**Python IDE**

Project Report

Course Code - CSE-3528

Course Title - Compiler Lab

**Submitted By**

Group - H

C201216 Kanita Haider

C201222 Rahanuma Zannat

C201224 Sabrina Mostary

C201228 Sanzida Hossain

C201239 Sadia Hasan Chowdhury

Semester - 5AF

**Submitted To**

Mrs. Israt Binteh Habib

Lecturer

Dept. of CSE, IIUC



Department of Computer Science and Engineering

International Islamic University Chittagong

Bangladesh

**INDEX**

1. **Introduction**………………………………………………………………………..**01**
2. **Problem statement**……….………………………………………………….**01**
3. **Methodology**…………………………………………………………………….**02**
4. **System design**……………………………………………………………..**04**
5. **System implementation** …………………………………………………………**05**
6. **System result**……………………………………………………………………...**06**
7. **Future Scope**……………………………………………………………………….**.08**
8. **Conclusion**………………………………………………………………………….**08**
9. **Appendix**………………………………………………………………………….**09**

**Introduction**

Compilers convert code from one language to another without affecting its meaning. Compilers should also optimize target code for time and space. It takes several steps to compile. Each compiler phase accepts input from the previous step, has its own source program representation, and outputs to the next phase. The compiler analyzes the source code and checks for lexical, grammatical, and syntax issues. Assembly Language Code is generated during analysis. Compilers turn text programs into executable ones. Manually installing and configuring a compiler on every system takes up a lot of space. Compiling makes a software platform-dependent. If a single system isn't possible, it's hard to transfer computer code to many systems.

**Problem Statement**

Our proposed compiler is built by using Python, a high-level programming language which can be easily understandable to us but not to machines. Therefore, we must convert our Python code to machine language. We constructed a compiler to turn our source code into executable code.C and C++ compilers convert straight to machine language, whereas Python compilers translate to byte code. Later, an interpreter converts byte code to machine language. That's why Python is an interpreted language. Since the interpreter reads instructions line by line and generates output till the point the code is correct, the ease of debugging increases as it is easier to get information about the source point of error. The size of programs written in Python is less as compared to other languages.

**Methodology**

* 1. **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.

* 1. **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.

* 1. **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.

* 1. **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.

* 1. **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.

**t1 = inttofloat (5)**

**t2 = id3\* tl**

**t3 = id2 + t2**

**id1 = t3**

### **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.

**System Design**

Python provides various options for developing graphical user interfaces (GUIs). To design our Python IDE , we need a Python compiler . For the user interface design, we have used Python and Tkinter.

Tkinter is the standard GUI library for Python. Python when combined with Tkinter provides a fast and easy way to create GUI applications. Tkinter provides a powerful object-oriented interface to the Tk GUI toolkit.

Creating a GUI application using Tkinter is an easy task. All you need to do is perform the following steps −

Import the Tkinter module.

Create the GUI application main window.

Add one or more of the above-mentioned widgets to the GUI application.

Enter the main event loop to take action against each event triggered by the user.

Tkinter provides various controls, such as buttons, labels and text boxes used in a GUI application. These controls are commonly called widgets.

For the compiler’s system design, we followed the Agile design method. The Agile methodology is a practice that encourages continuous development and testing throughout the software development lifecycle of a project. Unlike the Waterfall design, the Agile methodology allows for parallel development and testing.

**System Implementation**

Using Python language,we have implemented the following code in PyCharm to develope our Python IDE. It provides the IDE for client to compile their program.

**Code:**

from tkinter import \*

from tkinter.filedialog import asksaveasfilename, askopenfilename

import subprocess

compiler = Tk()

compiler.title('PyComp')

file\_path = ''

def set\_file\_path(path):

global file\_path

file\_path = path

def open\_file():

path = askopenfilename(filetypes=[('Python Files', '\*.py')])

with open(path, 'r') as file:

code = file.read()

editor.delete('1.0', END)

editor.insert('1.0', code)

set\_file\_path(path)

def save\_as():

if file\_path == '':

path = asksaveasfilename(filetypes=[('Python Files', '\*.py')])

else:

path = file\_path

with open(path, 'w') as file:

code = editor.get('1.0', END)

file.write(code)

set\_file\_path(path)

def run():

if file\_path == '':

save\_prompt = Toplevel()

text = Label(save\_prompt, text='Please save your code')

text.pack()

return

command = f'python {file\_path}'

process = subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)

output, error = process.communicate()

code\_output.insert('1.0', output)

code\_output.insert('1.0', error)

menu\_bar = Menu(compiler)

file\_menu = Menu(menu\_bar, tearoff=0)

file\_menu.add\_command(label='Open', command=open\_file)

file\_menu.add\_command(label='Save', command=save\_as)

file\_menu.add\_command(label='Save As', command=save\_as)

file\_menu.add\_command(label='Exit', command=exit)

menu\_bar.add\_cascade(label='File', menu=file\_menu)

run\_bar = Menu(menu\_bar, tearoff=0)

run\_bar.add\_command(label='Run', command=run)

menu\_bar.add\_cascade(label='Run', menu=run\_bar)

compiler.config(menu=menu\_bar)

editor = Text()

editor.pack()

