COMPILER CONSTRUCTION

Final Report

Submitted To:

Mr. Laeeq Khan Niazi

Submitted by:

Muhammad Yaqoob

Reg. No.:

2021-CS-118

Department of Computer Science

University of Engineering and Technology (UET) Lahore

ACKNOWLEDGEMENT

It is my pleasure to acknowledge you that I have received a project on compiler design from my teacher. My first sincere appreciation goes to **Mr. Laeeq Khan Niazi** for his guidance throughout the semester.

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of people whose ceaseless cooperation made it possible, whose constant guidance and encouragement crown all efforts with success. I am very grateful to my Compiler Construction's lab teacher **Mr. Laeeq Khan Niazi** for the inspiration and constructive suggestions that helped me in the preparation of this project.

I also thank my parents, family and friends at large for their moral support during the compiler construction to ensure successful completion of the project.

CONTENTS

Chapter 1

Introduction

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.
The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of the source program, and feeds its output to the next phase of the compiler.

The analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate code which is also referred as Assembly Language Code.

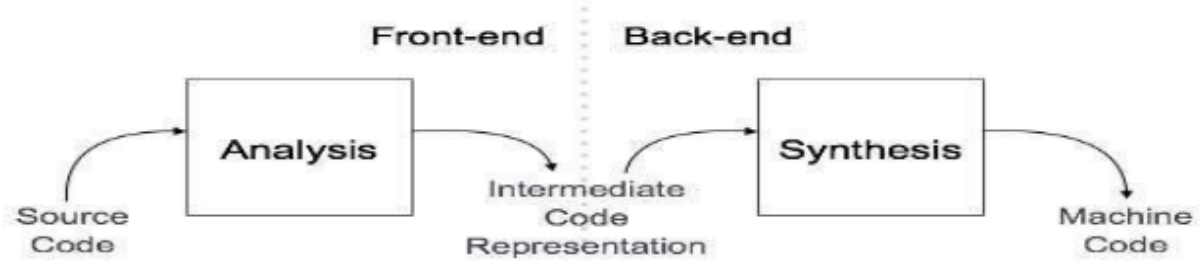


Figure 1 Compiler Architecture

My compiler takes a source code of C++ language and converts it to intermediate code. This part of compiler design is the front end. This Intermediate code generated is independent of the machine. This code is later optimised and converted to assembly code.

Chapter 2

Language Description

2.1. Language

My language has the following specifications.

Keywords

Control Flow

- T_IF: Represents the if keyword used for conditional branching.
- T_ELSE: Represents the else keyword for alternate conditions.
- T_AGAR: Alternate syntax for if (customized keyword).
- T_MAGAR: Alternate syntax for else (customized keyword).
- T_WHILE: Keyword for while loop, used for iterative constructs.
- T_FOR: Represents the for loop keyword.
- T_DO: Represents the do keyword for do-while loops.

Comparison Operators

- T_EQ: Represents == (equality comparison).
- T_NE: Represents != (inequality comparison).
- T_LT: Represents < (less than).
- T_GT: Represents > (greater than).
- T_LE: Represents <= (less than or equal to).
- T_GE: Represents >= (greater than or equal to).

Data Types

- T_INT: Keyword for declaring integer variables.
- T_FLOAT: Keyword for declaring float variables.
- T_DOUBLE: Keyword for declaring double precision variables.
- T_STRING: Keyword for declaring string variables.
- T_CHAR: Keyword for declaring character variables.
- T_BOOL: Keyword for declaring boolean variables.
- T_VOID: Represents the void keyword used to declare functions that do not return a value.

Special Values

- T_TRUE: Represents the boolean value true.
- T_FALSE: Represents the boolean value false.

Input/Output

- T_STREAM_INSERTION_OPERATOR: Represents the << operator (e.g., for output streams like cout).
- T_STANDARD_OUTPUT_STREAM: Represents the cout keyword for standard output.
- T_EXTRACTION_OPERATOR: Represents the >> operator (e.g., for input streams like cin).
- T_STANDARD_INPUT_STREAM: Represents the cin keyword for standard input.

Loops and Control Statements

- T_BREAK: Represents the break keyword to exit loops or switch statements.
- T_CONTINUE: Represents the continue keyword to skip the current iteration.

Switch-Case

- T_SWITCH: Represents the switch statement for multi-branch conditions.
- T_CASE: Represents the case keyword for switch conditions.
- T_COLON: Represents the colon (:) in switch-case statements.
- T_DEFAULT: Represents the default keyword in switch statements.

Operators

- T_ASSIGN: Represents the = operator for assignment.
- T_PLUS: Represents the + operator for addition.
- T_MINUS: Represents the - operator for subtraction.
- T_MUL: Represents the * operator for multiplication.
- T_DIV: Represents the / operator for division.

