Republic of the Philippines
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
College of Computer and Information Sciences
Sta.Mesa, Manila

E.C.H.O: Executable Code, Human Output

A Programming Language Proposal

In partial fulfillment for the Course:

COSC 303 – Principles of Programming Languages

Group 5:

Baptista, Nicko Adrian

Delos Reyes, Ariane Joy

Martinez, Bouie

Madelo, Mark Anthony

Bermudez, Mark Daniel

Jocson, Dan Louie

2025-2026

Table of Contents

I. Introduction

ECHO (Executable Code, Human Output) is a high-level imperative procedural language designed to make coding accessible, efficient, and logical. With a special emphasis on data analysis and computational thinking, ECHO provides a programming experience that reduces the necessity for memorizing complicated syntax for core programming ideas. Yang (2018) emphasizes the importance of readable syntax in helping programmers concentrate more on solving problems and less on mastering coding mechanics, particularly when languages are designed to be more approachable. By simplifying the coding process, ECHO enables users to perform quick, readable execution of analytical processes, without the burden of complex syntax. And as such, the name "ECHO" reflects its purpose, just as an echo repeats what it hears, ECHO mirrors the user's logical thinking in code, embodying its definition: *Executable Code, Human Output.*

ECHO draws inspiration from languages such as C and Java but creates a much more user-friendly environment by getting rid of heavy punctuation and case sensitivity. Instead, clear and readable commands like "start", "echo", and "end" are used. In emphasizing data, ECHO includes functions in the library such as mode(), median(), and mean() that facilitate experimentation and visualization with a minimum of importing libraries and common code. Complementating these features are logical utilities such as isEven() and isOdd(), alongside a String Insertion System (SIS) that enables direct variable references within output lines using the "@" symbol, which in turn produces a more natural or readable output.

In conclusion, ECHO translates human thought processes into executable computer logic. Studies in computer education have demonstrated how this can be achieved, increasing both engagement and understanding, through more simplified and interactive programming environments (Bau et al., 2017). Focusing on readability, structure, and conceptual clarity, ECHO facilitates users in converting their ideas into logical and executable programs. With this, users are most probably able to solve their data-driven problems much simply.

II. Data Analysis and Processing Principles

1. Schema-Based Function Binding

a. This feature enables developers to define structured records with named fields that automatically enforce validation, transformations, and business logic. By binding functions directly to individual fields, data integrity is maintained, invalid values are rejected, and computations or transformations occur seamlessly.

b. Example:

```
data_struct CustomerRecord {
  Name: string {capitalizeName},
  Age: integer {validateAge},
  ZipCode: string {validateZipCode}
}

function validateAge(value)
  if value >= 18 then
    return true
  else
    error("Age must be 18 or older.")
    return false
  end if
end function

function capitalizeName(value)
  return capitalizeEachWord(value)
end function

function validateZipCode(value)
  if length(value) == 5 then
    return true
  else
    error("ZipCode must be 5 digits.")
    return false
  end if
end function
```

2. sum()

a. Return the sum of all numerical elements in an array.

b. Example:

```
start
    list scores = [85, 90, 78, 92, 88]
    number total = sum(scores)
    echo "Total score: @total"
end
```

Output:

```
Total score: 433
```

3. **median()**

    a. Calculates and returns the middle value of a sorted numerical array.

    b. Example:

```
start
    list ages = [25, 30, 22, 35, 28]
    decimal middleAge = median(ages)
    echo "Median age: @middleAge"
end
```

Output:

```
Median age: 28
```

4. **mode()**

    a. Calculates and returns an array of the most frequently occurring value/s.

    b. Example:

```
start
    list grades = [85, 90, 85, 78, 85, 88]
    list mostCommon = mode(grades)
    echo "Most common grade: @mostCommon[0]"
end
```

Output:

```
Most common grade: 85
```

5. **average()**

    a. Calculates and returns the arithmetic mean of a numerical array.