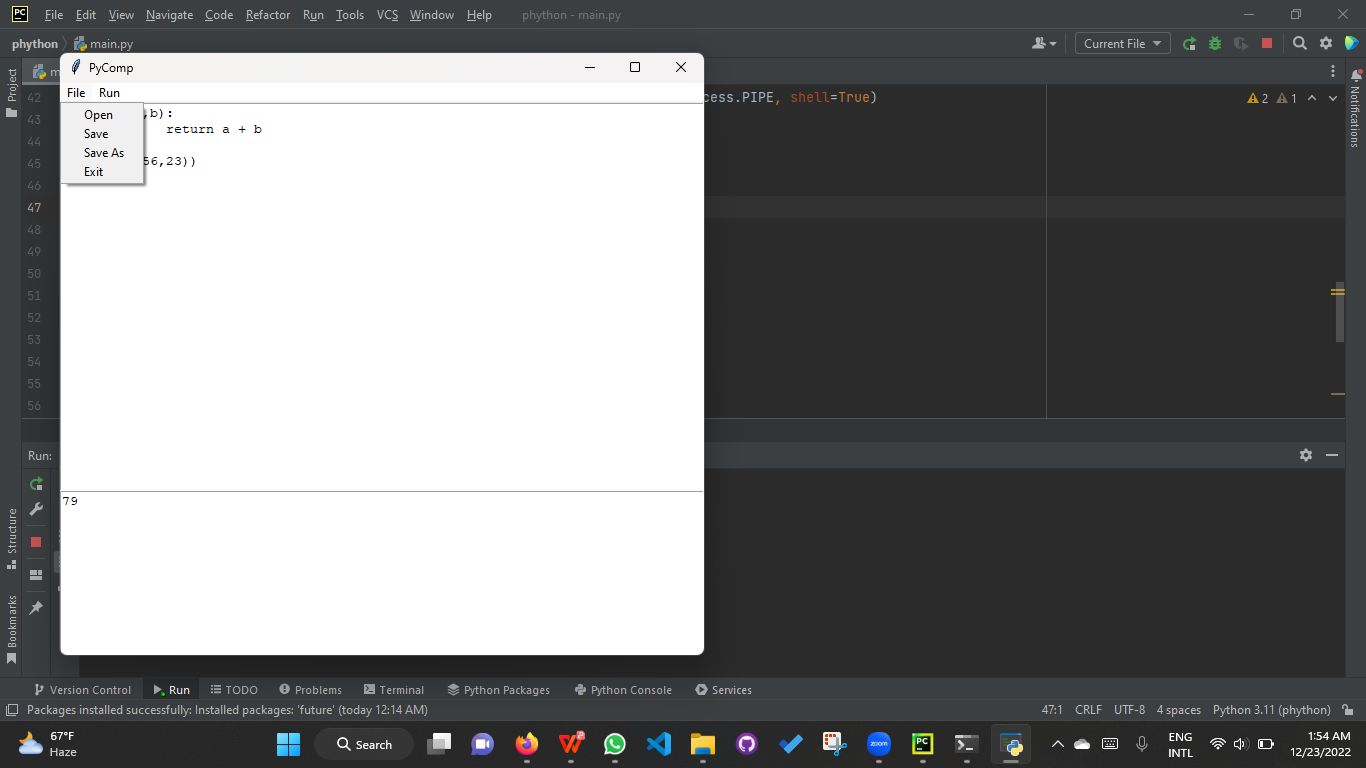
code\_output = Text(height=10)

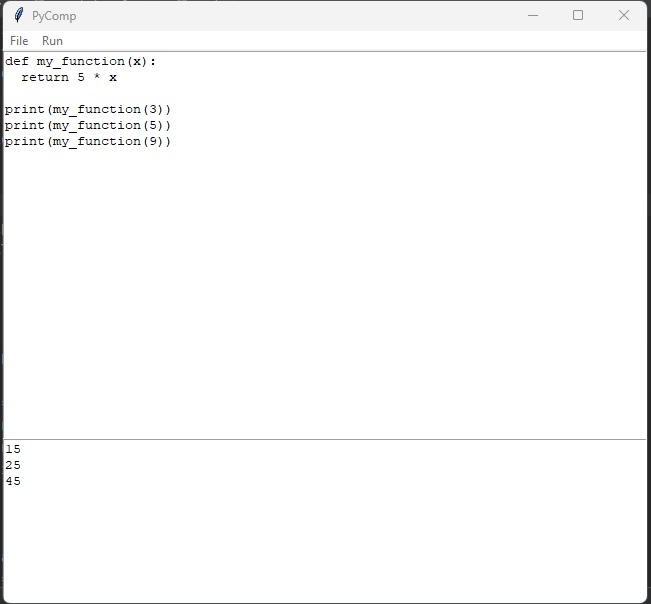
code\_output.pack()

compiler.mainloop()

**System Result**

As we can see the sneak peek of our IDE , here we can create new file , write code , save the code and run. After clicking the run, our IDE will compile the code and show the result in the output section below. If there is any error in the code , the errors will also be shown . We can also open other Python file in the compiler .

****

****

**Future Scope**

Our Python IDE has some deficiency in present as it’s now in a primary level.

* For now we have built our compiler in ‘PyCharm’ but in future we will launch our own IDE independently.
* In present , our compiler is appropriate only for Python language . In future we will add extentions for other languages like C, C++, C#, Javascript, Java etc.
* In our compiler there is no settings system for now ,we will add this in future.

**Conclusion**

A python compiler's intermediate code generation and target code conversion processes are both independent of the machine and the language being used, respectively. Consequently, we completed the front end of the compilation process. the process by which a compiler converts source code into object code in computer programming. This report describes the analytical stage of the creation of a Python compiler. Assembly level language is used for its implementation and source language.