year 2021 - 2022, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Guide Name

_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Signature of the Guide with Date

Place: NITK, Surathkal
Date: 18th May, 2022

Table of Contents Page no.

List of Figures

CHAPTER 1: INTRODUCTION

A compiler is a program that can read a program in one language - the source language and translate it into an equivalent program in another language - the target language.

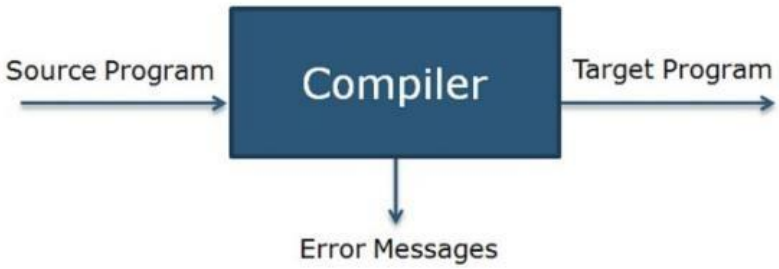

Figure 1.1 Working of a compiler

A compiler can broadly be divided into two phases based on the way they compile i.e. analysis phase (front end) and synthesis phase (back end). The analysis phase breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part. The synthesis phase constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

CHAPTER 2: OBJECTIVE

The main goal of this project is to design a mini compiler for a subset of the C language. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

Phases of the project

- Implementation of Scanner/Lexical Analyzer
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

CHAPTER 3: METHODOLOGY

3.1 Lexical Analyzer

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

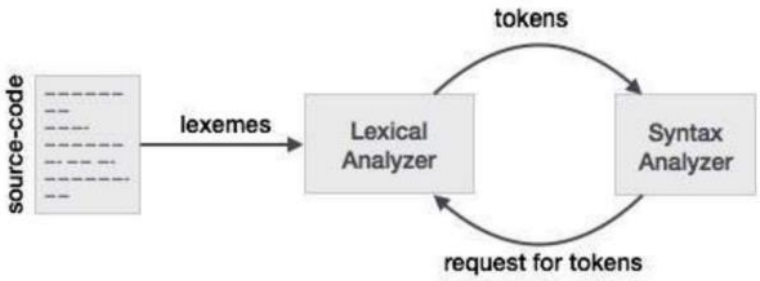

Figure 3.1 Working of lexical analyzer

In our script we have implemented all the functionalities targeted by our scanner. The definition section is used for including relevant header files and initializing the variables for the symbol table and the constant table. The rules section defines rules for all the tokens to be identified, this includes keywords, operators, identifiers, pre-processor directives, comments, strings, rules for all the tokens that have been implemented here.

3.2 Parser

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree

In the syntax analysis phase, the parser verifies whether the tokens generated by the lexical analyzer are grouped according to the syntactic rules of the language. The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

A lexer and a parser work in sequence: the lexer scans the input and produces the matching tokens, the parser then scans the tokens and produces the parsing result.

Some other principles which are crucial to parser design are:

1. Using Leftmost Derivations or Rightmost derivations
2. The core algorithm used, reduction or predictive parsers

Another classification is based on how parsers process the program; they can either process the program from top to bottom (top-down parsers) or they can process it from bottom to top (bottom-up).

Yacc stands for Yet Another Compiler-Compiler. Yacc provides a general tool for describing the input to a computer program. The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

The definition section is used to define any parameters for the C program, any header files to be included and global variable declarations. We also define all parameters related to the parser here, specifications like using Leftmost derivations or Rightmost derivations, precedence and left-right associativity are declared here, and data types and tokens which

will be used by the lexical analyzer are also declared in this stage. The Rules section contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses this rule for reducing the token stream received from the scanner, all rules are linked to each other from the start state, which is declared in the rules section. In the C code section, the parser is called, and the symbol table and constant tables are initialized in this section. The lex.yy.c file created by the lex script is also included from here which the parser calls. In this section we also define the error function used by the parser to report syntactical errors along with line numbers.

3.3 Semantic Analysis

The semantic analyzer 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.

In the C code section, the parser is called, and the symbol table and constant tables are initialized in this section. After parsing and semantic checking the tables are filled with nesting level, function flag, array flag, array dimension, etc. The lex.yy.c file created by the lex script is also included from here which the parser calls. In this section we also define the error function used by the parser/semantic checker to report syntactical and semantic errors along with line numbers.

3.4 Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end and the details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end of language and i and the back end of machine j. This tackles the need the create a native compiler for each language and machine.

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. The complexity of this code lies between the source language code and the object code. The intermediate code can be represented in the form of postfix

notation, syntax tree, directed acyclic graph (DAG), three-address code, quadruples, and triples.

Here in this project, we have represented the code in the form of three-address code.

