Compiler Construction

Project Phase-I

Name: Anas Maqsood
Reg: L1F22BSCS1078
Section: G10

Directed to: Aneela Mehmood

CC-Project — CE++

Overview

CE++ is a simplified version of C++ created to make programming more accessible and intuitive for beginners, underaged learners, and senior citizens. The language replaces traditional keywords with easy-to-understand English words so that code reads more like natural language rather than an abstract syntax. For example, instead of using cryptic terms like `if`, `else`, or `void`, CE++ uses descriptive words such as `Suppose`, `Suppose_else`, and `empty`, which clearly convey their purpose. Operators are also redesigned for clarity by replacing symbols with meaningful words like `add`, `subtract`, `mult`, and `divide`, making arithmetic and logical operations self-explanatory. Punctuations are simplified and standardized to maintain structure while reducing confusion; for instance, `;;` is used as a statement terminator to distinguish it from common single semicolons, while braces `{}` and parentheses `()` are retained for familiar block and grouping syntax. This design emphasizes readability, consistency, and educational simplicity, ensuring that programming feels less intimidating and more approachable for all users.

Regex Table

Token Type	Regular Expression (Regex)	Example
Keyword	Suppose | Suppose_else | Because_of | give_out | Correct | Incorrect | take | show | Digit | Word | STOP | empty | choice | also | Point	Suppose, show, Digit
Identifier	[a-zA-Z_][a-zA-Z0-9_]*	myProg, reg_no
Integer	[0-9]+	5, 123
Float	[0-9]+.[0-9]+	15.5, 3.14
String	"([^"\]	\[\"tn])*"
Character	'[^']'	'B'
Operator	= | > | < | equal | greater_or_equal | lesser_or_equal | not_equal | AND | OR | % | ! | ++ | -- | add | subtract | mult | divide	x < 5, A add B, %
Punctuation	//|// | { | } | , | ;; | ::	( ), { }, ;;

Lexical Elements

Keywords

Keyword	Meaning	Example (from your code)
Suppose	if (conditional)	Suppose(num > 3) { … }
Suppose_else	else (alternative branch)	Suppose_else { … }
Because_of	for (loop)	Because_of(Digit x = 0; x < 5; x = x + 1) { … }
give_out	return	give_out 0;;
Correct	boolean true	Correct
Incorrect	boolean false	Incorrect
take	input	take(reg_no);;
show	output/print	show("A + B = %d", A add B);;
Digit	int (data type)	Digit A = 4, B = 2;;
Word	char (data type)	Word character = 'B';;
STOP	break	STOP;;
empty	void (data type / return type)	empty myProg()
choice	logical OR (keyword form)	choice
also	logical AND (keyword form)	also
Point	float (data type)	Point float_val = 15.5;;

Operators

Symbol / Word	Operator Name	Example (from your code)
=	assignment	Digit A = 4;;
>	greater-than (relational)	num > 3
<	less-than (relational)	x < 5
add	addition	A add B
subtract	subtraction	A subtract B
mult	multiplication	A mult B
divide	division	A divide B
%	modulo	%
!	logical negation	!
equal	equality test	equal
greater_or_equal	≥ relational	greater_or_equal
lesser_or_equal	≤ relational	lesser_or_equal
not_equal	!= relational	not_equal

AND	logical AND (operator form)	AND
OR	logical OR (operator form)	OR
++	increment	++
--	decrement	--