Logical Operators

- T_LOGICAL_AND: Represents the logical AND operator (&&).
- T_LOGICAL_OR: Represents the logical OR operator (||).

Miscellaneous

- T_PREPROCESSOR: Represents preprocessor directives (e.g., #include, #define).
- T_ID: Token for identifiers (variable or function names).
- T_NUM: Token for numeric literals.
- T_RETURN: Represents the return keyword for returning values.
- T_SEMICOLON: Represents the ; token for statement termination.
- T_LPAREN: Represents the left parenthesis (.
- T_RPAREN: Represents the right parenthesis).
- T_LBRACE: Represents the left brace {.
- T_RBRACE: Represents the right brace }.
- T_EOF: Represents the end of file token.
- T_UNKNOWN: Represents any unknown or invalid token.

Identifier Rules

I. Identifiers can be of any length.
II. Identifiers are case-sensitive.
III. An Identifier can only have alphanumeric characters(a-z , A-Z , 0-9) and underscore(_).
IV. Keywords are not allowed to be used as Identifiers
V. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.
VI. Special Characters:Special characters are handled as tokens and not permitted within identifiers.

Data Types

My language supports the following data types

1. Integer
2. Float
3. Double
4. String
5. Character
6. Boolean

Expressions

A. Arithmetic operators (+, -, *, /)
B. Parenthesis
C. All kinds of Numbers support (integer support, float support, double support, scientific number support)
D. String Support
E. Relational expression to be supported (>, <, >=, <=, ==, !=)

Logical expressions

Language support Logical expressions inside if conditions like &&, ||, == and !=

Symbol table

I have used map data structure for the symbol table implementation. It stores information about

- variable name and
- type of that variable.
- Scope of variable
- Value stored in it, if it is initialized

Reason for choosing the map data structure is the access of entries of the symbol table takes constant time complexity.

Variable Name	Type	Scope	Value
counter	int	GLOBAL	t4
i	int	GLOBAL	2
result	bool	GLOBAL	true

Preprocessor Directives

- Preprocessor directives support

Comments

- Single Line comments support.
- MultiLine Line comments support.

Statements

- Declaration statement
- Assignment statement
- Single line declaration and assignment
- Conditional statements:
    - If statement
    - Else statement
    - Agar statement
    - Magar statement
- Loop statements:
    - For Loop statement
    - While Loop statement
    - DoWhile statement
- Increment statement
- Decrement statement
- Return statement
- Block statement (for grouping multiple statements)
- Break statement
- Continue statement
- Print statement
- Input statement

Function feature

Language supports the void function with void parameters.

Error Handling Support

Compiler provides error handling at each phase of the compiler construction. It provides proper error handling in lexer, parser, symbol table, intermediate code generator and assembly code generator. It will inform you about the line number in which you are making a mistake. So, you can correct it.

Human Friendly Errors

Language errors are more human friendly. They display the line number of the error for the ease of user.

Parsing from CMD

Compiler takes code like passing the file name from cmd and takes that code and passes accordingly.

2.2. Directory Structure

```
/COMPILER
|-- /src
|   |-- /lexer
|   |   |-- lexer.cpp
|   |   |-- lexer.h
|   |
|   |-- /parser
|   |   |-- parser.cpp
|   |   |-- parser.h
|   |
|   |-- /assembly
|   |   |-- acg.cpp
|   |   |-- acg.h
|   |
|   |-- /intermediate
|   |   |-- icg.cpp
|   |   |-- icg.h
|   |
|   |-- /symboltable
|   |   |-- symbTable.cpp
|   |   |-- symbTable.h
|
|-- /bin
|   |-- main.exe
|   |-- lexer.o
|   |-- parser.o
|   |-- acg.o
|   |-- icg.o
|   |-- symbTable.o
|
|-- /target
|   |-- icg.obj
|   |-- assembly.asm
|
|-- Makefile
|-- README.md
```

2.3. Sample Program

```
// Testing single-line comments
/* Testing
   multi-line
   comments */

// Testing while loop (using 'while')
int counter = 11;
while (counter < 50) {
    counter = 20;
}

// Testing for loop (using 'for')
bool result = false;
for (int i = 2; i < 5; i = i + 1) {
    result = true;
}
```

2.4. Code generated

```
section .data
    L4 dd 0
    counter dd 0
    L0 dd 0
    < dd 0
    i dd 0
    true dd 0
    t3 dd 0
    result dd 0
    t2 dd 0
    false dd 0
    t7 dd 0
    L1 dd 0
    L6 dd 0
    t8 dd 0

section .text
    global _start
_start:
    mov dword [counter], 11
    jmp L0
```



```
L0:
    mov dword [t2], counter < 50
```

```
    mov dword [t3], t2
    cmp dword [t3 ], 1
    je L1
    jmp L0
    mov dword [counter], 20
    jmp L0

L1:
    mov dword [result], false
    mov dword [i], 2

L4:
    mov dword [t7], i < 5
    cmp dword [t7 ], 1
    je L6
    mov eax, [i]
    add eax, [1]
    mov dword [t8], eax
    mov dword [i], t8

L5:
    mov dword [result], true
    mov dword [i], t8
    jmp L4

L6:
```

Chapter 3

Phases of Compiler

3.1. Overview

Analysis part of the compiler breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program. It is also termed as the front end of a compiler.