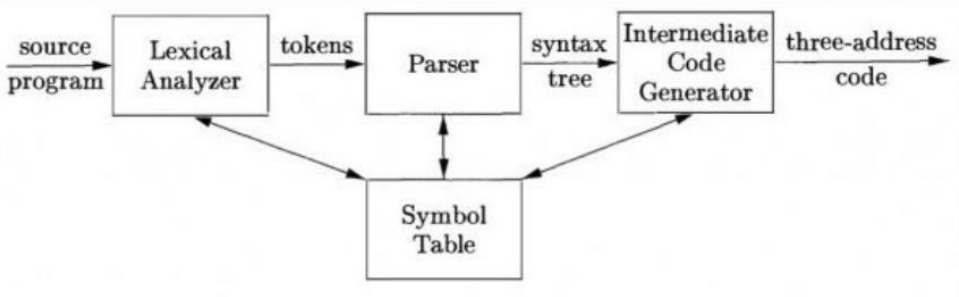

Figure 3.2 Phases of a compiler up till three address code

CHAPTER 4: FLOW CHART

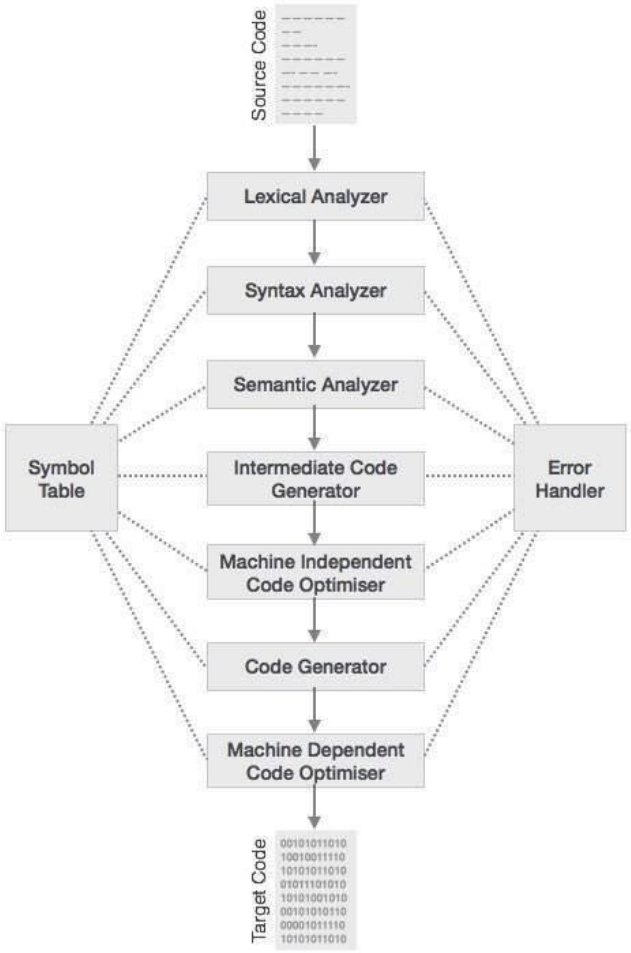

Figure 4.1

Symbol Table

The Symbol Table is an important data structure created and maintained by the compiler to keep track of semantics of variables i.e., it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc. It is used by all the phases of the compiler.

Error Handler

The tasks of the Error Handling process are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error. During this entire process processing time of the program should not be slow. Functions of the Error Handler include Error Detection, Error Report and Error Recovery

Analysis Phase – The analysis phase can be divided into three phases as follows:

1. Lexical Analyzer

Lexical Analysis is the first phase of the compiler, also known as a scanner. It converts the High-level input program into a sequence of Tokens. The sequence of tokens is sent to the parser for syntax analysis

2. Syntax Analyzer

The Syntax parser analyses the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

3. Semantic Analyzer

Semantic Analysis is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of the program are semantically correct.

Synthesis Phase – The synthesis phase can be divided into three phases as follows:

1. Intermediate Code Generator

Intermediate code generator receives input from semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code.

2. Code Optimizer

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result.

3. Code Generator

Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

At last, the code generator decides the order in which the instruction will be executed.

It creates schedules for instructions to execute them.

CHAPTER 5: IMPLEMENTATION

All header files are of the format “^"#include"[ ]*<.+.h>”. The main function does not take any input parameters. Numbers follow the format “[-]?{digit}+| [-

]?{digit}+.{digit}{1,6}”. Hence numbers of the type +123.45 are not permitted. The variables can be of int, float, or char type. No arrays are permitted. The body of an if and else block is enclosed in parenthesis even if it is a single line. The body of a for loop is enclosed in parenthesis, even if it is a single line. printf statements take a single string as a parameter. scanf statements take a string and &id as input.