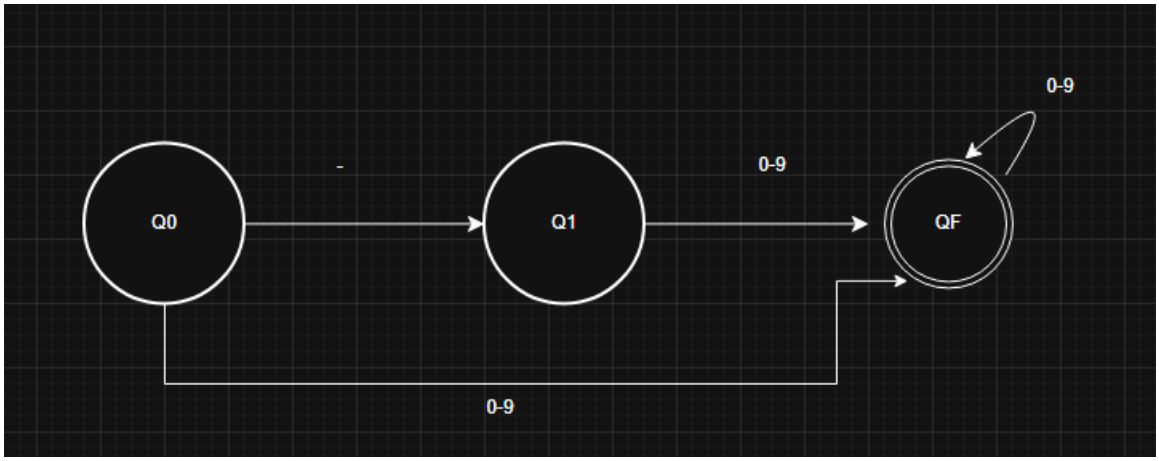
Punctuations / Delimiters

Symbol	Role	Example (from your code)
(	left parenthesis (grouping / call)	show("…")
)	right parenthesis (grouping / call)	show("…")
{	left brace (block start)	{
}	right brace (block end)	}
;;	statement terminator	show(x);;
,	separator in lists/arguments	Digit A = 4, B = 2;;
::	colon token (defined, not used in sample)	::

