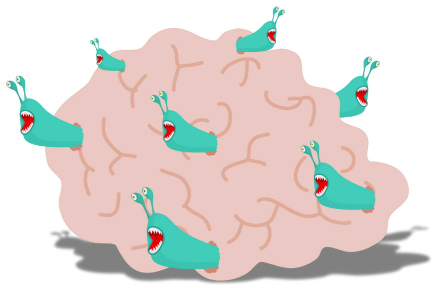
University of the Philippines Cebu

College of Science

Department of Computer Science

CMSC 124 Project: BrainRot Programming Language and Compiley Studio IDE



Creators:
Adrian Vaflor
Leian Carl Dela Cruz
Kimberly Padilla

Submitted To:
Dr. Robert Roxas

PART I
Rotting for Beginners: Writing Your First Lines in BrainRot

What is BrainRot?

BrainRot is a programming language whose syntax is inspired by Gen-Z and Gen Alpha terms and slangs. The language behaves just like C++, so, if you're familiar with C++'s syntax and logic, you'll pick up Brainrot quickly.

Inspiration

The idea for BrainRot was born from a mix of humor and fascination with the evolving language of the younger generations. The concept of "brain rot" reflects the kind of infectious humor that spreads through absurd, made-up words and phrases that resonate with Gen-Z and Gen Alpha. It's a playful nod to how these generations find creativity in the bizarre, creating a form of language that's both relatable and unintentionally comedic.

Adding to its relevance, "brain rot" was recently declared **Word of the Year**, solidifying its significance in 2024. This widespread recognition further highlights its influence and usage, making it the perfect foundation for a language designed to appeal to modern humor and trends. BrainRot is not just a language—it's a celebration of the quirky and ever-evolving creativity of the current generation.

Compiley Studio

Compiley Studio is your go-to platform for writing, running, and debugging **BrainRot Code**—this studio helps you bring your BrainRot programs to life.

Getting Started

If you want to try the app, go to https://github.com/DaRainFlavor/CMSC-124-PL-Project and download the zip file of the project. Extract the zip file on your computer, read the INSTRUCTIONS.txt , and double click the file “Compiley Studio.exe” and done.

If you want to clone and open the project in your IDE, consider the following.

I. Requirements

- Python: Make sure Python is installed on your system. You can download it from python.org. Download Python 3.11.1.
- pip: Ensure **pip** is available for managing Python packages.

II. Installation

1. Clone the Repository

Navigate to the cloned repository directory and start the IDE:

```
git clone https://github.com/DaRainFlavor/CMSC-124-PL-Project.git
```

2. Install dependencies

```
pip install -r Requirements.txt
```

3. Run the IDE

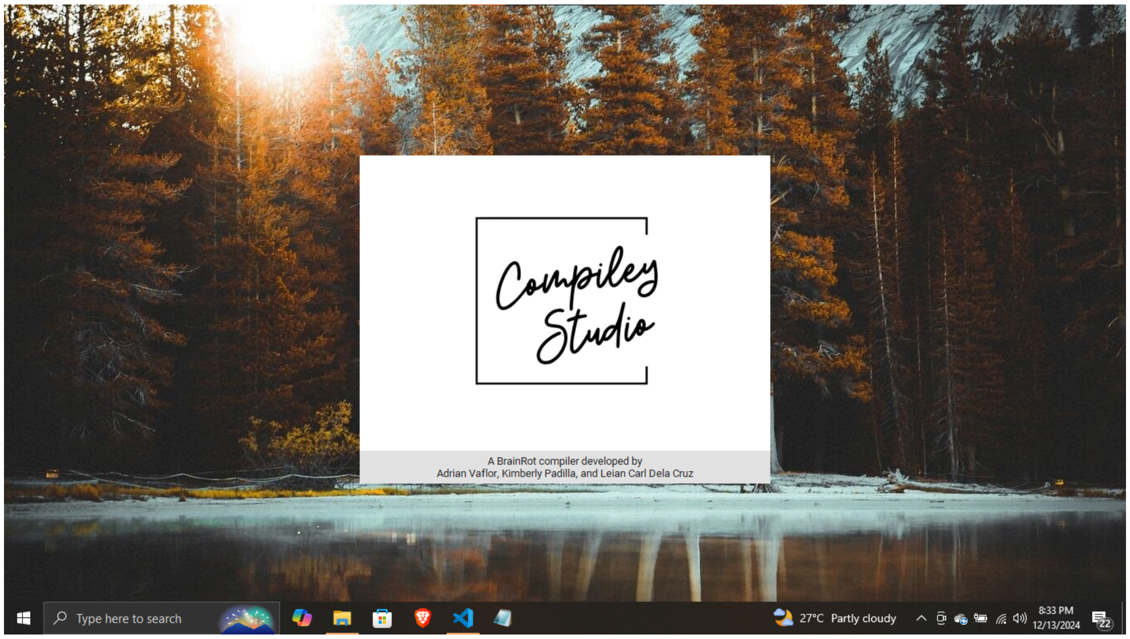
Navigate to the cloned repository directory and start the IDE:

```
cd <repository-folder>
py main.py
```

IDE Guide

I. Flash Screen

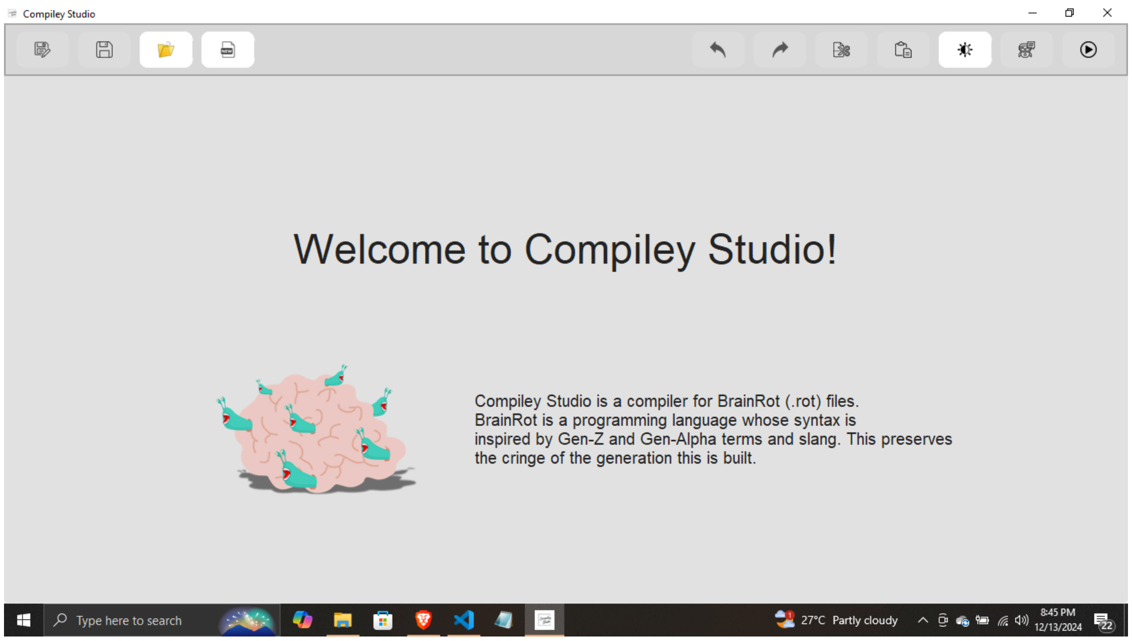
- The first screen you will see when they open the Compiley Studio. It provides a 3-second introduction to the tool, showcasing the logo and the creators.



A BrainRot compiler developed by
Adrian Vaflor, Kimberly Padilla, and Leian Carl Dela Cruz

II. Welcome Window

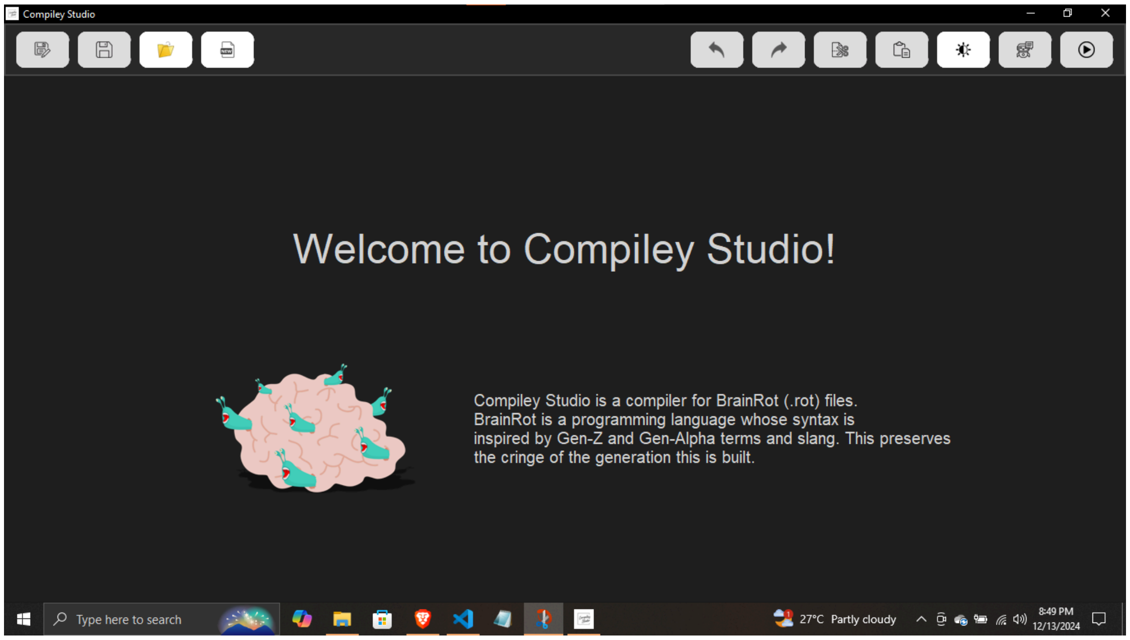
- The Welcome Window shows the main menu of the IDE. Click the button with the sun icon to switch themes. A tooltip also shows when you hover a button.



