**COMPILER CONSTRUCTION**

**LAB TERMINAL**

**Student Name:** Hafza Nawaz

**Registration no:** FA21-BCS-018

**Teacher Name:** Sir Bilal Ahmad

**Question :**

**Input output of your mini compiler?**

**Answer:**

Input: Source Code in the GUI

Source Code Text Area:

**Primary Input Method:** The primary input for mini-compiler is the source code entered by the user within the text editor part of the GUI.

**Language**: This code is written in the simplified C-like language that your compiler is designed to process. The language includes keywords like int, if, else, while, return, and print. It supports variable declarations, assignments, basic arithmetic expressions, conditional and looping statements, and function calls.

**Text Format**: The code is plain text. Your compiler's lexer and parser must be capable of correctly analyzing this plain text.

**File Extension**: The files used to enter the source code in the GUI are usually considered .txt files rather than .c.

**How the GUI Handles Input:**

**Text Widget:** The GUI (implemented in main.py) utilizes a text widget (like Tkinter's Text widget) where users can type or paste their source code.

No External File Loading: Based on the code, it does not appear that the GUI provides direct functionality for loading source code from an external file. Instead, the user directly types or pastes code into the text widget.

Button Trigger: The GUI will typically have a "Compile" or "Run" button or similar, which, when clicked, triggers the compilation process. The source code within the text widget is extracted and passed to the lexical analysis.

**Example of Valid Input**

int sum(int a, int b){

return a+b;

}

int main() {

int x = 10;

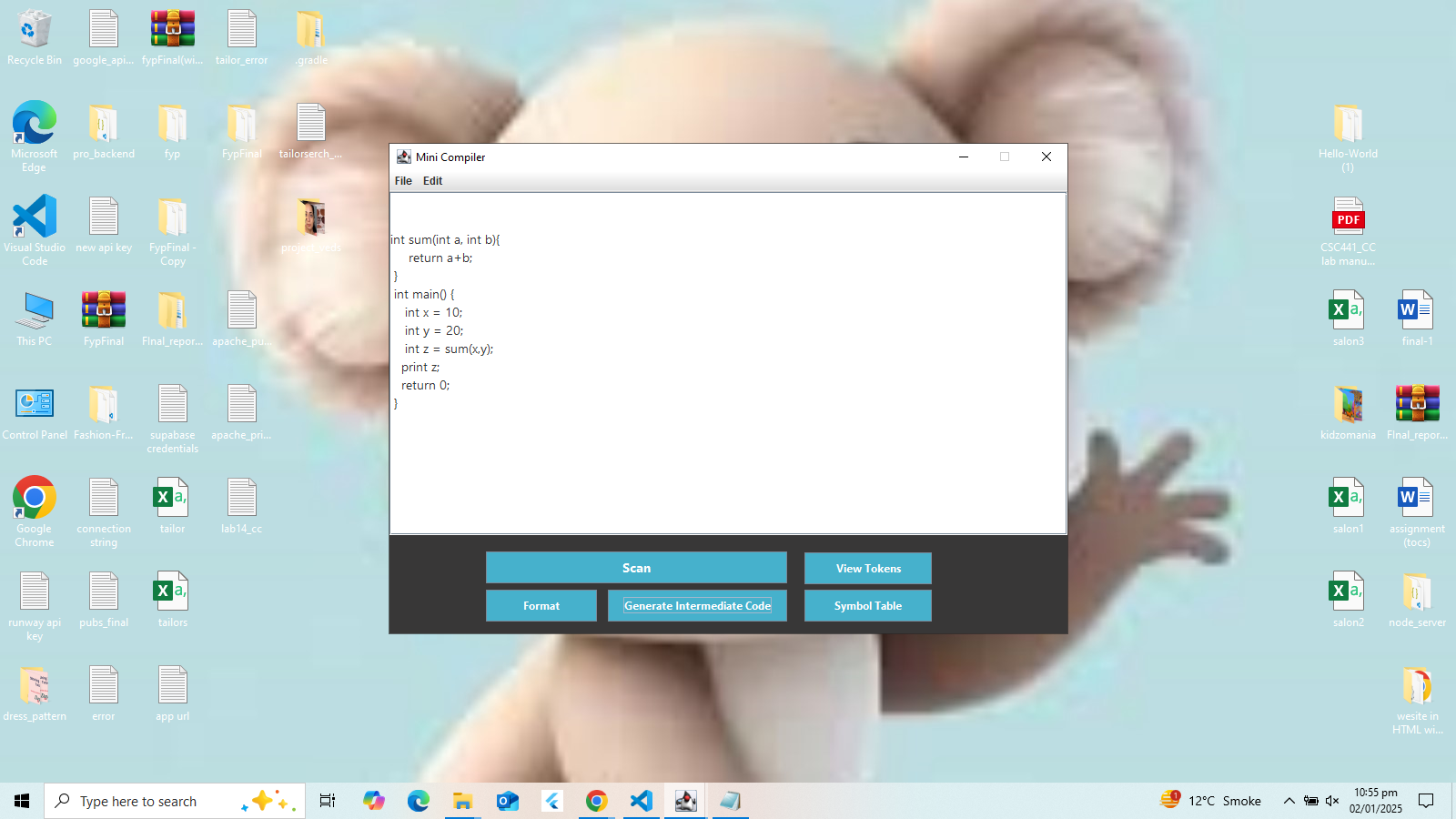
int y = 20;