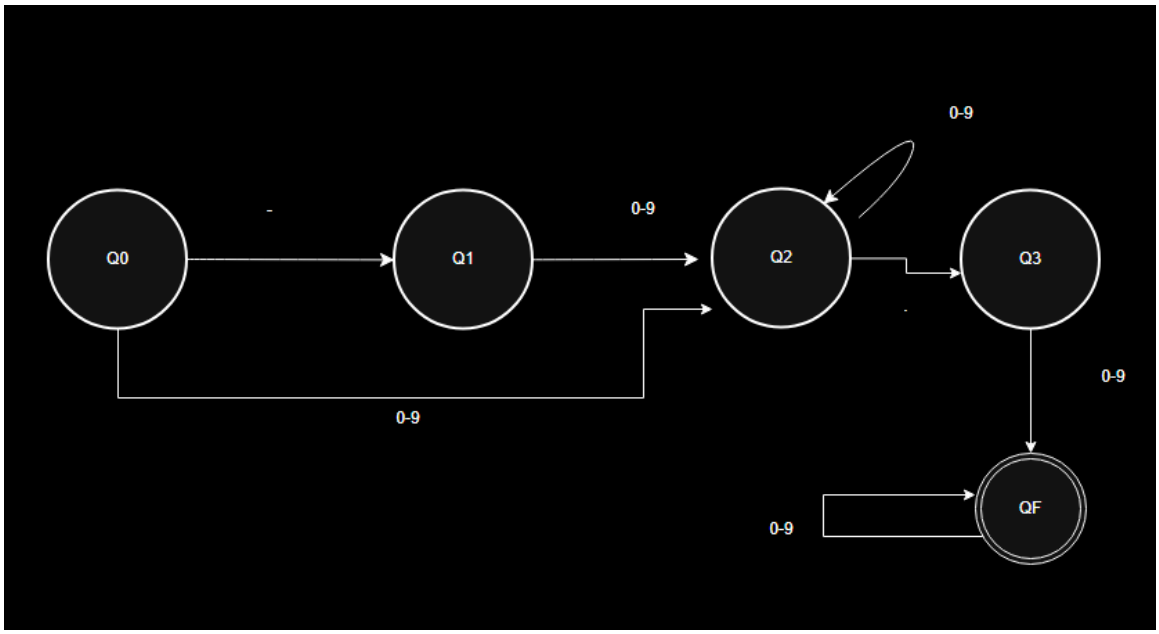
Finite Automata

DataTypes

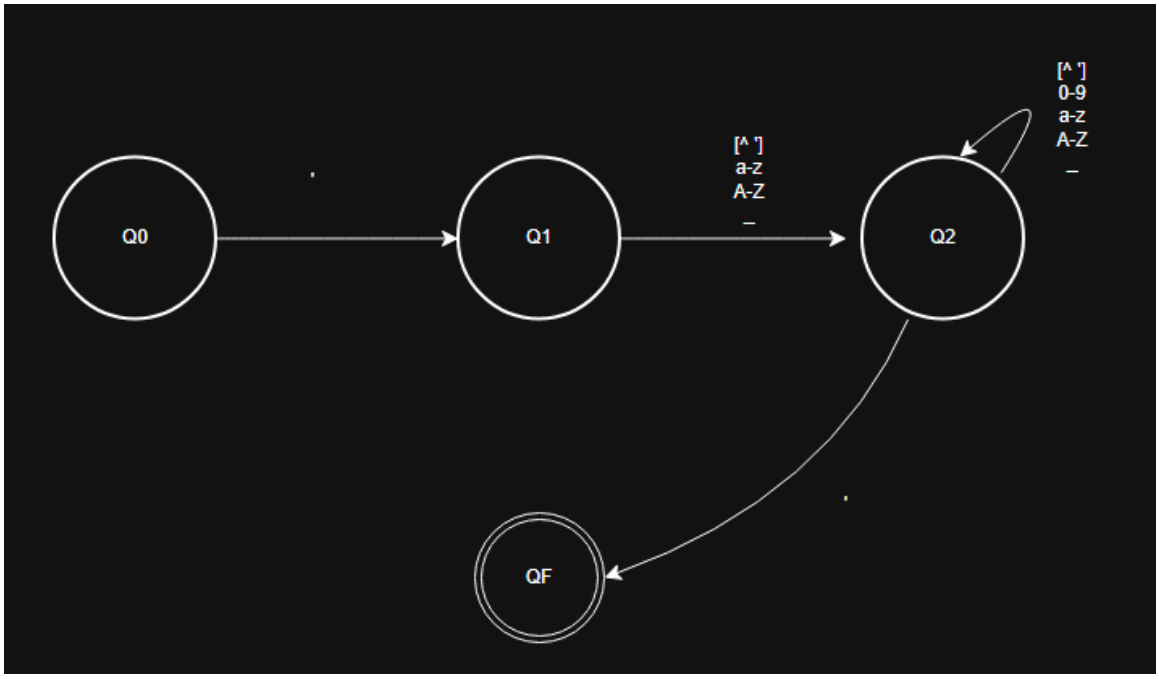
Integers:



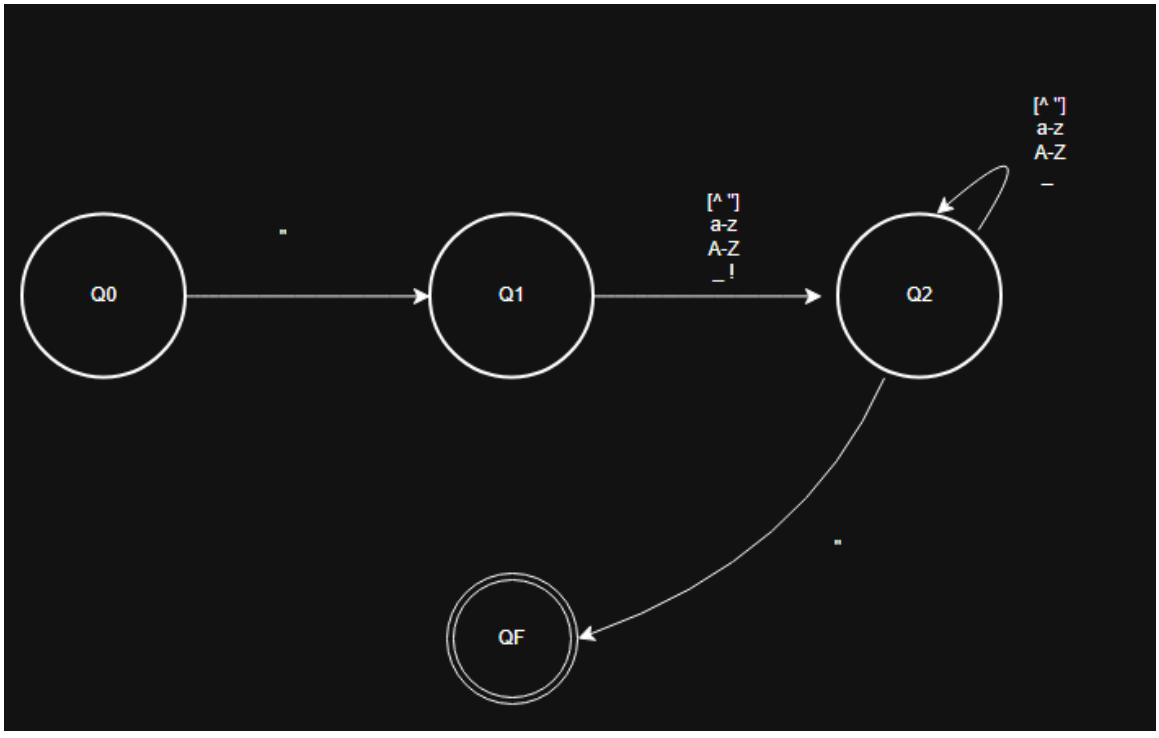
Float:



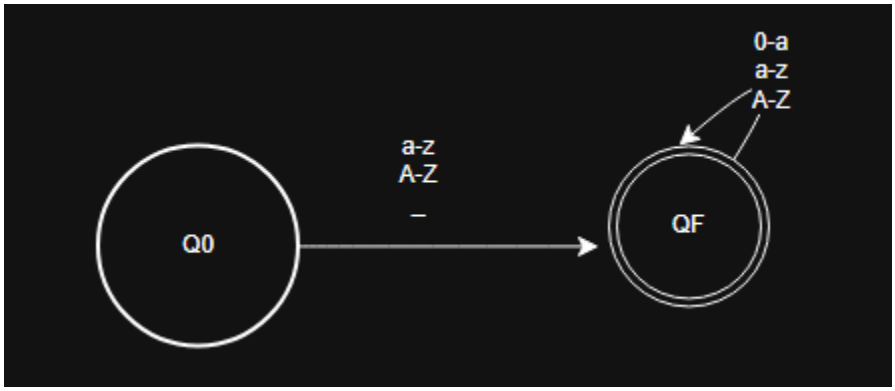
Char:



String:

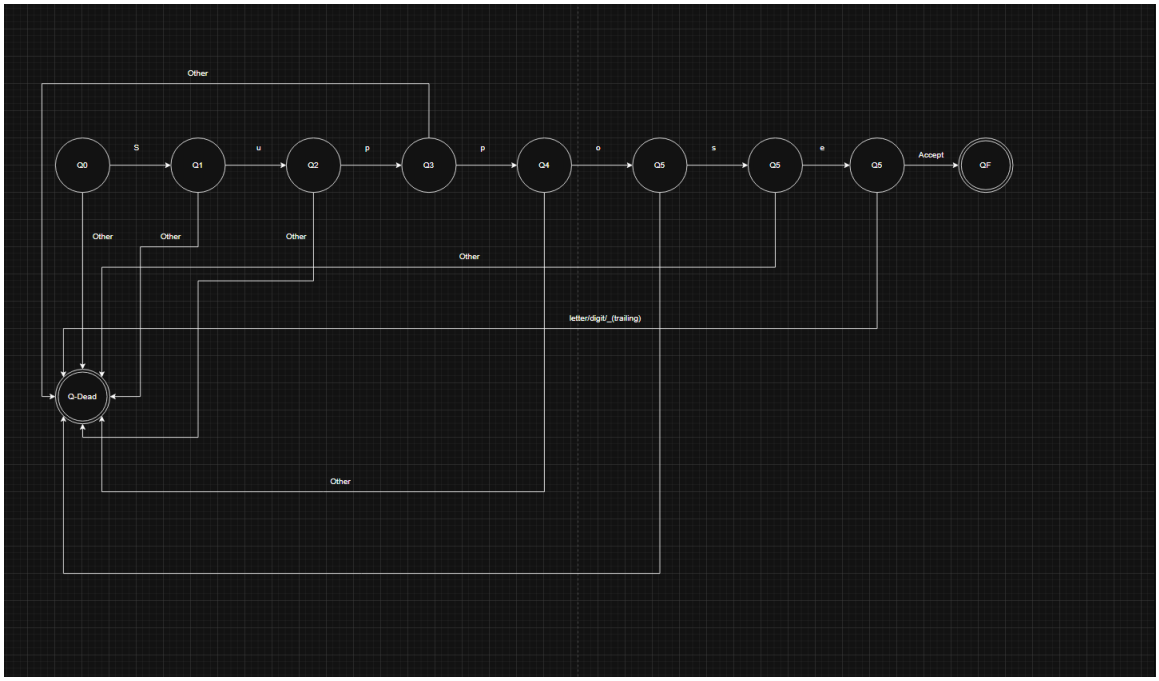


Identifier:

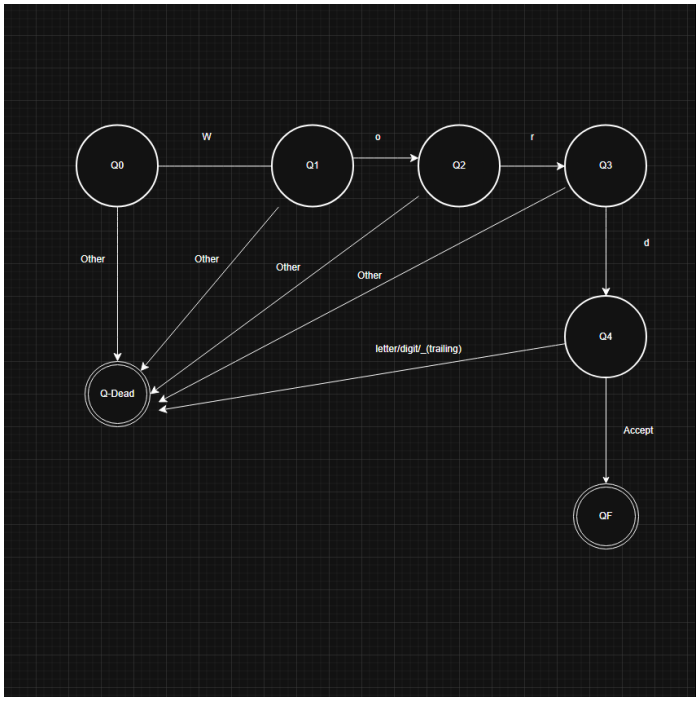


Custom Keywords

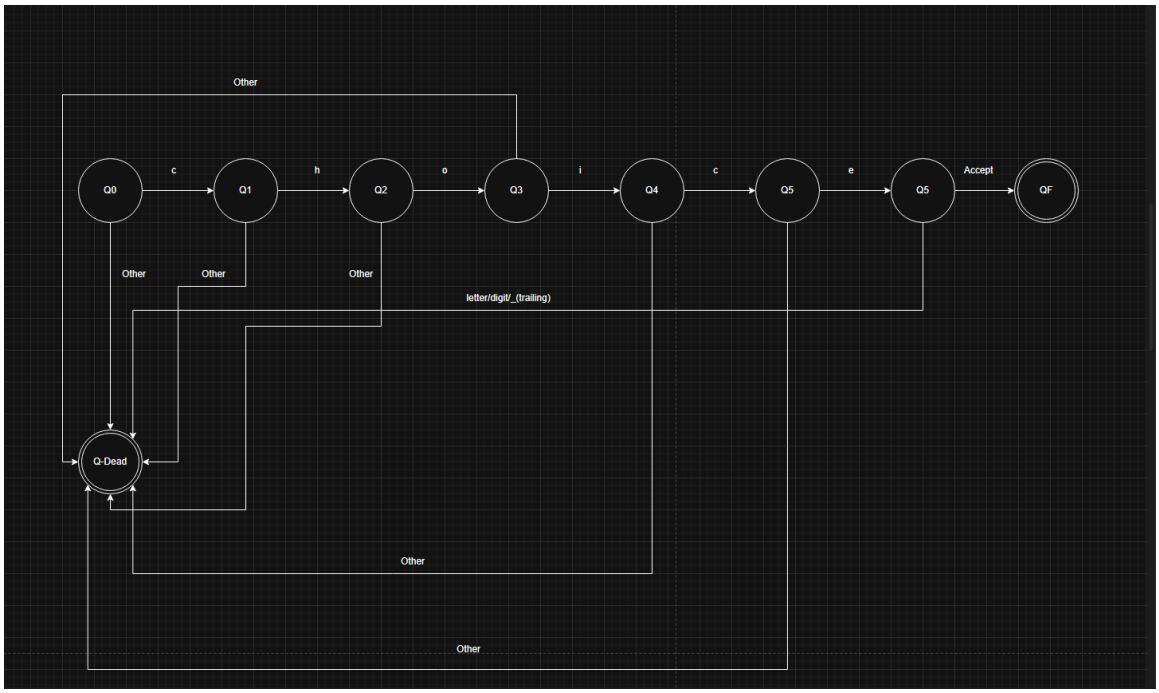
Suppose:



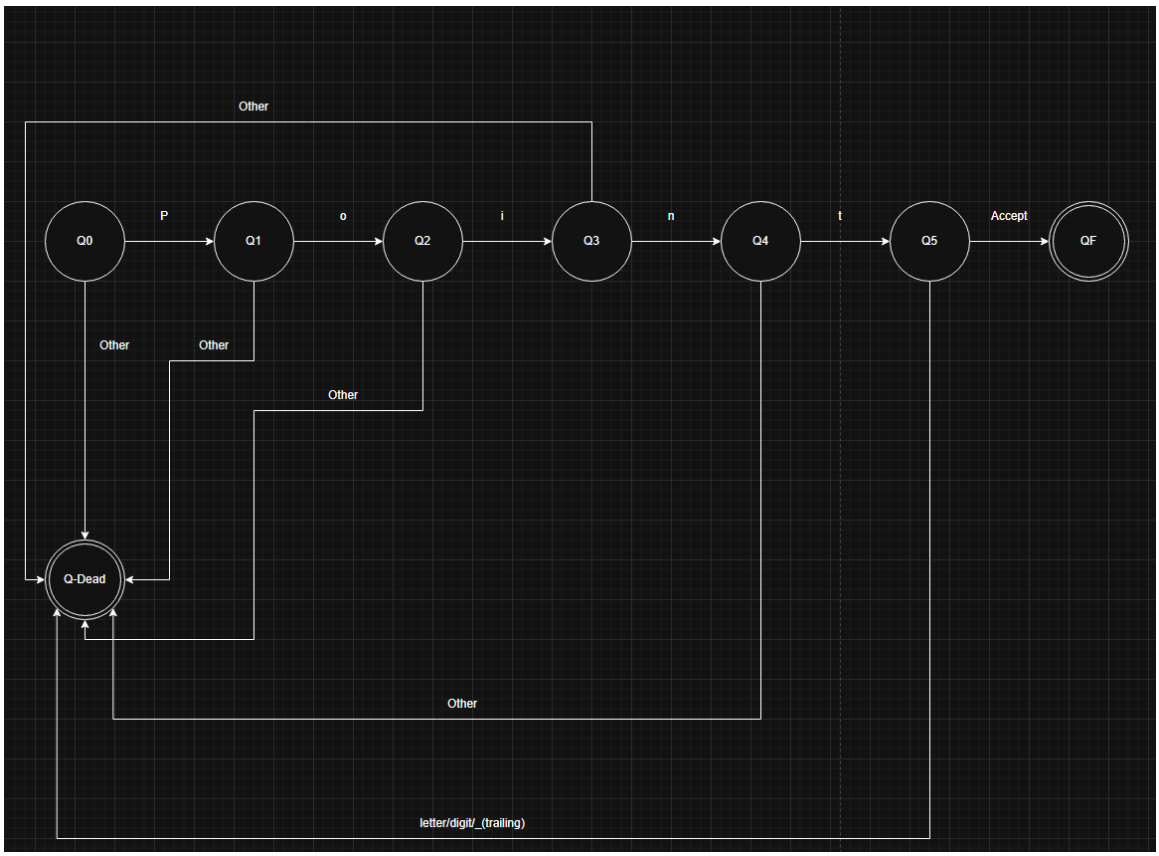
Word:



choice:



Point:



Brief Explanation

Keywords

- **Suppose**: Chosen as a natural-language alternative to *if*. Makes conditional logic more readable and intuitive for beginners.
- **Suppose_else**: Complements *Suppose* for branching. Explicit naming avoids confusion with other constructs.
- **Because_of**: Represents *for* loops. The phrase suggests "because of this condition, repeat," making iteration logic descriptive.
- **give_out**: Designed to replace *return*. The wording clearly conveys "give out a value" at the end of a function.
- **take**: For input. Simple and action-oriented, easy for learners to remember.
- **show**: For output. Chosen for clarity,"show something on screen."
- **Digit**: Represents integer type. The name directly relates to numeric digits, making its purpose obvious.
- **Point**: Represents floating-point type. Short and intuitive for decimals.
- **Word**: Represents character type. Suggests a textual element, making it beginner-friendly.
- **STOP**: Break statement. Strong, clear command to halt loop execution.
- **empty**: Represents *void*. Indicates "nothing returned," aligning with its semantic meaning.
- **Correct**: Boolean true. Chosen for clarity and natural language feel.
- **Incorrect**: Boolean false. Complements *Correct* for logical clarity.
- **choice:** Logical OR keyword. Suggests "choose one option," making OR semantics intuitive.
- **also**: Logical AND keyword. Suggests "include this too," aligning with AND logic.

Operators

- **=** : Assignment, fundamental for variable initialization.
- **add**: Addition, word-based for clarity and readability.
- **subtract**: Subtraction, descriptive alternative to -.
- **mult:** Multiplication, short and clear.
- **divide**: Division, intuitive word form.
- **>** : Greater-than, standard comparison operator.
- **<** : Less-than, standard comparison operator.
- **equal**: Equality check, word-based for readability.
- **not_equal:** Inequality check, descriptive alternative to !=.
- **!** : Logical negation, standard symbol for NOT.
- **AND** : Logical AND, uppercase for emphasis.
- **OR** – Logical OR, uppercase for emphasis.

- **++** : Increment, standard shorthand for loops.
- **--** : Decrement, standard shorthand for loops.
- **%** : Modulo, essential for remainder operations.

Punctuations

- **(** and **)** : Grouping and function calls, standard in most languages.
- **{** and **}** : Block delimiters, clear structure for code blocks.
- **;;** : Statement terminator, double semicolon chosen for uniqueness.
- **,** : Separator for multiple declarations or arguments.
- **::** : Reserved for special syntax, future extensibility.

Why this design?

- **Natural Language Readability**: Keywords like *Suppose, Because_of, give_out* make code self-explanatory for beginners. They reduce the cognitive load compared to cryptic symbols.
- **Consistency & Simplicity**: All keywords are case-sensitive and descriptive. Operators and punctuations follow predictable patterns.
- **Educational Focus**: The language is designed for clarity, so learners can understand logic without memorizing abstract symbols.
- **Uniqueness**: Using ;; as a statement terminator and word-based operators (*add, subtract*) differentiates this language from C-like syntax, making it distinct yet approachable.
- **Error Prevention**: Clear keywords reduce ambiguity and make parsing easier. For example, STOP is unambiguous compared to *break*.