Welcome to Compiley Studio!

Compiley Studio is a compiler for BrainRot (.rot) files.
BrainRot is a programming language whose syntax is
inspired by Gen-Z and Gen-Alpha terms and slang. This preserves
the cringe of the generation this is built.

Dark Theme

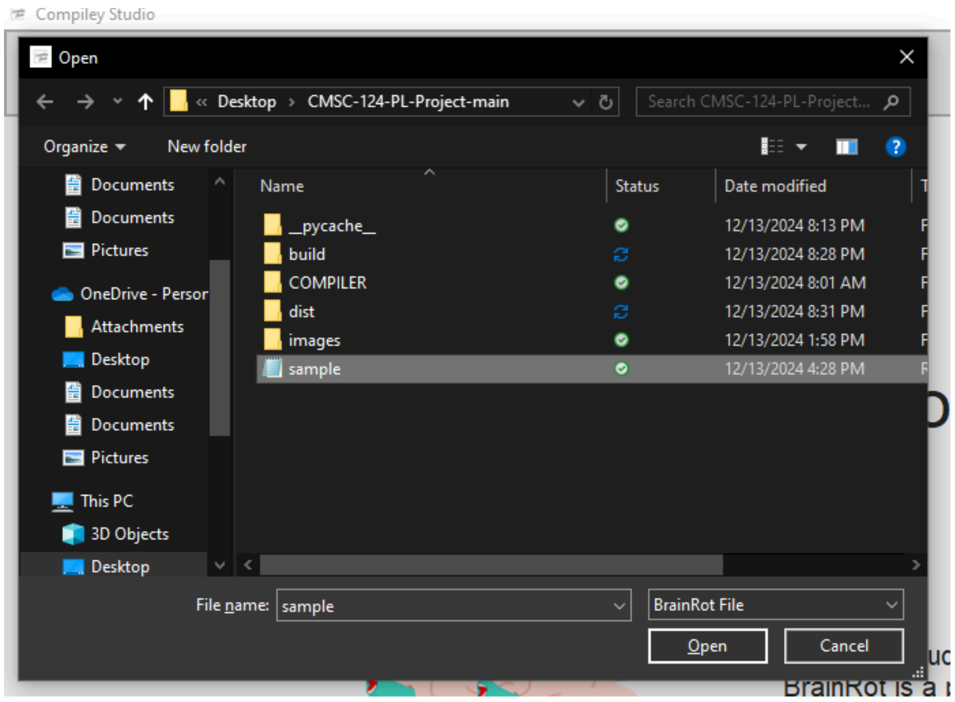


Welcome to Compiley Studio!

Compiley Studio is a compiler for BrainRot (.rot) files. BrainRot is a programming language whose syntax is inspired by Gen-Z and Gen-Alpha terms and slang. This preserves the cringe of the generation this is built.

III. Opening A File

Let's try opening a ready-made file to test the IDE. Click the Open button, go to the same folder you downloaded and select sample.rot



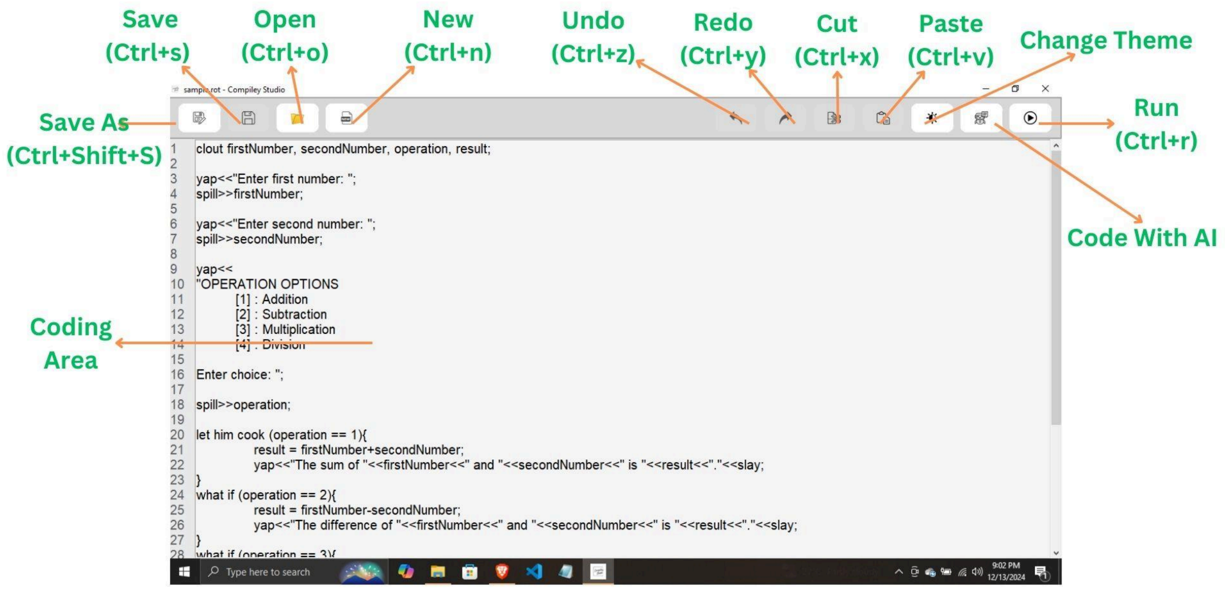
Name	Status	Date modified
__pycache__	✓	12/13/2024 8:13 PM
build	⟳	12/13/2024 8:28 PM
COMPILER	✓	12/13/2024 8:01 AM
dist	⟳	12/13/2024 8:31 PM
images	✓	12/13/2024 1:58 PM
sample	✓	12/13/2024 4:28 PM

File name: sample

BrainRot File

IV. Text Editor

- The Text Editor is the heart of the IDE. It provides all the functionalities you needed to code with each parts labeled below:



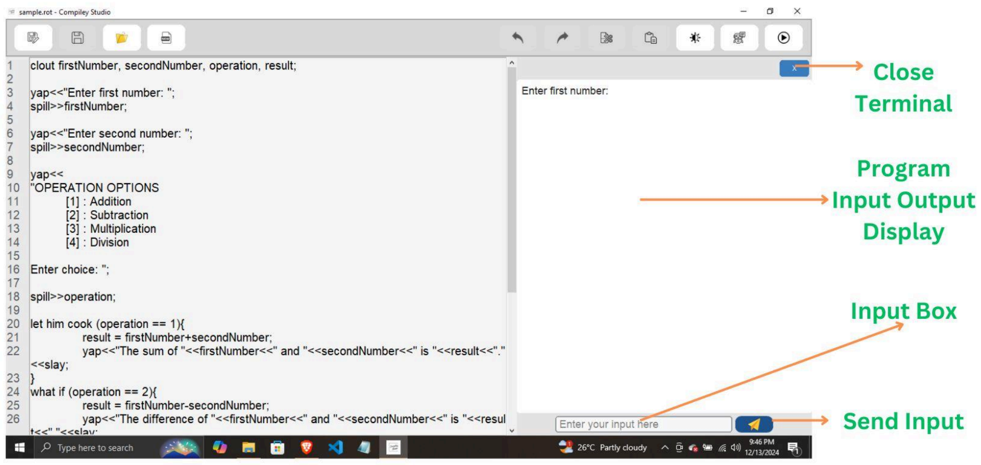
```
clout firstNumber, secondNumber, operation, result;

yap<<"Enter first number: ";
spill>>firstNumber;

yap<<"Enter second number: ";
spill>>secondNumber;

yap<<
"OPERATION OPTIONS
        [1] : Addition
        [2] : Subtraction
        [3] : Multiplication
        [4] : Division

Enter choice: ";

spill>>operation;

let him cook (operation == 1){
            result = firstNumber+secondNumber;
            yap<<"The sum of "<<firstNumber<<" and "<<secondNumber<<" is "<<result<<"."<<slay;
}
what if (operation == 2){
            result = firstNumber-secondNumber;
            yap<<"The difference of "<<firstNumber<<" and "<<secondNumber<<" is "<<result<<"."<<slay;
}
what if (operation == 3){
```

Name	Description
Coding Area	Area where you type BrainRot codes.
Save As	Saves the current file under a new name or location.
Save	Saves the current file with any changes made.
Open	Open an existing file from your system for editing.
New	Creates a new, blank file in the editor for starting a fresh project.
Undo	Reverts the most recent action in the editor.
Redo	Re-applies the most recent undone action.
Cut	Removes selected text and copies it to the clipboard.
Paste	Inserts text from the clipboard at the cursor's position.
Change Theme	Toggles between light and dark modes for the

	editor interface.
Code With AI	Opens a chat-box and allows you to code by typing or speaking directly the prompts and the AI will generate the code asked.
Run	Autosaves the current file, opens the terminal and executes the code.

