Compiler Construction Lab

Course Teacher: Miss Asia Samreen
Lab Instructor: Sir Bilal Vohra

Date: 31/Dec/2020

Project Members name:

Talha Maqsood	02-134182-080
Muhammad Ahmed Siddiqui	02-134182-070
Ayesha Younus	02-134182-048

Index

ACKNOWLEDGEMENT

It is my pleasure to acknowledge you that I have received a project on Mini compiler design from my teacher.My first sincere appreciation goes to **Sir Bilal Vohra**for her guidance throughout the project.

The satisfaction that accompanies that the successful completion of any task would be in complete 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 project supervisor **Miss Asia Samreen** for the inspiration and constructive suggestions that helped me in the preparation of this project.

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

PURPOSE

The purpose of the compiler is to translate the programs written in human readable language to machine language which is understandable to computer machine. A compiler reads instruction in the programs and translates it, if there is any error it also identify it and warns about it to user. The error could be typing mistake (syntax error) or a logical error. When all the errors are removed the instructions are sent to the computer for processing.

INTRODUCTION

A compiler is a computer program(or set of programs) that transforms source code written in a programming language(the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program. The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly languageor machine code). If the compiled program can only run on a computer whose CPU or operating system is different from the one on which the compiler runs the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

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.

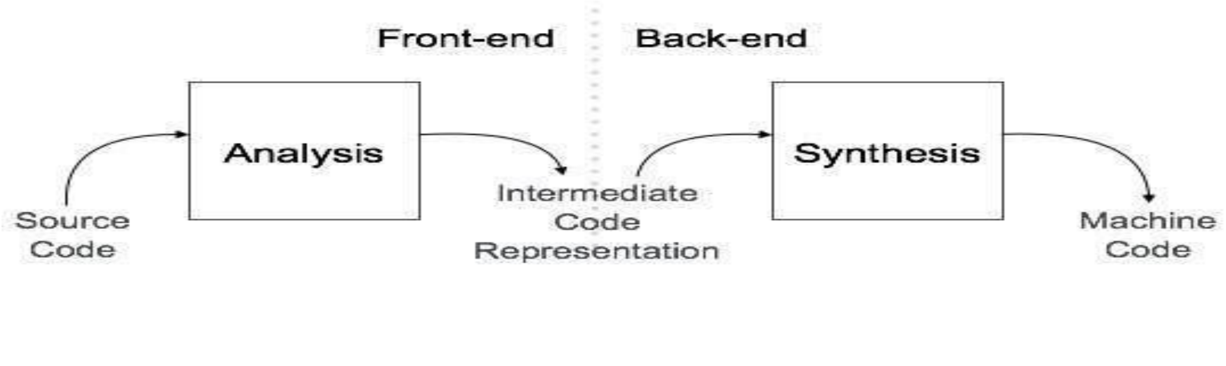


Figure 1 Compiler Architecture

Our project is to take a language and convert it to lexical phase. This part of compiler design is front end. This Intermediate code generated is independent of machine. This code is later optimised and converted to machine code.

Language Description

Language:

Our language will have the following specifications.

Identifier Rules

I. Identifier can be of maximum length 6.
II. Identifiers are not case sensitive.
III. An indentifier can only have alphanumeric characters(a-z , A-Z , 0-9) andunderscore(_).
IV. The first character of an identifier can only contain alphabet(a-z , A-Z).
V. Keywords are not allowed to be used as Identifiers.
VI. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

Data Types:

Our language supports only 3 datatypes

1. Integer
2. String
3. Character
4. Bool
5. Double
6. Float
7. char

Expressions

a. Arithmetic operators (+, -, *, /, %)
b. Uniray operator
c. Paranthesis
d. Only Integer supported
e. Relational expression to be supported (>, <, >=, <=, ==, !=)
f. Character string and integer constants

e.g.

intconst 4

charconst ‘4’

stringconst “4”

Statements

a. Declaration statement :int a;
b. Declaration and Initialisation :int a=5;
c. Assingment Statement : a=6;
d. Conditional statement

Phases of compiler:

Overview

Analysis part of 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 front end of compiler.