3.2. Lexical analyser

Lexical analysis is the process of converting a sequence of characters from a source program into a sequence of tokens. A program which performs lexical analysis is termed as a lexical analyzer (lexer), tokenizer or scanner.

Lexical analysis consists of two stages of processing which are as follows:

- Scanning
- Tokenization

Token is a valid sequence of characters which are given by lexeme. In a programming language,

- keywords,
- constant,
- identifiers,
- numbers,
- operators and
- punctuation symbols are possible tokens to be identified.

  For example : c=a+b;

  In this c,a and b are identifiers and '=' and '*' are mathematical operators.

Lexical Errors

- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.
- Misspelling of identifiers, keywords, or operators are considered as lexical errors.

Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

3.3. Syntax analyser

- Syntax analysis is the second phase of the compiler. Syntax analysis is also known as parsing.
- Parsing is the process of determining whether a string of tokens can be generated by a grammar.
- It is performed by a syntax analyzer which can also be termed as a parser.
- In addition to construction of the parse tree, syntax analysis also checks and reports syntax errors accurately. Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of the compiler for further processing.
- Parser implements context free grammar for performing error checks.

Types of Parser

- Top down parsers Top down parsers construct parse tree from root to leaves.
- Bottom up parsers Bottom up parsers construct parse tree from leaves to root.

Role of Parser

- Once a token is generated by the lexical analyzer, it is passed to the parser.
- On receiving a token, the parser verifies the string of token names that can be generated by the grammar of the source language.
- It calls the function getNextToken(), to notify the lexical analyzer to yield another token.
- It scans the token one at a time from left to right to construct the parse tree.
- It also checks the syntactic constructs of the grammar.

3.4. Semantic analyser

- Semantic analysis is the third phase of the compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.

3.5. Intermediate code generator

Intermediate code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- o Postfix notation
- o Three address code
- o Syntax tree

Most commonly used form is the three address code.

t_1 = inttofloat (5)

$t_2 = id_3 * tl$

$t_3 = id_2 + t_2$

$id_1 = t_3$

Properties of intermediate code

- It is easy to produce.
- It is easy to translate into a target program.

After intermediate code generation the front end part of the compiler finishes.The output to intermediate code generated is fed as input to the back end of the compiler , which converts this Intermediate code to machine code.

3.6. Assembly code generator

The Assembly Code Generator is responsible for converting intermediate code, such as Three-Address Code (TAC), into machine-executable assembly instructions. This is the final stage where abstract instructions are translated into a concrete format that a processor can understand and execute.

Overview

The assembly code generator processes TAC by:

- Declaring necessary variables in the data section.
- Translating assignments, arithmetic operations, and control flow statements into assembly instructions.
- Generating executable sections with proper labels and instructions.

Features of the Assembly Code Generator

Variable Declaration: Variables from TAC are collected and declared in the .data section as double words (dd).

```
section .data
    x dd 0
    y dd 0
```

Arithmetic Operations: The generator translates arithmetic operations in TAC into equivalent assembly instructions.

TAC:

```
t1 = x + y
```

Assembly:

```
mov eax, [x]
add eax, [y]
mov dword [t1], eax
```

Conditional Statements: Conditional instructions in TAC, such as if statements, are converted into comparison (cmp) and branching instructions (je, jg, etc.).

TAC:

```
if x > y goto L1
```

Assembly:

```
mov eax, [x]
cmp eax, [y]
jg L1
```

Labels and Gotos: Labels and goto instructions are directly mapped to assembly syntax.

TAC:

```
L1:
goto L2
```

Assembly:

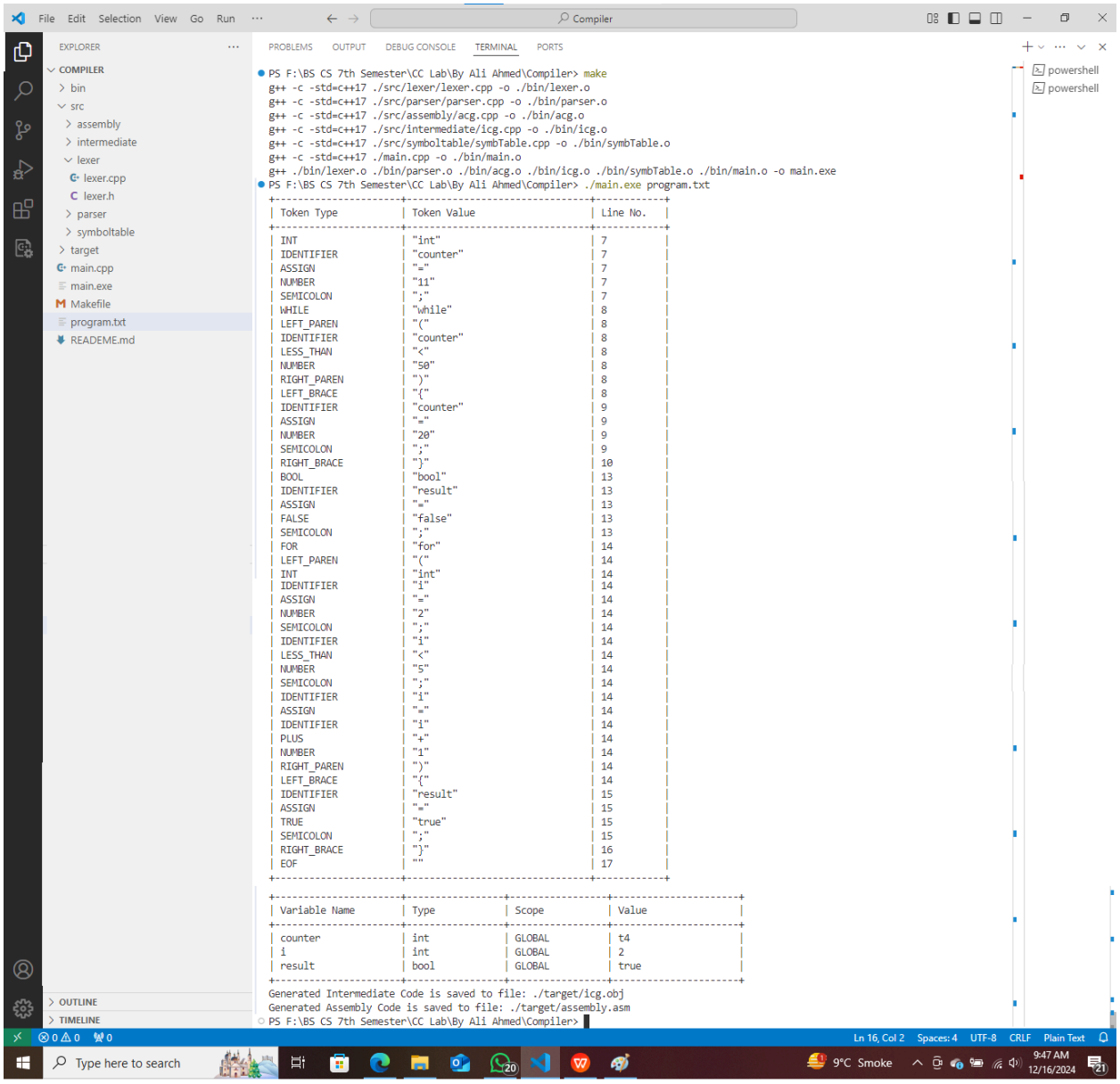
```
L1:
    jmp L2
```

Properties of the Assembly Code Generator

- Simple Translation: Designed for easy translation of TAC into assembly code.
- Executable Code: Produces an output that can be executed by an x86-compatible processor.
- Completes Back-End: The generated assembly serves as the final output of the back-end compiler phase.

Chapter 4

Screenshots



Chapter 5

Feasibility and future scope

New languages which are more close to general languages are being invented.With the growth of technology, ease of working is given priority. We have emerged from C , C++ to python ,ruby , etc. which require less lines of code . There are other platforms such as Android Studio, Qt which provide easy GUI creation and use the popular languages Java and C++ respectively.

My project can be extended to form a new language which is easy to learn, faster , has more inbuilt features and has many more qualities of a good programming language.

A compiler is a program used for automated translation of computer programs from one language to another. It translates input source code to output machine code which can be executed.Often, the input language is one that a given computer can’t directly execute, for example, because the language is designed to be human-readable. Often, the output language is one that a given computer can directly execute.

We don’t ever need a compiler to execute a program. It is just an intermediate to convert a High Level Language program to machine executable code.

Chapter 6

Conclusion

In a compiler the process of Intermediate code generation is independent of the machine and the process of conversion of Intermediate code to target code is independent of language used.

Thus we have done the front end of the compilation process. It includes 3 phases of compilation lexical analysis, syntax analysis and semantic analysis which is then followed by intermediate code generation and assembly code generation.

In computer programming, the translation of source code into object code by a compiler. This report outlines the analysis phase in compiler construction. In its implementation and source language is converted to assembly level language.