b. Example:

```
start
    list temperatures = [72.5, 68.3, 75.8, 70.2, 73.9]
    decimal avgTemp = average(temperatures)
    echo "Average temperature: @avgTemp°F"
end
```

Output:

```
Average exam score: 87.0
```

III. Code Simplification Principles

1. Human-Readable Block Delimiters

a. Use explicit keywords like start, echo, end instead of braces or punctuation for blocks.

b. Example:

```
start
  echo "Hello, world!"
end
```

2. Echo Command for Immediate Output

a. Special statement echo that prints expressions instantly (like print but simpler and integrated).

b. Examples:

```
echo "Data loaded"
```

3. String Insertion System (SIS) with @ Symbol

a. Inline variable references inside string literals using the "@" symbol.

b. Example:

```
echo "The result is @result"
```

4. **Logical Utility Predicates**

    a. Include functions like isEven(), isOdd() as built-in predicates for common logic checks.

    b. Example:

```
start
  var x = 10
  if isEven(x)
    echo "x is even"
  end if
end
```

5. **Case-Insensitive Keywords and Identifiers**

    a. Language ignores case differences for keywords and variable names to reduce syntax errors.

    b. Example:

```
START
  EcHo "Test"
END
```

6. **Whitespace Delimited Statements with Flexible Line Endings**

    a. Statements separated by line breaks and indentation instead of semicolons or braces.

    b. Example:

```
start
  x = 5
  echo x
end
```

IV. Syntactic Elements of the Language

1. Character Set

The encoding system that allows computers to recognize characters from A-Z, numbers, punctuations, and whitespace.

- **CHARACTER_SET =** {LETTER, NUMBER,SPECIAL_CHARACTER, WHITESPACE}

a. Letter Set

It consists of alphabetic letters, including both the upper and lowercase letters that are valid and recognizable in the language's syntax.

- **LETTER =** {UPPERCASE_LETTER, LOWERCASE_LETTER}

1. **UPPERCASE_LETTER** = {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z}
2. **LOWERCASE_LETTER** = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

b. Number Set

Includes all numeric digits (0–9) that are permitted in source code to construct numerical literals, indices, or parts of identifiers.

- **NUMBER** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

c. Special Character and White Space Set

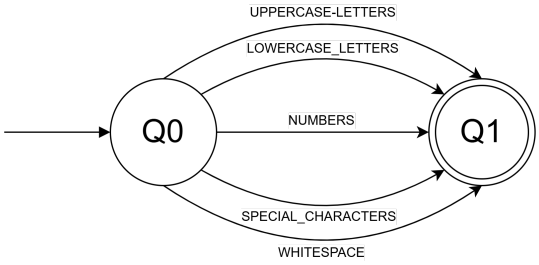
This set comprises non-alphanumeric symbols (such as punctuation and mathematical operators) and whitespace characters (spaces, tabs, newlines).

- **SPECIAL_CHARACTER** = {., ,, +, -, *, /, %, =, <, >, ?, ', ", (,), !, ^, &, |, [,], @, _, \}
- **WHITESPACE** = {<space>, <tab>, <newline>}

Special Character Purpose Map	
SPECIAL_CHARACTER	**Purpose**
.	decimal point in numbers, member access (data.length)
,	separator in lists
+, -, *, /, %, ^	arithmetic operators
=	assignment operator
<, >	comparison operators
"	string delimiter
(,)	function calls, expression grouping
!, &, |	logical operators
[,]	list/array delimiters
@	String Insertion System marker
_	identifier connector (student_age)
\	escape character in character and strings

Whitespace Purpose Map	
WHITESPACE	**Purpose**
<space> ( )	Token separator
<tab> (\t)	Indentation
<newline> (\n)	Statement terminator

Machine for Character Set:



2. Identifiers

a. Composition

- Identifiers in ECHO may only use the characters defined in the language's Character Set, specifically letters (A–Z, a–z), digits, and underscores. It cannot contain blank spaces.
- The first character of an identifier **must be a letter** (uppercase or lowercase) or an **underscore** (_). An identifier cannot begin with a digit.
- After the first character, identifiers can contain any combination of letters (uppercase or lowercase), digits (0-9), and underscores.
- Identifiers cannot contain any special symbols or characters other than the underscore.

b. Case Sensitivity

- ECHO is a case-insensitive language, meaning that myVariable, MyVariable, and MYVARIABLE are all treated as the same identifier.

c. Reserved Words Handling

- Keywords or reserved words predefined in ECHO like *start, end, echo, while, if, else, etc.* cannot be used as identifiers.

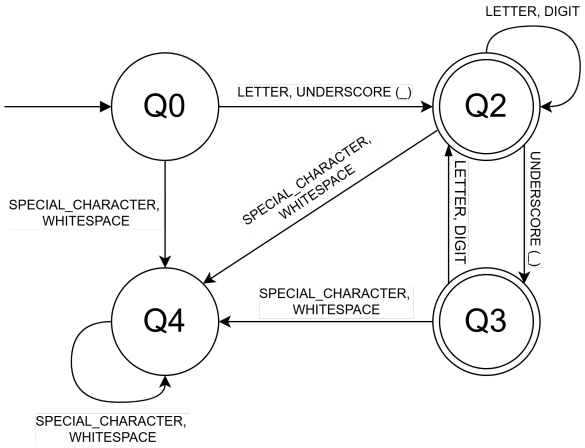
d. Length Constraints

- Identifiers should contain a minimum of 1 character and a maximum of 64 characters.

e. Formal Definition

```
<identifier> ::= <letter> <id_tail> |  _ <id_tail>
<id_tail> ::= ε |  <letter> <id_tail> |  <digit> <id_tail> |  _ <id_tail>
<letter> ::= [A-Z] | [a-z]
<digit>  ::= [0-9]
```

Machine for Identifiers:



3. Operation Symbols

An operator is a symbol that tells the compiler to perform certain mathematical, logical, or relations on one or more operands.

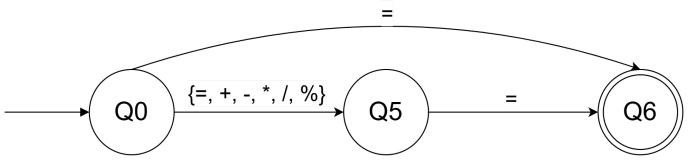
a. Assignment Operator

A symbol used in programming for assigning a value to a variable.

Assignment_Operator	Example	Description
= (Assignment Operator)	x = 1	Assigns the value of variable x to 1.
+= (Addition Assignment Operator)	x += 1	Adds 1 to the current value of x and returns the sum.
-= (Subtraction Assignment Operator)	x -= 1	Subtracts 1 to the current value of x and returns the difference.
*= (Multiplication Assignment Operator)	x *= 1	Multiplies the current value of x by 1 and returns the product.
/= (Division Assignment Operator)	x /= 1	Divides the current value of x by 1 and returns the quotient.

%= (Modulo Assignment Operator)	x %= 1	Divides the current value of x by 1 and returns the remainder.

Machine for Assignment_Operator:



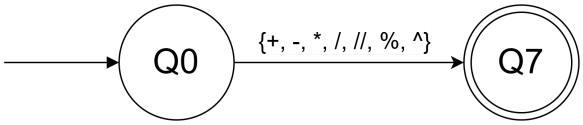
Regular Expression: (= | (\+|-|*|/|%)=)

b. Arithmetic Operator

It is a symbol that indicates the mathematical operation that should be performed on values or variables.