V. Executing The Code

sample.rot file simply asks the user two numbers and an operation to perform with those two numbers. Let's check if this works. Click the Run button and a terminal should appear.



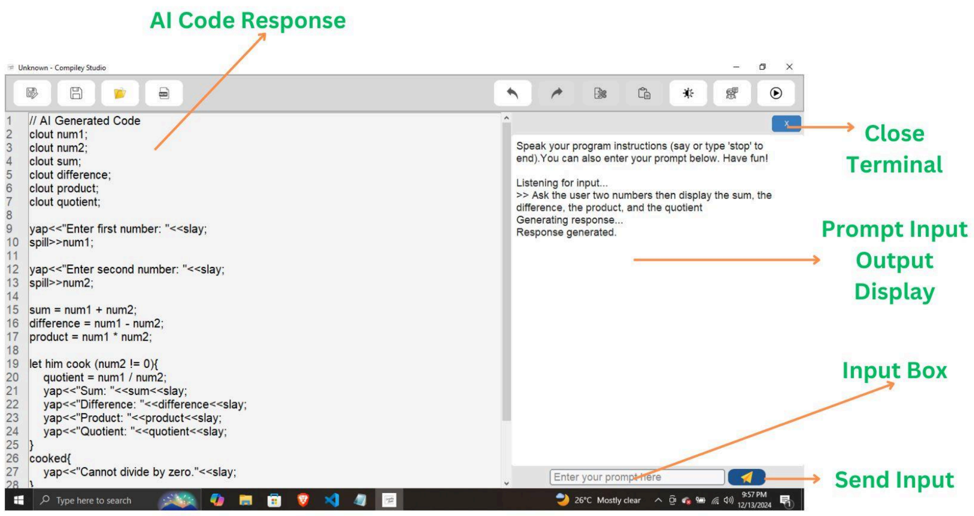
```
clout firstNumber, secondNumber, operation, result;

yap<<"Enter first number: ";
spill>>firstNumber;

yap<<"Enter second number: ";
spill>>secondNumber;

yap<<
"OPERATION OPTIONS
        [1] : Addition
        [2] : Subtraction
        [3] : Multiplication
        [4] : Division

Enter choice: ";

spill>>operation;

let him cook (operation == 1){
            result = firstNumber+secondNumber;
            yap<<"The sum of "<<firstNumber<<" and "<<secondNumber<<" is "<<result<<"."
<<slay;
}
what if (operation == 2){
            result = firstNumber-secondNumber;
            yap<<"The difference of "<<firstNumber<<" and "<<secondNumber<<" is "<<resul
t<<" "<<slay;
```

```
Enter first number:
```

I'll let you finish the program.

VI. Code with AI

There are times that typing code is tiring. Compiley Studio offers a more convenient way of coding and it is with the help of artificial intelligence. All you have to do is to ask what you need to code through direct speaking or through typing commands. Click the Code With AI button and check this out!



```
// AI Generated Code
clout num1;
clout num2;
clout sum;
clout difference;
clout product;
clout quotient;

yap<<"Enter first number: "<<slay;
spill>>num1;

yap<<"Enter second number: "<<slay;
spill>>num2;

sum = num1 + num2;
difference = num1 - num2;
product = num1 * num2;

let him cook (num2 != 0){
    quotient = num1 / num2;
    yap<<"Sum: "<<sum<<slay;
    yap<<"Difference: "<<difference<<slay;
    yap<<"Product: "<<product<<slay;
    yap<<"Quotient: "<<quotient<<slay;
}
cooked{
    yap<<"Cannot divide by zero."<<slay;
}
```

```
Speak your program instructions (say or type 'stop' to
end).You can also enter your prompt below. Have fun!

Listening for input...
>> Ask the user two numbers then display the sum, the
difference, the product, and the quotient
Generating response...
Response generated.
```

When Code With AI runs, it accesses the computer's microphone and if you don't want the AI to listen, just type (to the input Box) or say "stop". This feature is not only helpful to make writing programs faster, but it can also be used as a debugger by copy-pasting the code to the input box and asking the AI to fix it.

Brainrot Syntax Guide

Before creating your very first BrainRot program, here’s a quick comparison of C++ and BrainRot syntax.

C++	Brainrot	Description
int	clout	Integer data type.
string	sigma	String data type.
if	let him cook	Beginning a conditional statement.
else if	what if	Defines else if condition.
else	cooked	Defines else condition.
return	it’s giving	Ends the program.
cout	yap	Prints output to console.
cin	spill	Takes input from the user.
//	//	Single line comment.
/**/	/**/	Multiple line comment.
endl; \n	slay; \n; {hitting enter in keyboard}	Inserts new line in string.
\t	\t	Inserts tab in string.
\”	\”	Escapes “ in string.
\\	\\	Escapes \ in string.

Coding with BrainRot Tutorial

Now is the time to create your first BrainRot program.

I. Variable Declaration and Assignment

```
clout a;                  // Declares an integer variable, zero is the default value
clout b, c;               // Multiple declarations in one line
clout x = 10;             // Declares an integer variable with assignment
a = 3;                    // Assigns an integer to a variable
sigma str;                // Declares a string variable, “” is the default value
sigma name = "John Doe";  // Declares a string variable with assignment
```

II. Printing with Variables and Literals

```
clout a = 10;
clout b = 20;
sigma name = "Compiley Studio";

yap<<"The value of a is " << a << " and b is " << b << "."<<slay;

yap<< "Welcome to " << name << ", your IDE for BrainRot coding!" << slay;
```

Output:

```
The value of a is 10 and b is 20.
Welcome to Compiley Studio, your IDE for BrainRot coding!
```

III. Scanning of Input (Multiple Inputs in One Line)

```
clout x, y;
sigma username;
spill>>x;                          // Scanning 1 input
spill >> y >> username;            // Scanning multiple inputs in one line
```

IV. Operations

BrainRot is still under development, and it can only perform addition, subtraction, multiplication, and division to integer variables and integer literals, and only concatenation through addition among string variables, string literals, and slay. It is also restricted to only one operator per computation.

```
sigma a = "Lady", b = "bug";

sigma c = a+b;                          // both are variables
yap<<c<<slay;

sigma d = "Sun" + "flower";             // both are literals
yap<<d+slay;                            // new line and variable

yap<<5 * 5;                             // works same as string but can do +,-,/
```

Output:

```
Ladybug
Sunflower
25
```

V. If-Else Statement:

Just like the operations, there are also some constraints in conditional statements. Here are the key rules:

1. Conditions are limited to integers only.
2. Conditional operators allowed are ==, !=, <, >.
3. Conditions are strictly variable or literal or <variable> <conditional operator> variable
4. Strictly no string declarations but assignment and operation of strings is possible when string variables used are outside the condition.
5. No Nesting Allowed
6. Optional Bracket for single statement inside conditional statements

```
clout x = 10;
clout y = 15;

let him cook (x > y)
   yap << "x is greater than y" << slay;
what if (x < y)
   yap << "x is less than y" << slay;
cooked
   yap << "x is equal to y" << slay;
```

Output:

```
x is less than y
```

VI. Comments And Force End

```
clout x = 5, y = 0;
// yap<<"Single line comment";

clout quotient;

/*
yap<<"Multiple "<<slay;
yap << "lines"<<slay;
yap<<"comment"<<slay;
*/

let him cook(y == 0){
        yap<<"Cannot divide by zero!";
        it's giving;                    // End the program
}
quotient = x/y;
yap<<quotient;
```

Output:

```
Cannot divide by zero!
```

When using multiple line comments (/**/), make sure that there is no nesting of /**/ in order to make it work properly.

Error Messages

Familiarize yourself with the following error messages you might encounter.