Lexical analyser

Lexical analysis is the process of converting a sequence of characters from 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
- punctuations 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

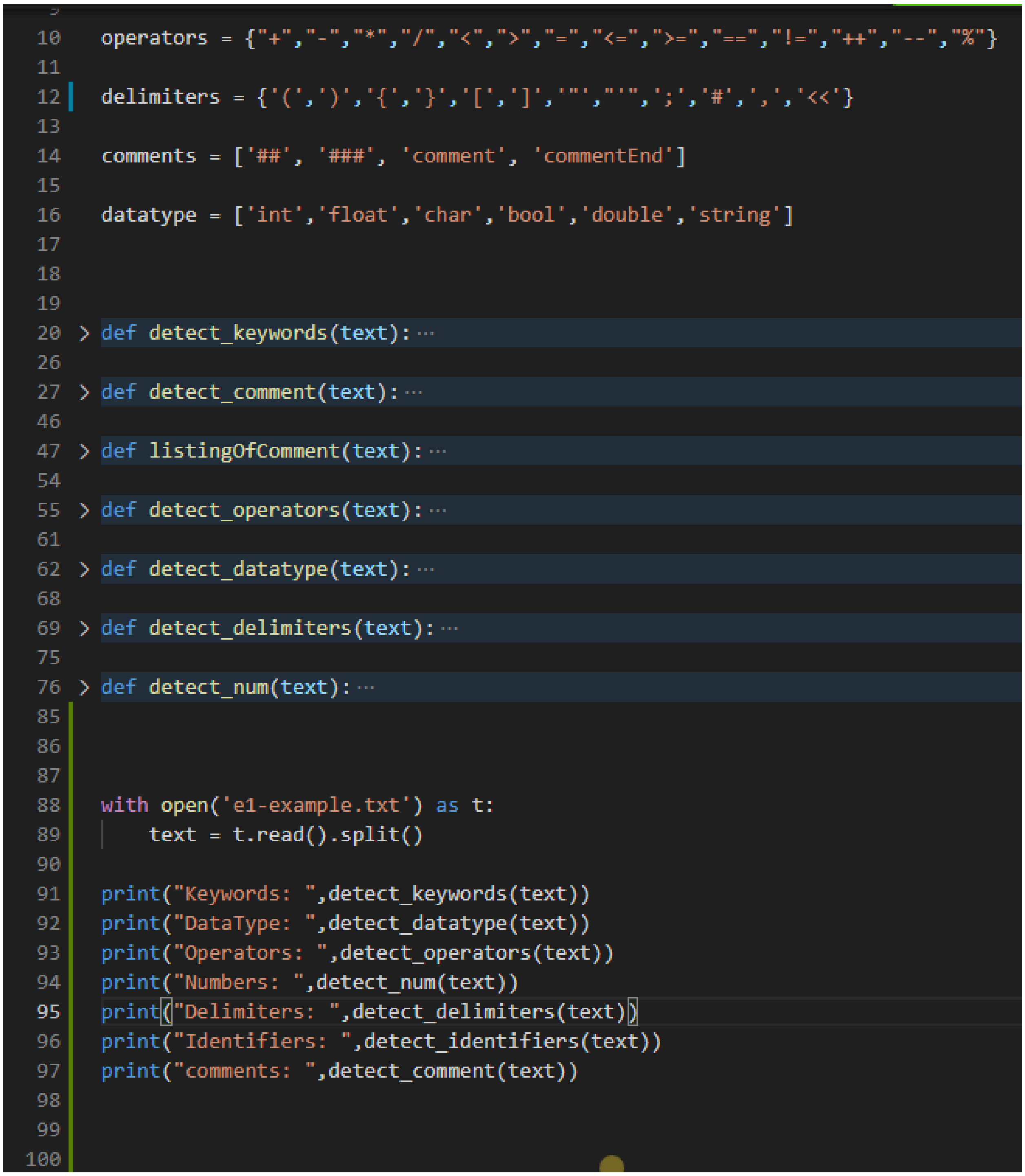
- A character sequence that cannot be scanned into any valid token is a lexical error.
- Lexical errors are uncommon, but they still must be handled by a scanner.

- Misspelling of identifiers, keyword, 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.

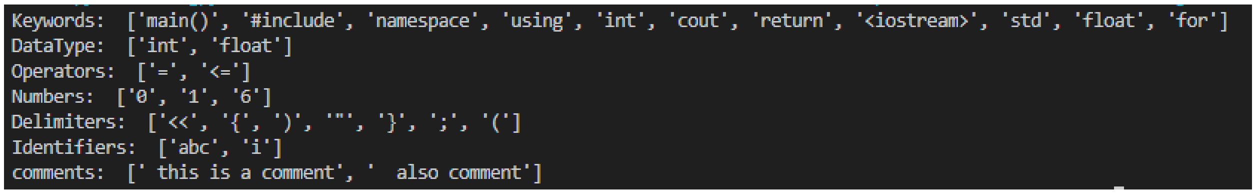
Code from project:

Scanning and Tokenization :



```python
operators = {"+","-","*","/","<",">","=","<=",">=","==","!=","++","--","%"}

delimiters = {'(',')','{','}','[',']','"',"'",';','#',',','<<'}

comments = ['##', '###', 'comment', 'commentEnd']

datatype = ['int','float','char','bool','double','string']



def detect_keywords(text): ···

def detect_comment(text): ···

def listingOfComment(text): ···

def detect_operators(text): ···

def detect_datatype(text): ···

def detect_delimiters(text): ···

def detect_num(text): ···



with open('e1-example.txt') as t:
    text = t.read().split()

print("Keywords: ",detect_keywords(text))
print("DataType: ",detect_datatype(text))
print("Operators: ",detect_operators(text))
print("Numbers: ",detect_num(text))
print("Delimiters: ",detect_delimiters(text))
print("Identifiers: ",detect_identifiers(text))
print("comments: ",detect_comment(text))
```

Output



```
Keywords:  ['main()', '#include', 'namespace', 'using', 'int', 'cout', 'return', '<iostream>', 'std', 'float', 'for']
DataType:  ['int', 'float']
Operators:  ['=', '<=']
Numbers:  ['0', '1', '6']
Delimiters:  ['<<', '{', ')', '"', '}', ';', '(']
Identifiers:  ['abc', 'i']
comments:  [' this is a comment', '  also comment']
```

Syntax analyser

Syntax analysis is the second phase of 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 syntax analyzer which can also be termed as 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 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 source language.

• It calls the function getNextToken(), to notify the lexical analyzer to yield another token.

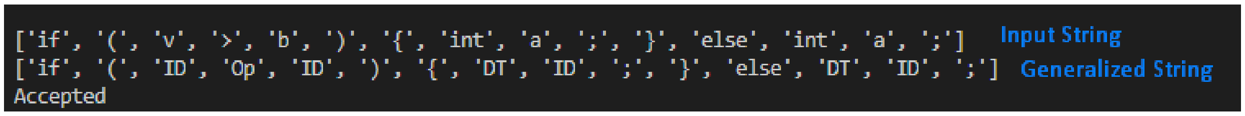
• It scans the token one at a time from left to right to construct the parse tree.

• It also checks the syntactic constructs of the grammar.

Function to make Generalize String for parsing

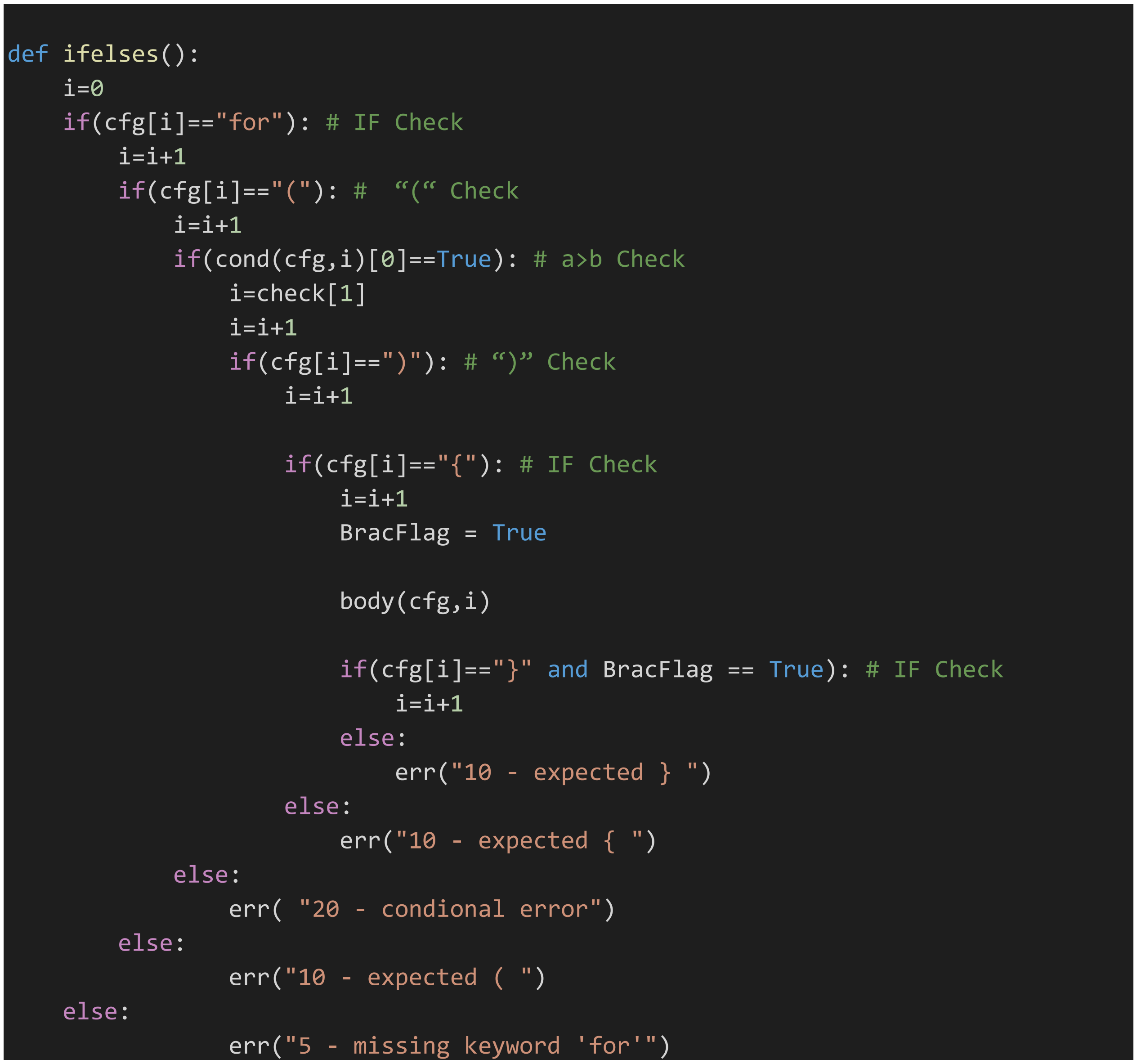
```python
def parseTree(value):
    if(value in datatype):
        return "DT"
    if(value == "="):
        return "Asop"
    if(value == "cout"):
        return "out"
    if(value in operators):
        return "Op"
    if(value == ","):
        return ","
    if(value in numbers):
        return "num"
    if(value==";"):
        return ";"
    if(value=="<<"):
        return "loop"
    if(value=="i++"):
        return "loop"
    if(value == ")"):
        return ")"
    if(value == "("):
        return "("
    if(value == "}"):
        return "}"
    if(value == "{"):
        return "{"
    if(value == "else"):
        return "else"
    if(value == "for"):
        return "for"
    if(value == '"'):
        return "StrOpen"
    if(value in alpha):           # if(value == alpha):    #right one
        return "ID"
    else:
        return "Error"
```