Arithmetic_Operator	Example	Description
+ (Addition Operator)	x + y	Adds the value of variables x and y.
- (Subtraction Operator)	x - y	Subtracts the value of variables x and y.
* (Multiplication Operator)	x * y	Multiplies the value of variable x by variable y.
/ (Division Operator)	x / y	Divides the value of variable x by variable y.
// (Floor Division)	x // y	Divides the value of X by Y and returns the integer result after rounding down to the nearest whole number
% (Modulo Operator)	x % y	Divides the value of variable x by variable y and returns the remainder.
^ (Exponent Operator)	x ^ n	Computes the power of base x to exponent n.

Machine for Arithmetic_Operator:



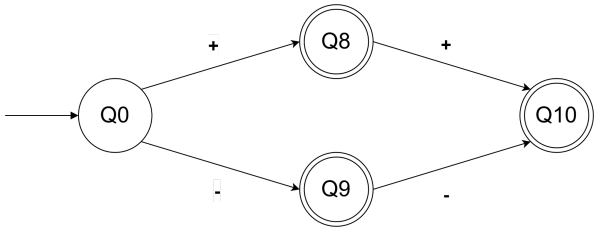
Regular Expression: (\+ | - | * | // | / | % | \^)

c. Unary Operator

Operators that perform on one operand.

Unary_Operator	Example	Description
+ (Unary Plus Operator)	+x	Indicates that the value of variable x is positive.
- (Unary Minus Operator)	-x	Indicates that the value of variable x is negative.
++ (Increment Operator)	++x	Increases the value of operand x by 1 before the expression is evaluated.
	x++	Increases the value of operand x by 1 after the expression is evaluated.
– (Decrement Operator)	--x	Decreases the value of operand x by 1 before the expression is evaluated.
	x--	Decreases the value of operand x by 1 after the expression is evaluated.

Machine for Unary_Operator:



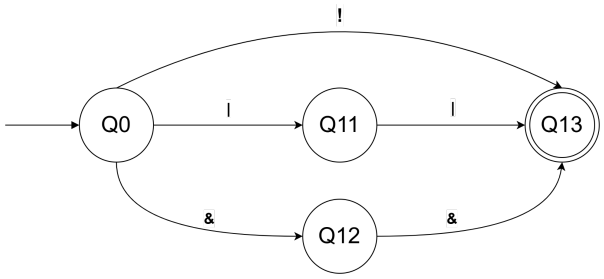
Regular Expression: (+ | - | ++ | --)

d. Logical Operator

A logical operator is one or two symbols or a keyword that tells how to combine conditional statements.

Logical_Operator	Example	Description
! (Logical NOT Operator)	!x	Returns the opposite value of an expression.
|| (Logical OR Operator)	x || y	Returns true if either one of the operands is true. Otherwise, if both operands are false, then it returns false.
&& (Logical AND Operator)	x && y	Returns true (1) only if both operands are true. Otherwise, if either one operand is false, then it returns false.

Machine for Logical_Operator:



Regular Expression: (!|\|\||&&)

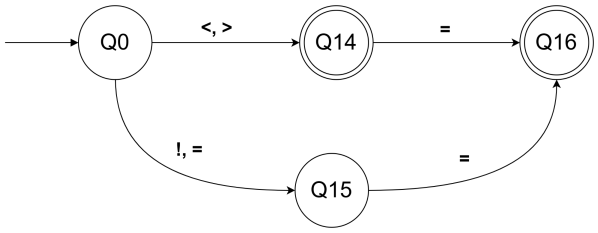
e. Relational Operator

A symbol utilized in comparing two values to obtain their relationship, such as equality, inequality, or ordering.

Relational_Operator	Example	Description
== (Equal To Operator)	x == y	Checks and returns true if the value of both operands are equal. Otherwise, returns false.