1. **Unexpected Character**
   - This error is raised when an unexpected or some unexpected symbolic character is detected by the scanner.

```
Skibidi in Toilet [line number]: Unexpected rizz [illegalChar] in [word].
```

- This error is also raised when an unexpected keyword is encountered.

```
Skibidi in Toilet [line number]: Unexpected rizz: [keyword].
```

2. **Use of the Disallowed Symbol §**
   - This error is raised when the character **§** is found in the code, as it is a restricted symbol.

```
Skibidi in Toilet [line number] : you can’t rizz ‘ § ‘ . Yeet in [symbol error].
```

3. **Expected Data Type**
   - This error is raised when a statement or expression is missing a required data type.

```
Skibidi in Toilet [line number] : Expected data type.
```

4. **Redeclaration of Variable**
   - This error is raised when a variable is declared again in the same scope, which is not allowed.

```
Skibidi in Toilet [line number] : Redemption arc of {varName}.
```

5. **Undeclared Variable**
   - This error is raised when an undeclared variable is used in an expression or operation.

```
Skibidi in Toilet [line number] : {varName} is not it.
```

6. **Invalid Value**
   - This error is raised when incompatible data types are used together in an operation or assignment. This is also seen when printing non-variable, non-literal, and non-slay.

```
Skibidi in Toilet [line number] : Invalid fanum tax.
```

7. **Operator Expected**
   - This error is raised when a variable is used without being assigned a value.

```
Skibidi in Toilet [line number] : Rizzler Expected.
```

8. **Invalid String Operation**
   - This error is raised when an operator is missing between operands in an expression.

```
Skibidi in Toilet [line number] : Rizzler {operator} not supported on sigma.
```

9. **Division by Zero**
   - This error is raised when a division operation attempts to divide by zero (obviously typed) , which is undefined.

```
Skibidi in Toilet [line number] : Division by zero.
```

- For non-obvious division by zero.

```
Skibidi : Division by zero occurred.
```

10. **Type Mismatch**
    - This error is raised when incompatible data types are used together in an operation or assignment.

```
Skibidi in Toilet [line number] : Negative aura between {varName1}`: {datatype1} and
`{varName2}`: {datatype2}."      // Type mismatch
```

11. **Missing Closing Comment**
    - This error is raised when /* is not paired with */.

```
Skibidi in Toilet [line number] : Missing closing comment.
```

12. **Error in MIPS and Unknown Error**
    - This error appears when something went wrong with the MIPS. MARS simulator sends an error message. This usually happens when the code asks for an integer but non-numeric characters are passed.

```
Error in [Error message from MARS]
```

- This error exists but has not yet been encountered. If you encountered the following error, send us an email, this is likely caused by the compiler.

```
What the sigma!
```

PART II
BrainRot and Compiley Studio Deconstruction

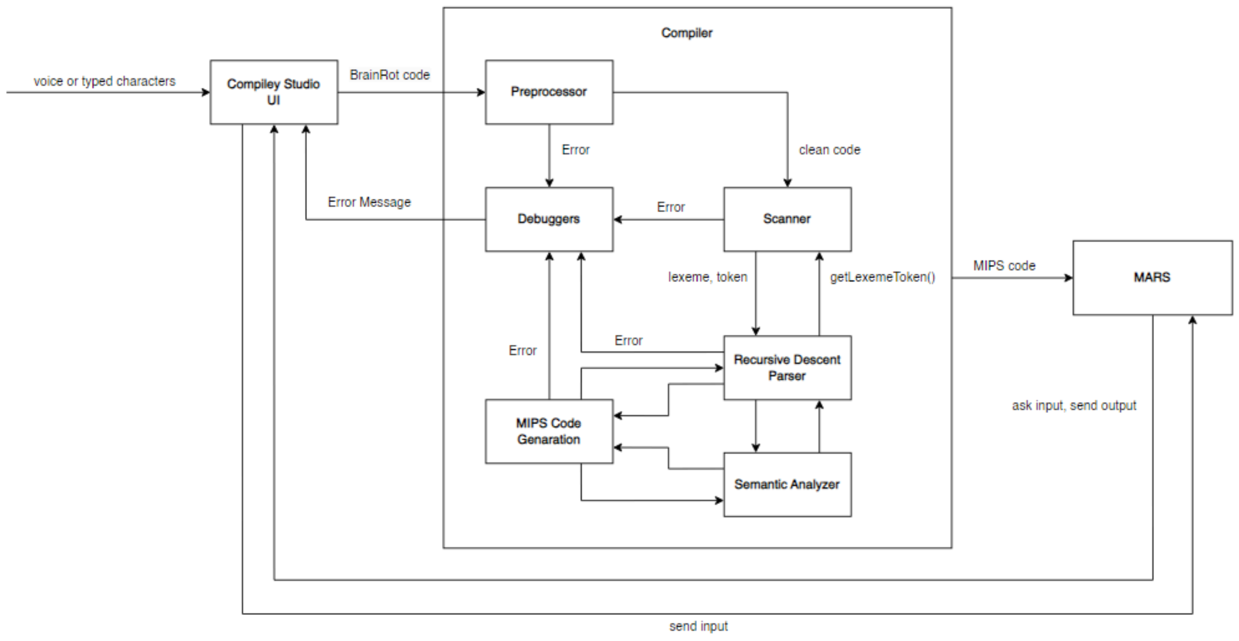


Figure 1. BrainRot Pipeline

Figure 1 shows the blueprint on how everything works. Observe that there are back-and-forth transitions among recursive descent parser, semantic analyzer, and MIPS code generation. This hints that the flow of processing the BrainRot code does not happen by completing each component first with the entire code processed. It processes code line by line or when a statement is complete for MIPS code generation to be already possible. Figure 1 will help us guide each component.

To view the literal table, symbol table, and MIPS code, open terminal, go to the same folder and run main.py. Those are printed in the terminal whenever you run a code.

The IDE UI

Compiley Studio is the integrated development environment (IDE) of BrainRot language. It is created in Python with the use of the Tkinter and CustomTkinter library for UI designs. These libraries power the interactive UI with widgets such as buttons, text areas, and file selectors, while CustomTkinter enhances the visual aesthetic with modern themes. Custom_hovertip was used for the hover effect of the buttons. Frames for text widgets are used for the code editor, with added syntax highlighting logic and for displaying terminal interactions for BrainRot execution and MIPS output. Additionally, the scroll package from Python is used for displaying scrolling text in the code editor. The IDE executes BrainRot code through Python’s subprocess module. A temporary file is made containing the MIPS code and then passed as an input to MIPS Assembler and Runtime Simulator (MARS). It uses the threading module to read the output from the Java process. The output would be then displayed in the terminal pane.

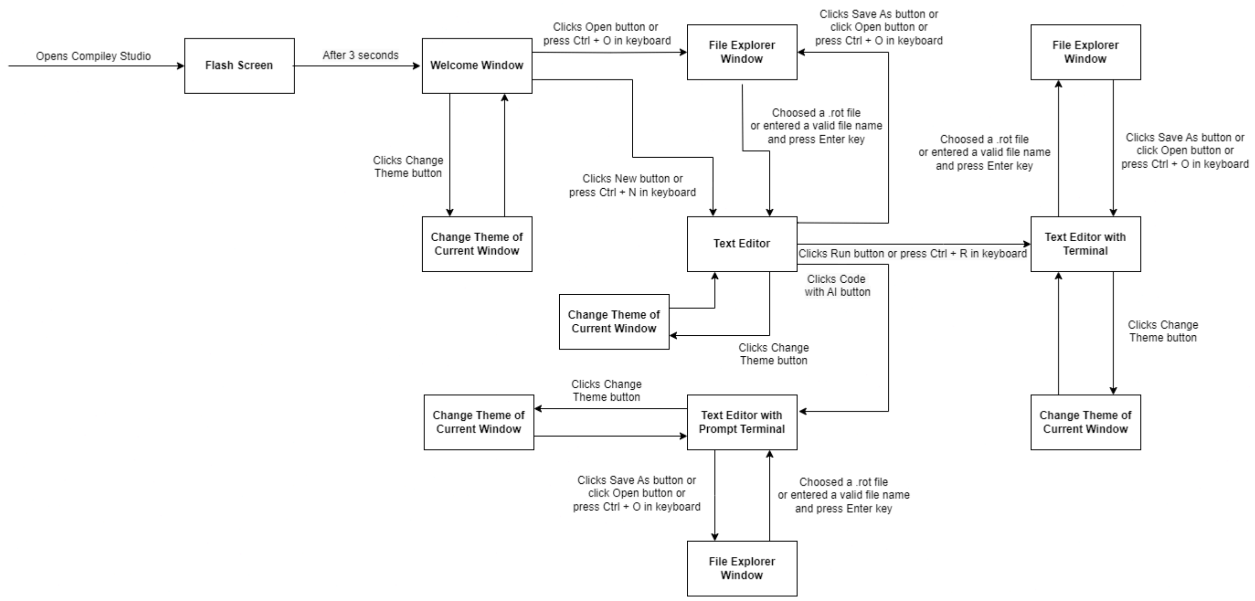

Opens Compiley Studio
Flash Screen
After 3 seconds
Welcome Window
Clicks Open button or press Ctrl + O in keyboard
File Explorer Window
Clicks Save As button or click Open button or press Ctrl + O in keyboard
File Explorer Window
Choosed a .rot file or entered a valid file name and press Enter key
Choosed a .rot file or entered a valid file name and press Enter key
Clicks Save As button or click Open button or press Ctrl + O in keyboard
Clicks Change Theme button
Clicks New button or press Ctrl + N in keyboard
Change Theme of Current Window
Text Editor
Clicks Run button or press Ctrl + R in keyboard
Text Editor with Terminal
Clicks Code with AI button
Change Theme of Current Window
Clicks Change Theme button
Clicks Change Theme button
Clicks Change Theme button
Change Theme of Current Window
Text Editor with Prompt Terminal
Change Theme of Current Window
Clicks Save As button or click Open button or press Ctrl + O in keyboard
Choosed a .rot file or entered a valid file name and press Enter key
File Explorer Window

Figure 2. Transition Diagram of Compiley Studio Between States

Flash Screen: Initializes the Compiley Studio and displays the startup screen for 3 seconds.
Welcome Window: Waits for user interaction, allowing users to create new files (Ctrl + N), open existing files (Ctrl + O), or change the interface theme.
File Explorer Window: Allows users to open or save .rot files by entering a valid file name or selecting an existing one.
Text Editor: Serves as the primary interface for writing code. Users can run their code (Ctrl + R), enable AI assistance, or customize the theme.
Text Editor with Terminal: Combines the text editor with a terminal for running code and viewing results.
Text Editor with Prompt Terminal: Provides an enhanced editor with additional terminal features and prompt-based interactions.
Change Theme of Current Window: Allows users to customize the appearance of the current interface.