The first step was to code the lexer file to take the stream of characters from the input programs and identify the tokens defined with the help of regular expressions. Next, the yacc file was created, which contained the context-free grammar that accepts a complete C program constituting headers, main function, variable declarations, and initializations, nested for loops, if-else constructs, and expressions of the form of binary arithmetic and unary operations. At this stage, the parser will accept a program with the correct structure and throw a syntax error if the input program is not derived from the CFG.

5.1 Lexical Analysis

A struct was defined with the attributes id_name, data_type (int, float, char, etc.), type (keyword, identifier, constant, etc.), and line_no to generate a symbol table. The symbol table is an array of the above-defined struct. Whenever the program encounters a header, keyword, a constant or a variable declaration, the add function is called, which takes the type of the symbol as a parameter. In the add function, we first check if the element is already present in the symbol table. If it is not present, a new entry is created using the value of yytext as id_name, datatype from the global variable type in case of variables,

type from the passed parameter, and line_no. After the CFG accepts the program, the symbol table is printed.

5.2 Syntax Analysis

For the syntax analysis phase, a struct was declared that represented the node for the binary tree that is to be generated. The struct node has attributes left, right, and a token which is a character array. All the tokens are declared to be of type nam, a struct with attributes node and name, representing the token's name. To generate the syntax tree, a node is created for each token and linked to the nodes of the tokens, which occur to its left and right semantically. The inorder traversal of the generated syntax tree should recreate the program logically.

5.3 Semantic Analysis

The semantic analyzer handles three types of static checks.

1. Variables should be declared before use. For this, we use the check_declaration function that checks if the identifier passed as a parameter is present in the symbol table. If it is not, an informative error message is printed. The check declaration function is called every time an identifier is encountered in a statement apart from a declarative statement.

2. Variables cannot be redeclared. Since our compiler assumes a single scope, variables cannot be redeclared even within loops. For this check, the add function is changed to check if the symbol is present in the symbol table before inserting it. If the symbol is already present, and it is of the type variable, the user is attempting to redeclare it, and an error message is printed.

3. Pertains to type checking of variables in an arithmetic expression. For this, the check_types function is used, which takes the types of both variables as input. If the types match, nothing is done. If one variable needs to be converted to another type, the corresponding type conversion node is inserted in the syntax tree (int to float or float to int). The type field was added to the struct representing value and expression tokens to keep track of the type of the compound expressions that are not present in the symbol table. The output of this phase is the annotated syntax tree.

5.4 Intermediate code generation

For intermediate code generation, the three-address code representation was used. Variables were used to keep track of the next temporary variable and label to be generated. The condition statements of if and for were also declared to store the labels to goto in case the condition is satisfied or not satisfied. The output of this step is the intermediate code.

CHAPTER 6: RESULTS

Initially there are lex and yacc files

Figure 6.1

Step 1: Running Lex file

```
lex lexer.l
```

Figure 6.2

Step 2: Running Yacc file

```
yacc -d parser.y
```

Figure 6.3

Step 3:

```
gcc lex.yy.c y.tab.c
```

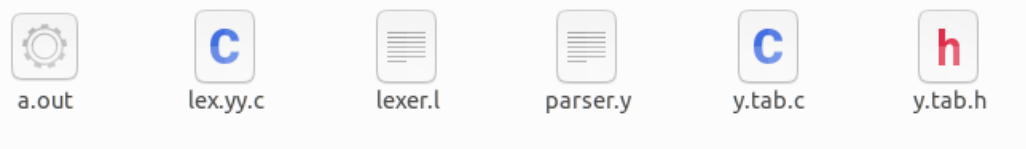

Figure 6.4

Step 4: Results for compiling a sample c program

```
./a.out 1.c
```

```
Compilation Successful!!

The Entries in the symbol table are :

Name            Type                    DataType
_______________________________________________
main()          Function                none
n               Identifier              int
i               Identifier              int
factorial       Identifier              float
printf()        Function                none
printf()        Function                none
printf()        Function                none
```

```
Header : #include <stdio.h>
Data Type : int
Function : main()
Curly Bracket : {
Data Type : int
Identifier : n
Comma : ,
Identifier : i
Comma : ;
Data Type : float
Identifier : factorial
Equals : =
Digits : 1
Comma : ;
                                ICG: factorial = 1
Identifier : factorial
Equals : =
Digits : 3
Comma : ;
                                ICG: factorial = 3
Printf : printf("Factorial of 7: ")
Comma : ;
Identifier : n
Equals : =
Digits : 7
Comma : ;
```

Figure 6.5

References:

1. Lex and YACC - 2nd Edition - Levine, Mason Brown.]

2. Jenkins Hash Function on Wikipedia:

https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time