!= (Not Equal To Operator)	x != y	Checks and returns true if the value of both operands are not equal. Otherwise, returns false.
< (Less Than Operator)	x < y	Checks if the value of the left operand is less than the value of the right operand, then returns true. Otherwise, returns false.
> (Greater Than Operator)	x > y	Checks if the value of the left operand is greater than the value of the right operand, then returns true. Otherwise, returns false.
<= (Less Than or Equal To Operator)	x <= y	Checks if the value of the left operand is less than or equal to the value of the right operand, then returns true. Otherwise, returns false.
>= (Greater Than or Equal To Operator)	x >= y	Checks if the value of the left operand is greater than or equal to the value of the right operand, then returns true. Otherwise, returns false.

Machine for Relational_Operator:



Regular Expression: (==|!=|>=|<=|>|<)

4. Keywords & Reserved Words

a. Program

It represents the complete set of instructions that the computer executes. It defines the structure, flow and interaction of all components such as functions, execution blocks, input/output, and termination points.

Program	Definition
`function`	Defines a function or subroutine block that can return a value using return.
`start`	Explicitly defines the beginning of the program's main execution or the beginning of a specific code block.
`end`	Explicitly closes the current code block, conditional, or loop. Used in combination with other keywords to close blocks of code.
`echo`	Used to output or print text, variables, or expressions to the user's console or output screen.
`input`	Used to pause execution and accept data provided by the user, storing it in a specified variable.

b. Data Types

It defines the kind of values that variables can store and the operations that can be performed on those values.

Data Types	Definition
`number`	Declares a variable capable of storing whole integer numbers
`decimal`	Declares a variable capable of storing double-precision floating-point numbers
`string`	Declares a variable capable of storing an ordered sequence of characters
`boolean`	Declares a variable capable of storing boolean literals (true/false).
`list`	Declares a variable capable of storing collections of elements of the same type.

c. Loop

Control structures that let a block of code run repeatedly.

Loops	Definition
`for`	A count-controlled loop used for iteration where the number of repetitions is known or calculated.
`while`	A condition-controlled loop that runs as long as the condition remains true.
`do`	Paired with while to create a post-test loop, which ensures the code block executes at least once.

d. Conditional

Control structures that allow a program to make decisions whether certain conditions are true or false.

Conditionals	Definition
`if`	Executes a block of code if the specified condition evaluates to true.
`else`	Provides an alternative block of code to execute when the if-condition evaluates to false.
`switch`	Multi-way branch statement that evaluates an expression and executes code based on matching cases.
`case`	Defines an individual branch option within a switch statement.
`default`	Defines the fallback code block in a switch statement when no case matches.