Code with AI

Code With AI uses Gemini API to send instructions and receive codes. Below is the pipeline of Code With AI.

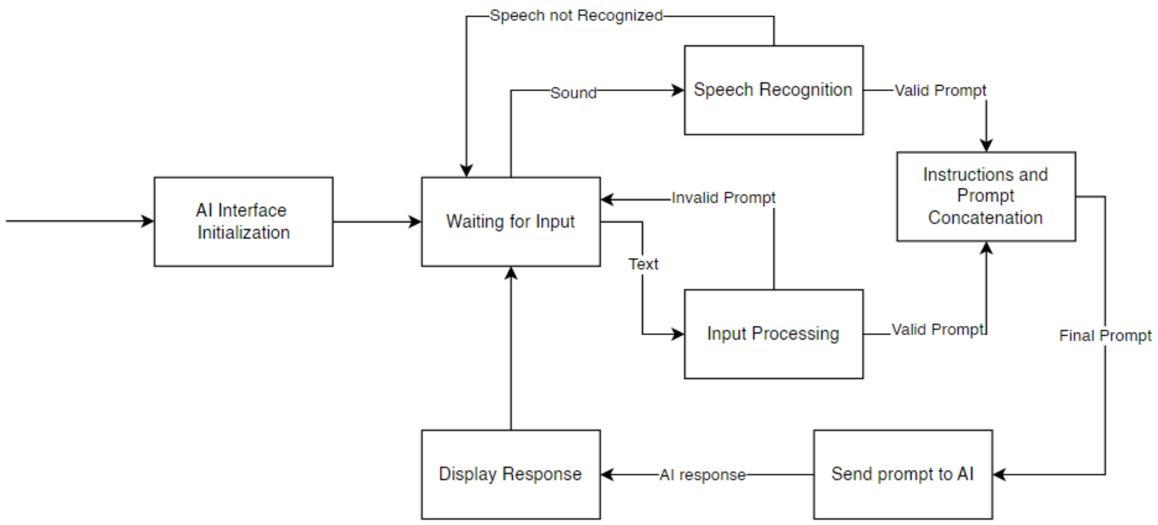


Figure 3. Transition Diagram of Code with AI Feature

AI Interface Initialization: initializes the Gemini API and the threading events
Waiting for Input: waits for the user interaction
Speech Recognition: Analyzes the sound heard and disables Speech recognition when speech recognized is “stop”.
Input Processing: Analyzes the user input if the prompt is valid or not. It also disables Speech recognition if the prompt is “stop”.
Instructions and Prompt Concatenation: Concatenates the user prompt and the instructions we told the AI on what to do with the user prompt.
Send prompt to AI: AI processes the input and formulates a reply
Display response: An arrow loops back to **Waiting for Input**, allowing for continuous interaction

The instructions we passed to AI just filter irrelevant prompts (gibberish, not asking for code) and generate a code in accordance with BrainRot grammar.

Preprocessor, Comments, and DFA Diagram

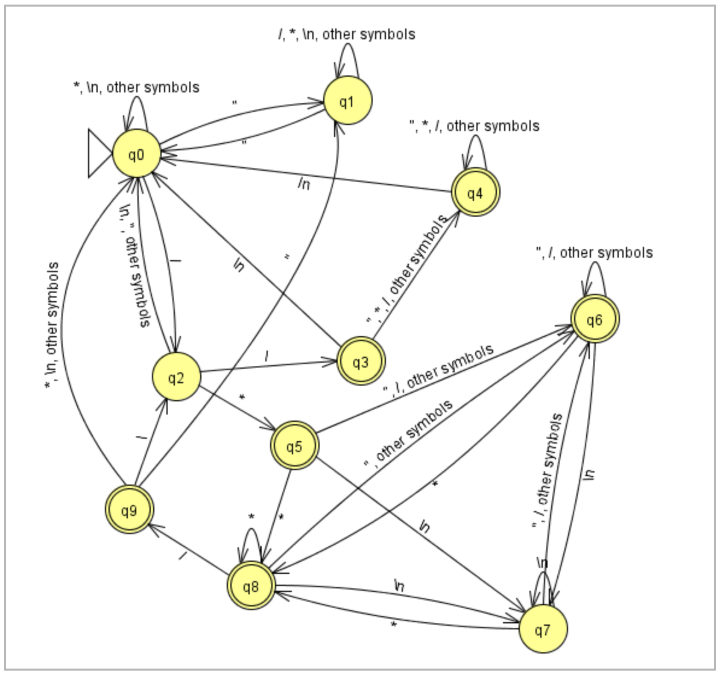


Figure 4. Preprocessor and comments DFA

Figure 4 shows a deterministic finite automata (DFA), however, the final states or the accepted states function differently from FSMS in Figures 5, 6, 7 and 8. When in states 3 and 5, it removes the previous 2 characters. When in states 4, 6, 8, and 9, it removes the string read. Notice that reading the character '\n' was specially transitioned to state 7. The reason for that is that it should not be removed. Preserving '\n' helps the debugger track the line number of code. This ultimately recognizes comments and passes the code to the scanner without unnecessary codes.

However, this FSM has a limitation. It can't handle nested multiple line comments like "/*/*Hello*/" as FSMs in nature, are not powerful enough to recognize the balance of starting and ending comment in a nested context. It still throws an error message, however, those do not specifically address a missing closing comment. A solution to this is by Pushdown Automata (PDA) implementation which will be part of the app's future updates.

Scanner and Finite State Machines

The lexical analyzer or scanner creates a lexer table holding values of token and lexeme. The scanner provides a lexeme and its equivalent token to the recursive descent parser for syntactic analysis.

1. **Single Character tokens**
   Reading the following symbols: semicolon, comma, open parenthesis, close parenthesis, open curly braces, close curly braces, equal, plus, minus, times, and divide (/) is not represented with FSM diagram because if the current index scanned is among those symbols, it returns right away that symbol and the corresponding token it represent. It does not involve any other transition of states.