int z = sum(x,y);

print z;

return 0;

}



Output: Assembly Code, Errors, GUI Feedback

Assembly Code (Text Output):

Target Machine: The compiler generates assembly code for a hypothetical machine architecture. This assembly code is not for a real processor but for some basic set of instructions.

Format: The generated assembly code is in plain text format and shown in the GUI output text area.

Content: The assembly code will consist of instructions for moving data into registers, performing arithmetic operations, executing control flow statements (e.g., conditional jumps and loops), calling functions etc.

No Separate File: The generated assembly is not written into a separate file (like .asm). Instead, it's directed to the GUI's output area.

Error Messages:

Types of Errors:

**Lexical Errors:** Errors in the structure of tokens in the input (e.g., invalid characters) will be displayed.

**Syntax Errors:** Errors in the structure of code with respect to the grammar (e.g., missing semicolons, incorrect order of tokens) will also be shown.

**Semantic Errors:** Errors related to the meaning of the code like undeclared variables will be outputted.

Format: Error messages are typically plain text strings. They provide information like:

Line Number and Column Number: Where the error occurred in the source code.

Error Type or Description: A brief explanation of the error.

GUI Display: Error messages are displayed within the GUI's output area along with the assembly output. This allows the user to identify and correct errors.

**GUI Feedback:**

Button Clicks: Buttons like "Compile" will have a visual change or animation when pressed.

**Status Messages**: Status messages indicating the progress or success/failure of the compilation process may be present.

**Output Area:** The output text area in the GUI not only shows the assembly code but also displays error messages, compiler warnings, and any other relevant compiler output.

Example of Output

If you input the source code above, and assuming there are no compilation error, then you will see an output similar to the below code

; Assembly code for program

main:

; Function body of main

; allocate space on stack

; x = 10

load\_int 10 R0

store R0, [fp - 4]

; y = 20

load\_int 20 R0

store R0, [fp - 8]

; function call sum(x,y)

load\_var [fp-4], R0

push R0

load\_var [fp-8], R0

push R0

call sum

add\_sp 2

store R1, [fp-12]

; print z

load\_var [fp-12], R0

print R0

load\_int 0 R0

return R0

sum:

; Function body of sum

; load a into R0

load\_var [fp + 4], R0

;load b into R1

load\_var [fp + 8], R1

add R0, R1

return R0