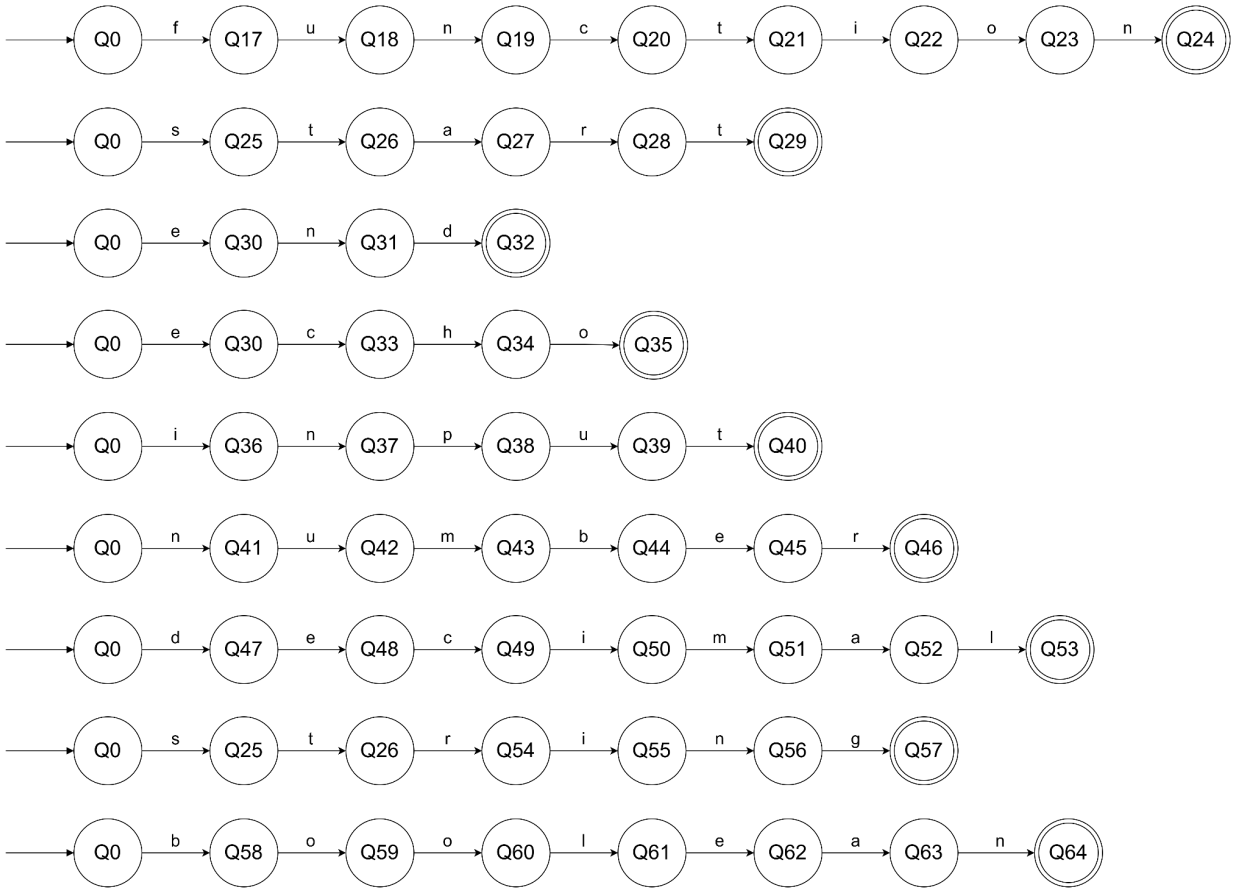
e. Reserved Words

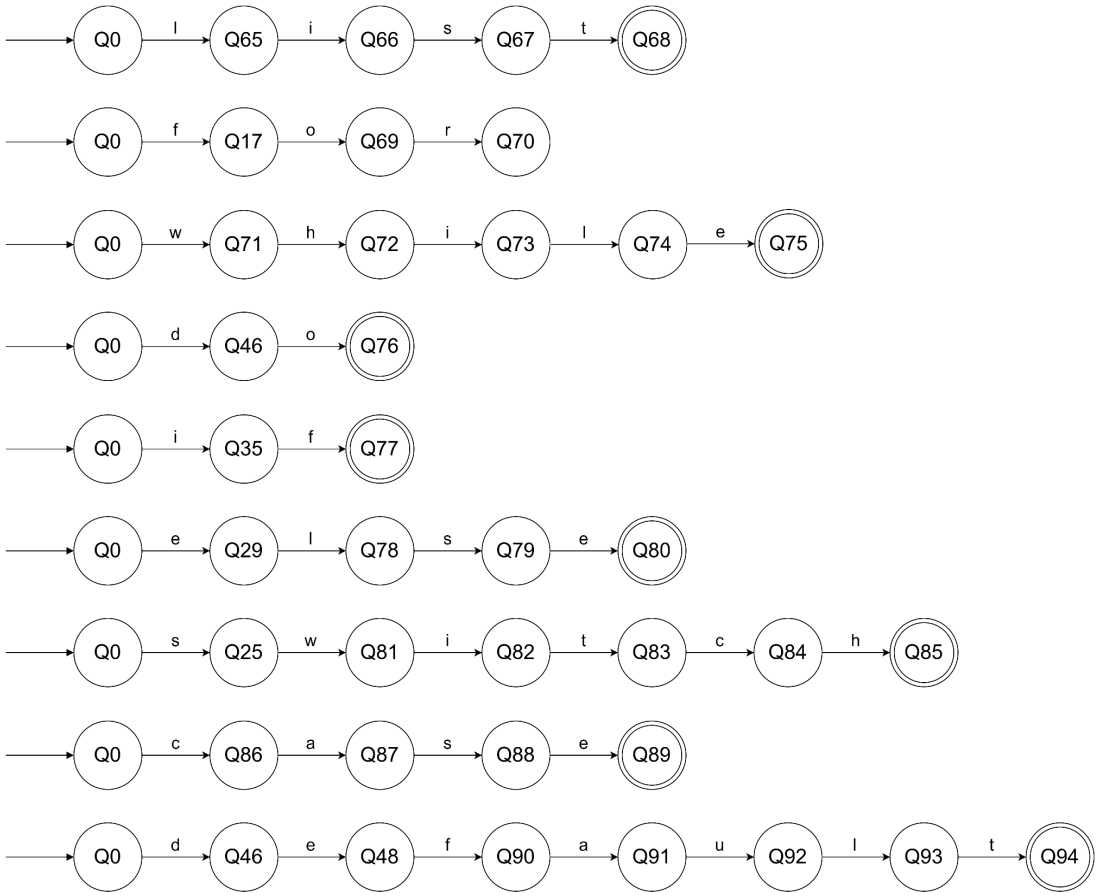
These are predefined keywords that have special meanings and are integral to the language's syntax and structure.