2. **Keywords**

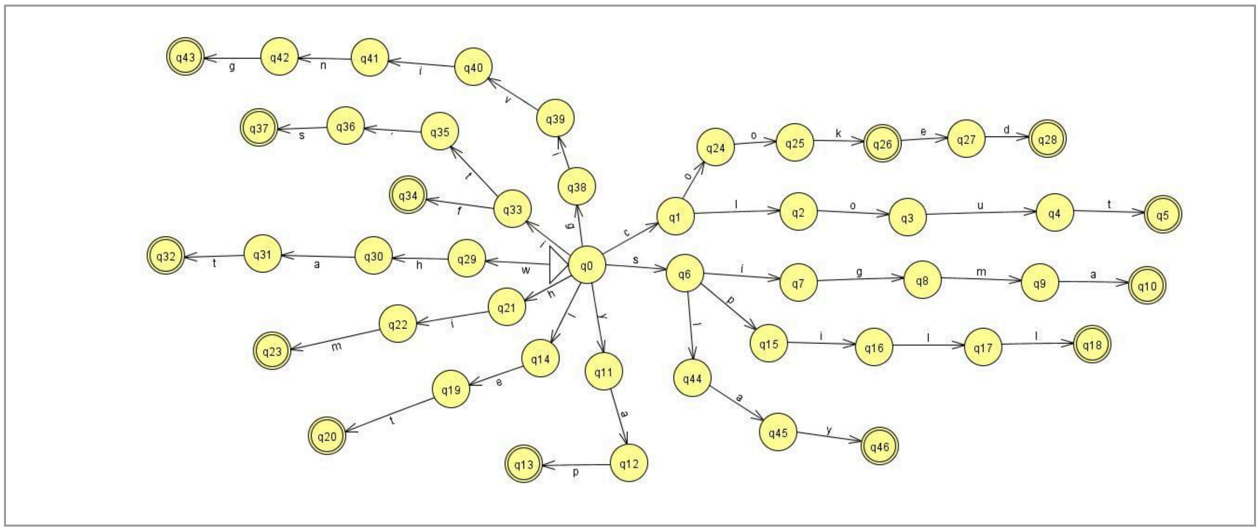


Figure 5. NFA for keywords

3. **Identifier**

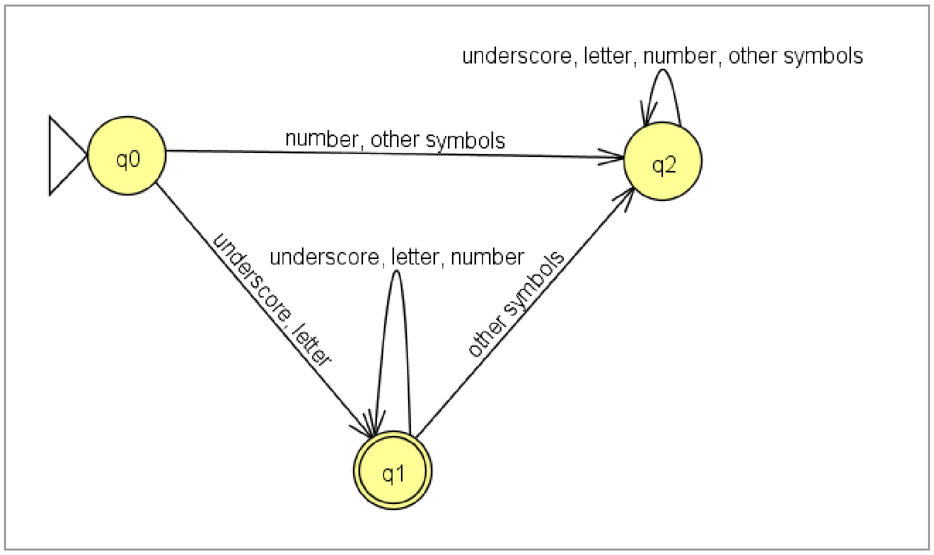

q0
q1
q2
number, other symbols
underscore, letter
underscore, letter, number
other symbols
underscore, letter, number, other symbols

Figure 6. DFA for identifiers

4. **Clout Literal** (Integer Literal)

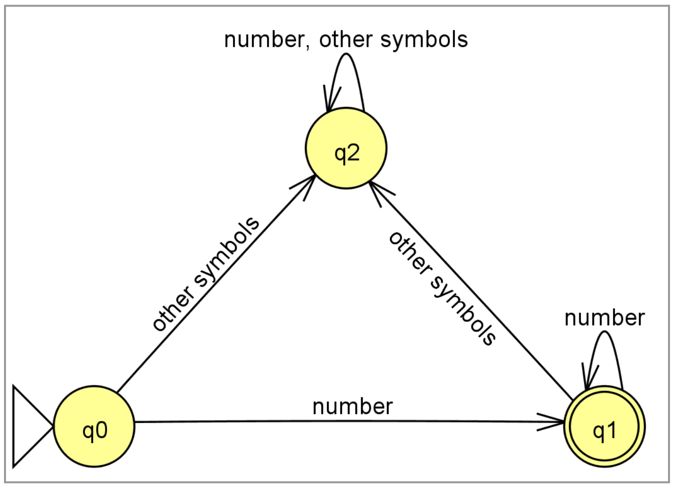


Figure 7. DFA for Clout Literal

You may ask “what about negative numbers?”, well, for making things simple, negative numbers are handled by the grammar. The decision was made because ‘-’ is also interpreted as a “MINUS” token. So, it is simpler to just make a grammar of it and multiply the number with -1 once read.

5. **Sigma Literal** (String Literal)

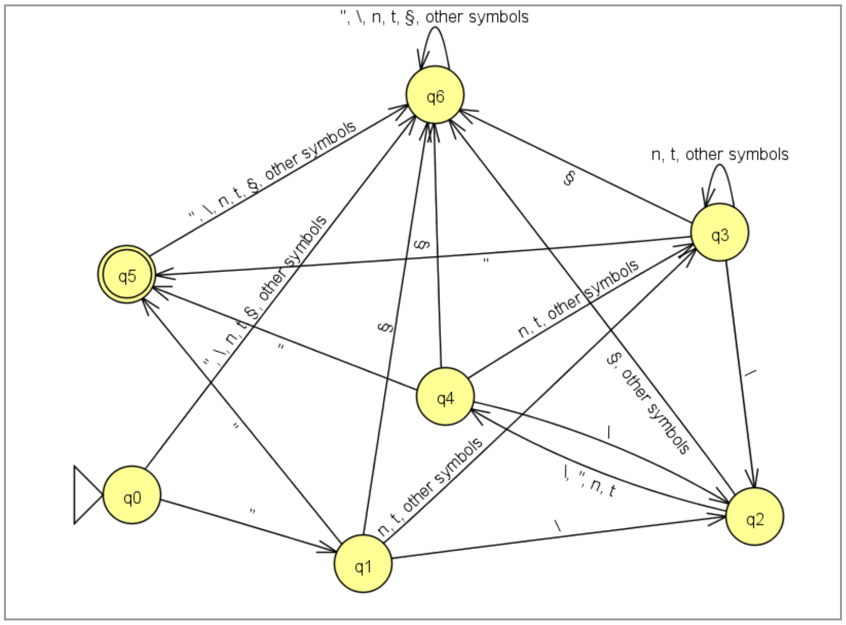


Figure 8. Sigma Literal

The character ‘§’ is specially added to the DFA in Figure 8 because it serves as a flag from MIPS code’s output to the IDE’s terminal. Thus, this special character can’t be anywhere, specifically in sigma literals to avoid interrupting the outputs. This is further explained in the Code Generation section.

Code Implementation
For NFA diagram particularly in Figure 4 is coded using dictionary of dictionaries in python with the following format:

```
nfa = {
        state number : {
                symbol : [state number/s to transition to]
        }
}
```

For DFAs, it is coded using a list of lists where row index represents the state, column index represents the symbol read and the value stored is the state to transition to upon reading the symbol.

```
table = [
            //symbol1                 symbol2                 symbol3
        [state to transition to, state to transition to, state to transition to], // state 0
        [state to transition to, state to transition to, state to transition to] // state 1

        ]
```

Table 1. A Lexer table showing all possible tokens

LEXEME	TOKEN
clout	CLOUT
sigma	SIGMA
x	IDENTIFIER
=	EQUAL
5	CLOUT_LITERAL
“hi”	SIGMA_LITERAL
,	COMMA
+	PLUS
-	MINUS
*	MULTIPLY
/	DIVIDE

;	SEMICOLON
yap	YAP
<	LESS_THAN
slay	SLAY
let	LET
him	HIM
cook	COOK
(	OPEN_PARENTHESIS
!	NOT
)	CLOSE_PARENTHESIS
{	OPEN_CURLY_BRACES
spill	SPILL
>	GREATER_THAN
}	CLOSE_CURLY_BRACES
what	WHAT
if	IF
cooked	COOKED
it’s	IT’S
giving	GIVING

Table 1 defines the mapping between some lexemes, which are the specific strings of characters, and their corresponding tokens, which then will be processed by the parser. For example, some of the reserved words like clout, yap, slay, let, cook, and so on are directly mapped to their respective tokens; say, for instance, CLOUT, YAP, SLAY, LET, COOK indicate that they are predefined for certain functionality in the BrainRot language. Similarly, x, y, str would be classified as an identifier and could be used in storing values or in expressions. Number literals like 5 and 3, and string literals "hi" and "HELLO" were generalized to type-specific tokens called CLOUT_LITERAL and SIGMA_LITERAL, respectively. Arithmetic operators (+, -, *, /), comparison operators (<, >, =), logical operators (!), and delimiters (;, ',', {, }, (,)) are also tokenized for clear syntactic roles. Unique keywords such as 'what', 'if', 'spill', 'cooked', etc., with their tokens- WHAT, IF, SPILL, COOKED- identify special syntax in BrainRot. This lexer table is created by the lexical analyzer or scanner in which a lexeme and token is passed to the recursive descent parser one-by-one for syntactic analysis. This lexer table is kept for debugging purposes.