Output:

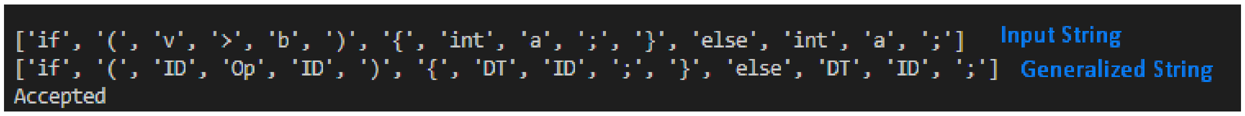


```
['if', '(', 'v', '>', 'b', ')', '{', 'int', 'a', ';', '}', 'else', 'int', 'a', ';']
['if', '(', 'ID', 'Op', 'ID', ')', '{', 'DT', 'ID', ';', '}', 'else', 'DT', 'ID', ';']
Accepted
```

Checking Syntax (Passing Validation):



```
def ifelses():
    i=0
    if(cfg[i]=="for"): # IF Check
        i=i+1
        if(cfg[i]=="("): #  “(“ Check
            i=i+1
            if(cond(cfg,i)[0]==True): # a>b Check
                i=check[1]
                i=i+1
                if(cfg[i]==")"): # “)” Check
                    i=i+1

                    if(cfg[i]=="{"): # IF Check
                        i=i+1
                        BracFlag = True

                        body(cfg,i)

                        if(cfg[i]=="}" and BracFlag == True): # IF Check
                            i=i+1
                        else:
                            err("10 - expected } ")
                    else:
                        err("10 - expected { ")
            else:
                err( "20 - condional error")
        else:
                err("10 - expected ( ")
    else:
                err("5 - missing keyword 'for'")
```



```
['if', '(', 'v', '>', 'b', ')', '{', 'int', 'a', ';', '}', 'else', 'int', 'a', ';']
['if', '(', 'ID', 'Op', 'ID', ')', '{', 'DT', 'ID', ';', '}', 'else', 'DT', 'ID', ';']
Accepted
```

```
Accepted is showing that the string was valid…
Successful… Parsing
```

Semantic analyser

• Semantic analysis is the third phase of compiler.

• It checks for the semantic consistency.

• Type information is gathered and stored in symbol table or in syntax tree.

• Performs type checking.

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 should be easy to produce.

• It should be easy to translate into target program.

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

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 uses the popular languages Java and C++ respectively.

Our 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.

Conclusion

In a compiler the process of Intermediate code generation is independent of 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 compilation process. It includes 3 phases of compilation lexical analysis, syntax analysis and semantic analysis which is then followed by intermediate 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.