Reserved Words	Definition
`@`	Used to reference variables

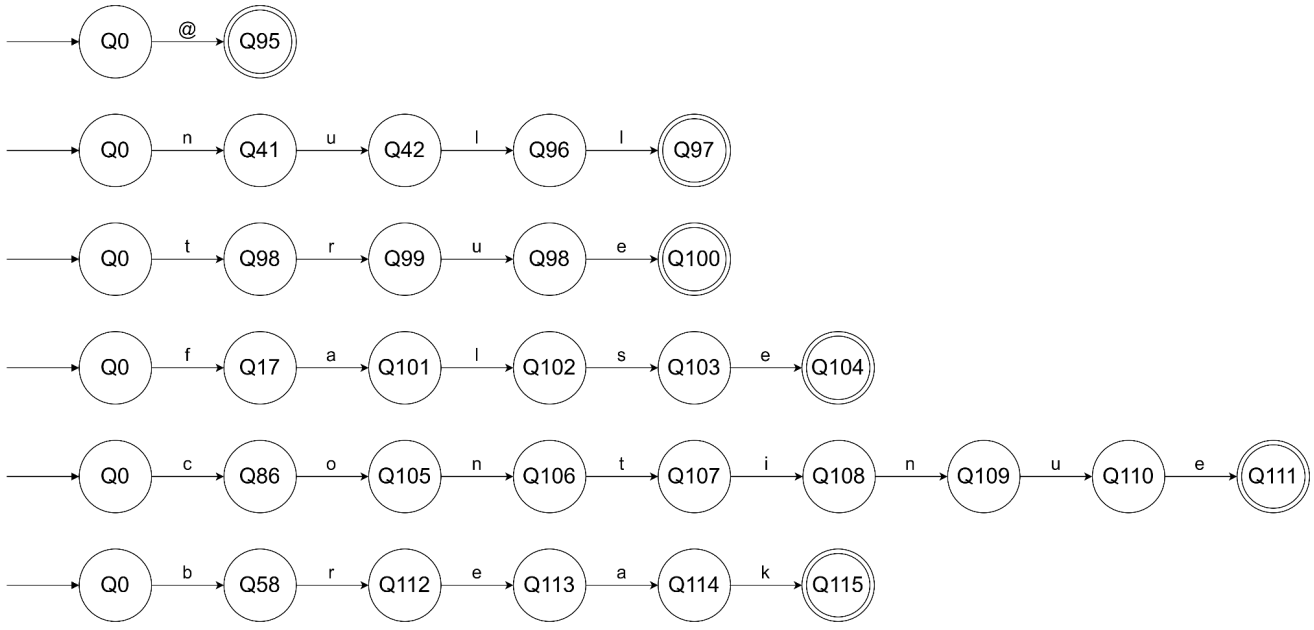
`NULL`	Used to represent no value or empty value
`true`	A boolean literal value.
`false`	A boolean literal value.
`continue`	Skips the remainder of the current loop iteration and forces the loop to proceed to the next cycle.
`break`	Exits the innermost loop or switch block immediately
`return`	Ends function and optionally returns a value
`new`	Used to create an instance of a data structure (like a list) or objects.
`this`	Refers to the current object or context within a function or code block.