Grammar

The following grammar written in Backus–Naur form is what the compiler is currently capable of.

```
<Program> ::= <Statement> <Program> | 'SEMICOLON' <Program> | ε
<Statement> ::= <Declaration> | <Assignment> | <if> | <Print> | <Scan> |
"IT'S" "GIVING" | "SEMICOLON"
<Declaration> ::= <Data_type> <Variable_list> 'SEMICOLON'
<Data_type> ::= 'CLOUT' | 'SIGMA'
<Variable_list> ::= <Variable> <Variable_list_prime>
<Variable> ::= 'IDENTIFIER' <Variable_prime>
<Variable_list_prime> ::= 'COMMA' <Variable_list> | ε
<Variable_prime> ::= 'EQUAL' <Expression> | ε
<Expression> ::= <Term><Expression_prime>
<Expression_prime> ::= <Operator> <Term> | ε
<Term> ::= 'IDENTIFIER' | <Literal>
<Literal> ::= 'MINUS' 'CLOUT_LITERAL' | 'CLOUT_LITERAL' | 'STRING_LITERAL' |
'SLAY'
<Operator> ::= 'PLUS' | 'MINUS' | 'MULTIPLY' | 'DIVIDE'
<Assignment> ::= 'IDENTIFIER' 'EQUAL' <Expression> 'SEMICOLON'
<If> ::= 'LET' 'HIM' 'COOK' <Condition> <Block> <Else_if> <Else>
<Else_if> ::= 'WHAT' 'IF' <condition> <block> <Else_if> | ε
<Else> ::= 'COOKED' <block> | ε
<Condition> ::= 'OPEN_PARENTHESIS' <Expression> (<Relational_operator>
<Expression> | ε) 'CLOSE_PARENTHESIS'
<Relational_operator> ::= 'EQUAL' 'EQUAL' | <Less> |'NOT' 'EQUAL' | <Great>
<Less> ::= 'LESS_THAN' <Relational_prime>
<Great> ::= 'GREATER_THAN' <Relational_prime>
<Relational_prime> ::= 'EQUAL' | ε
<Block> ::= 'OPEN_CURLY_BRACE' <Block_program> 'CLOSE_CURLY_BRACE' |
<Statement>

<Block_program> ::= <Block_statement> <Block_program> | 'SEMICOLON'
<Block_program> | ε
<Block_statement> ::= <Declaration> | <Assignment> | <Print> | <Scan> | "IT'S"
"GIVING" | "SEMICOLON"

<Print> ::= 'YAP' 'LESS_THAN' 'LESS_THAN' <Expression> <Print_prime>
'SEMICOLON'
<Print_prime> ::= 'LESS_THAN' 'LESS_THAN' <Expression> <Print_prime>
<Scan> ::= 'SPILL' 'GREATER_THAN' 'GREATER_THAN' 'IDENTIFIER' <Scan_prime>
'SEMICOLON'
<Scan_prime> ::= 'GREATER_THAN' 'GREATER_THAN' 'IDENTIFIER' | ε
```

Variable declarations allow chaining, enabling multiple variables to be declared and optionally initialized in a single statement. Similarly, yap (print) and spill (scan) statements support stream chaining. Expressions are limited to two operands with one operation. The <Block> non-terminal accommodates standalone statements or full blocks enclosed in curly braces. If statements in BrainRot only evaluate integer-based conditions, does not support string declarations if-else scope, and do not yet support nested if statements, thus, **`<Block_program>`** and **`<Block_statement>`** are used instead of **`<Program>`** and **`<Statement>`** in **`<Block>`**.

Parser

The type of parser used is the recursive descent parser. It calls the scanner for lexeme and tokens, and based on the grammar, it checks if it matches the expected tokens, checks for semantics, and can call or translate directly to code generators, which are referred to as “translators” in the code. The non-terminals in the grammar are represented as individual functions in the code and the production rules are identified inside the function. Below is the implementation of **`<Data_type>`**.

```
def parseDataType(self):
  # <Data_type> ::= 'CLOUT' | 'SIGMA'
  if self.currentToken == 'CLOUT':
    self.match('CLOUT')
    return 'CLOUT'
  elif self.currentToken == 'SIGMA':
    self.match('SIGMA')
    return 'SIGMA'
  else:
    self.debugMissingDataType()
```

The parser creates a symbol table shown in Table 2.

Table 2. BraintRot Symbol table

VARIABLE (key)	SCOPE (key)	DATATYPE	STACK
identifier	int	CLOUT | SIGMA	int

Parser inserts a new symbol in the symbol table whenever there is a variable declaration. If the variable is outside any conditional statements, its scope and stack are 0 and -1 respectively. The scope in the symbol table is incremented when a conditional statement is encountered and the stack is incremented whenever a variable is declared inside a conditional statement. The scope is back to zero when a variable is declared after conditional declarations and the stack is back to -1. Consider this code showing variables at varying scopes.

```
let him cook (1) clout x, y, z;
what if(1) clout x, y, z;
cooked clout x, y, z;

clout x, y, z;

let him cook (1) clout x, y, z;
what if(1) clout x, y, z;
cooked clout x, y, z;
```

And the symbol table generated is this:

VARIABLE (key)	SCOPE (key)	DATATYPE	STACK
x	1	CLOUT	0
y	1	CLOUT	1
z	1	CLOUT	2
x	2	CLOUT	0
y	2	CLOUT	1
z	2	CLOUT	2
x	3	CLOUT	0
y	3	CLOUT	1
z	3	CLOUT	2
x	0	CLOUT	-1
y	0	CLOUT	-1
z	0	CLOUT	-1
x	4	CLOUT	0
y	4	CLOUT	1
z	4	CLOUT	2
x	5	CLOUT	0
y	5	CLOUT	1
z	5	CLOUT	2
x	6	CLOUT	0
y	6	CLOUT	1
z	6	CLOUT	2

Semantic Analyzer

Currently, the semantic analyzers are the following:

1. Type checking
2. Division by zero (obvious and dynamic)
3. Undeclared variable (scoping included)
4. Invalid operations (e.g. doing operations aside from addition to strings)

The semantic analyzers play a major role in ensuring that the program follows all rules of its respective programming language on issues related to the meaning, rather than syntax. The semantic analysis in Compiley Studio involves undeclared variable checks to ensure a variable is declared upon usage. Checking for type mismatch prevents operations from performing with different data types, which may cause unforeseen behavior in a program, like attempting to add a string to a number. Dynamic division by zero detection is significant because it determines whether some division operation may yield undefined behavior that could cause program crashes during execution. Finally, analyzers also detect operations on invalid operands, such as subtraction, multiplication, or division of strings. Only addition (concatenation) is the available operation for the string data type. These checks enhance code robustness and help developers identify logical issues early in the development process.

The only semantic analyzer separated through a function is checking if two lexemes are compatible for operation by using the symbol table, and the rest are handled directly in parts of the code where a simple check is done or a call to another function to retrieve some values not particularly created only for semantic checking.

```
def isSameType(self, varName, firstTermLexeme, firstTermToken)
```

Debugger

The preprocessor, scanner, semantic analyzer, and code generation call the debugger when something wrong is detected. The pipeline in Figure 1 shows that the code is processed by each part, ensuring that the syntax and the semantics, as well as generating the MIPS code, are error-free and completed before proceeding to the next parts of the code. The debuggers of Compiley Studio are the following:

1. **Unexpected Character and Use of the Disallowed Symbol §**

```
def debugger(self, errorNumber, index=None)
```

2. **Expected Data Type**

```
def debugMissingDataType(self)
```

3. **Redeclaration of Variable**

```
def debugVariableRedeclaration(self, varName)
```

4. **Undeclared Variable**

```
def debugUndeclaredVariable(self, varName)
```

5. **Invalid Value Assigned**

```
 def debugInvalidValue(self):
```

6. **Operator Expected**

```
def debugNoOperator(self):
```

7. **Invalid String Operation**

```
def debugInvalidStringOperation(self, operator):
```

8. **Division by Zero**

```
def debugDivisionbyZero(self):
```

9. **Invalid Print Statement**

```
def debugInvalidPrint(self, token):
```

10. **Type Mismatch**

```
def debugTypeMismatch(self, varName1, datatype1, varName2, datatype2):
```

11. **Unexpected Keyword**

```
def debugUnexpectedKeyword(self, lexeme)
```

Other debuggers that are displayed in some other way such as in exception handling (not separated to a function) are the following:

1. Error raised by MIPS
2. No internet connection (for code with AI)
3. Invalid prompt
4. Unknown error

Code generation

This section will only cover the techniques and methods used to convert BrainRot code to MIPS code. This section assumes the basic understanding on how mips syntax works as well as common syscalls operations. If you are a complete beginner of MIPS, you can watch this playlist:
https://www.youtube.com/watch?v=u5Foo6mmW0I&list=PL5b07qlmA3P6zUdDf-o97ddfpvPFuNa5A

Scoping

Compiley Studio translates BrainRot to MIPS line by line into its corresponding MIPS assembly, which involves consideration of the registers usage and memory access methods supported by the MIPS architecture. For global declarations, memory locations (.data) are used while variables declared inside the conditional statements are stored in the stack pointer ($sp).

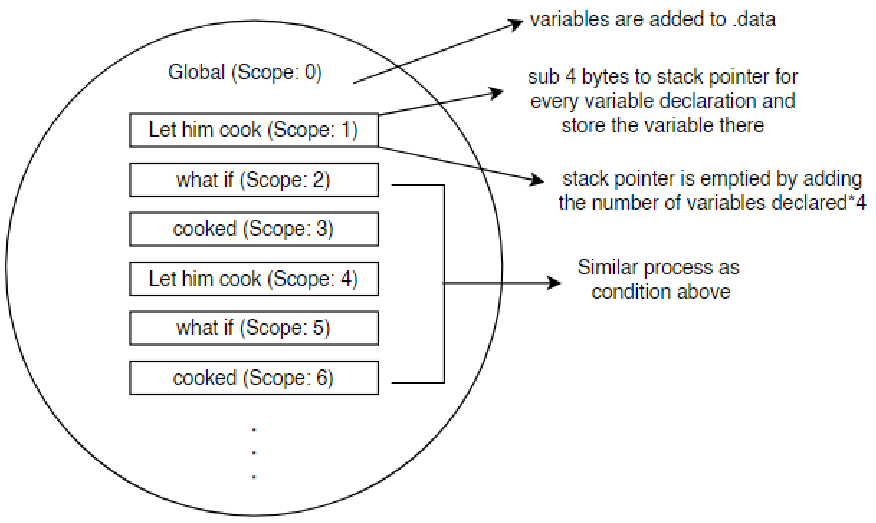

Global (Scope: 0)
Let him cook (Scope: 1)
what if (Scope: 2)
cooked (Scope: 3)
Let him cook (Scope: 4)
what if (Scope: 5)
cooked (Scope: 6)
variables are added to .data
sub 4 bytes to stack pointer for every variable declaration and store the variable there
stack pointer is emptied by adding the number of variables declared*4
Similar process as condition above

Figure 9. Diagram for Different Variable Declaration Storage

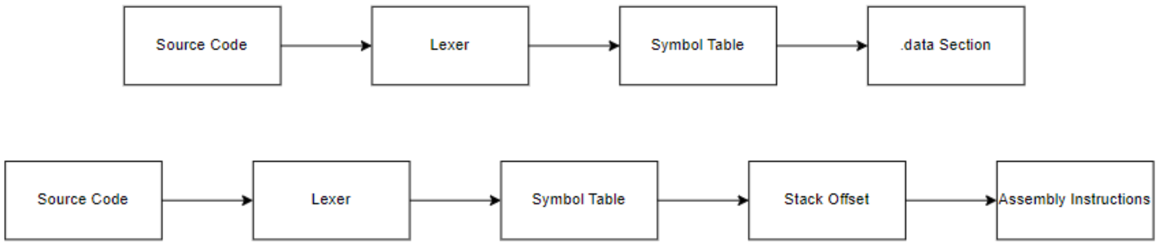


Figure 10. Transition Diagram for Storing Global and Local Variables

As we can see from Figures 9 and 10, global variables and static data are identified by the lexer and added to the symbol table. These entries are then translated into the .data section of the MIPS code. Variables within conditional scopes are assigned offsets in the stack. During code generation, values are pushed to or popped from these offsets based on the program flow.

The reason why stack pointers are used to store variables is because using .data can restrict having the same variable name declared on different scopes and variables declared inside the conditional statement will be accessible anywhere which should not be the case.

In knowing if a variable is within the reachable scopes, we use the symbol table to locate if the variable exists in the current scope, and if not, check the global scope, and if there really isn't then that variable is undeclared. Since there is no nesting, checks of scopes are only done to the current scope the code is in and the global scope.

Storing Strings

Storing strings is different because you have to consider the size. Unlike integers, which can easily be handled by the .data section, storing strings right away in the .data section by .asciiz followed by the string is not modifiable because they are treated as read-only memory in many MIPS environments and assemblers (e.g. MARS). To solve this problem, we instead use .space 1024, where we allocate sufficient space for a string and just store each of the characters one-by-one.

In the case of strings that have no variable names such as those found in printing string literals or performing operations to string literals, we implemented the concept of “set break” (sbrk in mips where we allocate additional memory dynamically during program execution, this effectively handles string literals of varying lengths. This is how sbrk is implemented.

```
li $v0, 9
li $a0, 1024
syscall         # $v0 now contains the new heap break address after allocating 1024 bytes
```

Retrieving and Updating Variable Values

In retrieving values of a certain variable, we have to use the stack column of the symbol table in order for us to know whether we can directly use the variable name or if we have to calculate the stack pointer offset.

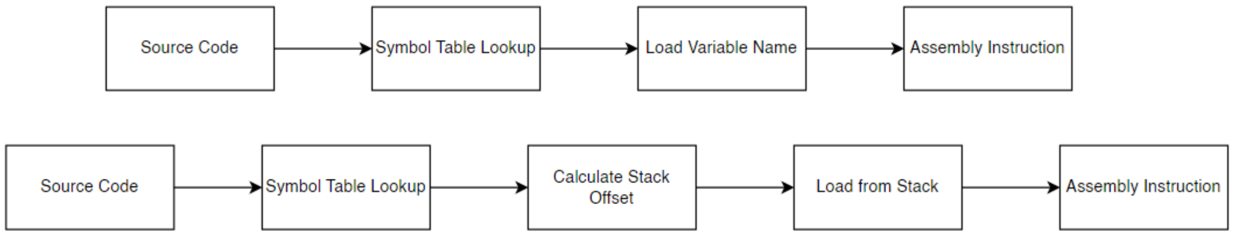


Figure 11. Transition Diagram for Retrieving and Updating Variables from Different Scopes

From Figure 11, if a variable is a global variable, as checked in the symbol table, then its memory addresses can be accessed to retrieve its value. On the other hand, offsets relative to the $sp register are used to retrieve value from the stack. The formula used to calculate the offset is (numberOfStackItems - variableStackValue)*4.

If Else

If ‘Let’ ‘Him’ ‘Cook’ are successfully matched, then the condition will be parsed. The left side of the relational operator will be stored in $t6, the relational operator will be converted to its MIPS equivalent and the right side will be stored in $t7, the label is self.block (what block and scope are we in). For instances that one or both sides is an expression, then we will call the parseExpression function. The resulting answer will be stored in $t6 or in $t7, depending on which side of the operator it is located. The final MIPS conversion of the condition will be stored first in the string called ifConditionMips. The statement/s inside the if block will then be converted and stored in a string called ifMips, and after the conversion, j END_IF{self.ifScope} is added. This goes on every time ‘What’ ‘If’ is matched or ‘Cooked’ is encountered. After parsing the whole if-else block, END_IF{self.ifScope}: is added, which indicates the end of the if-else block.

Subroutines

There are 3 available subroutines that can be used in doing operations in BrainRot.

1. **String copy**
   This is called when assigning a string variable to another string variable.
   The arguments passed are the following:
   - **$a0**: Pointer to the source string (string to be copied).
   - **$a1**: Pointer to the destination memory location (where the string is copied).

```
string_copy:
copy_loop:
lb $t0, 0($a0)
sb $t0, 0($a1)
beq $t0, $zero, copy_done
addi $a0, $a0, 1
addi $a1, $a1, 1
j copy_loop

copy_done:
jr $ra
```

2. **String concatenation**
   This is called when performing addition among string variables, slay, and string literals. The arguments passed are the following:
   - **$a0**: Pointer to the first string (source 1).
   - **$a1**: Pointer to the second string (source 2).
   - **$a2**: Pointer to the destination (where the concatenated result is stored).

```
concat_strings:
copy_first:
lb $t0, 0($a0)
beq $t0, $zero, copy_second
sb $t0, 0($a2)
addi $a0, $a0, 1
addi $a2, $a2, 1
j copy_first

copy_second:
lb $t0, 0($a1)

beq $t0, $zero, concat_done
sb $t0, 0($a2)
addi $a1, $a1, 1
addi $a2, $a2, 1
j copy_second

concat_done:
li $t0, 0
sb $t0, 0($a2)
jr $ra
```

3. **Check for Dynamic Division by Zero**
   This subroutine is called everytime that there is an attempt to divide two numbers and the divisor is zero. It prints the flag §2 to indicate that there is a division by zero error, then ends the program.

```
# Division by zero handling flag §
division_by_zero:
li $a0, 167

li $v0, 11
syscall
# return code: 2
li $v0, 1

li $a0, 2
syscall
li $v0, 10
syscall
```

Flags

You may notice that this illegal character "§" is frequently mentioned and return codes are not yet understood, but small details contribute a lot especially when printing characters to the terminal sent from MARS. The problem with the communication of the Python subprocess module to MARS is that it isn't consistent in printing characters. There are times that it skips other characters to be printed and it isn't consistent in telling when the MIPS code asks for input or whether the MIPS code has already finished its execution.

The solution we came up with is by using flags. The flags serve as an indicator on what Compiley Studio terminal should do. The choice of "§" is simply because it is hard to type on a keyboard and it is less likely to be used in coding. The summary of what it does is shown in Table 3.

Table 3. Flag definition

Flag	Meaning	What the Terminal Does (Steps)
[no flag]	Either MARS is sending characters or MARS is executing some other instructions.	Waits and stores the characters sent from MARS to some variable if there is.

§0	The MIPS program is asking for input.	1. Stop asking/receiving characters from MARS. 2. Display the stored characters to the terminal UI. 3. Wait for the user to send input. 4. Process user input and send it to MARS. 5. Resume asking/receiving characters from MARS.
§1	MIPS program is done.	1. Stop asking/receiving characters from MARS. 2. Display the stored characters to the terminal UI.
§2	Division by zero occurred.	1. Stop asking/receiving characters from MARS. 2. Display the stored characters to the terminal UI. 3. Display an error message to terminal UI.

The restriction of the symbol “§” highlights the need that the scanner should raise an error once those are encountered in BrainRot code as they may disrupt the terminal.

MARS

To execute the generated code, the jar file “MARS” is connected with the IDE. MIPS Assembler and Runtime Simulator (MARS) is chosen as it has Command Line Interface (CLI) that allows it to run MIPS programs without the graphical user interface and extract the output and know when MARS asks for input, which is also made possible using the subprocess library in Python.

MARS communicates with the terminal through the use of certain codes that are constructed during runtime to indicate the different stages or events of the program. Upon executing MIPS code compiled to the terminal, there should be specific prints of special indicators: §0, §1, §2. These codes are handled in the read_output function in the terminal implementation to provide contextual feedback.

- END -