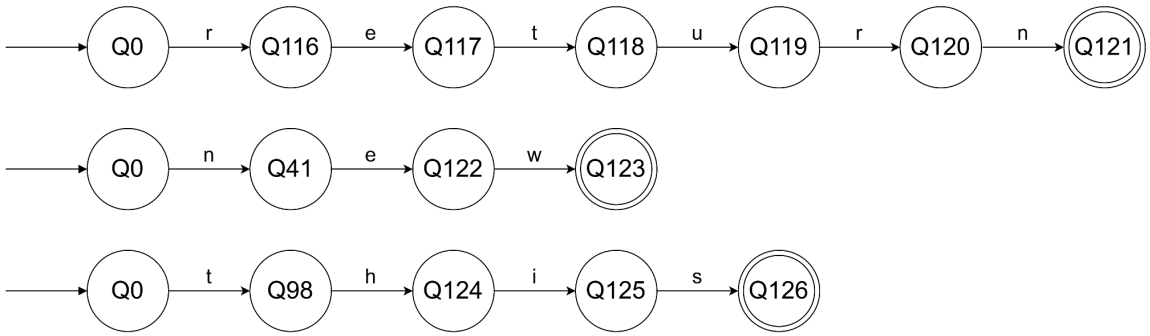
Machine for Keywords:





Machine for Noise Words:



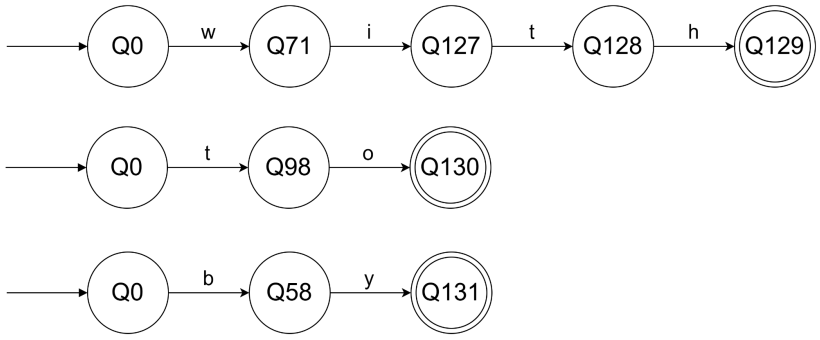


5. Noise Words

These are also keywords or connectors that are added to the syntax simply to make code more readable , but program logic and execution are not affected.

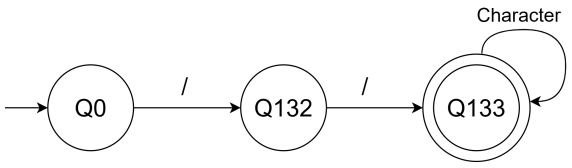
Noise Word	Example	Description
`with`	`function add with (number a, number b)`	Connects the function name to its parameter list. Used only for readability and has no runtime meaning.
`to`	`for i = 1 to 5`	Indicates the range limit in a for loop. Helps make the loop condition more readable.
`by`	`for i = 1 to 10 by 2`	Specifies the step or increment value in a for loop.

Machine for Noise Words:

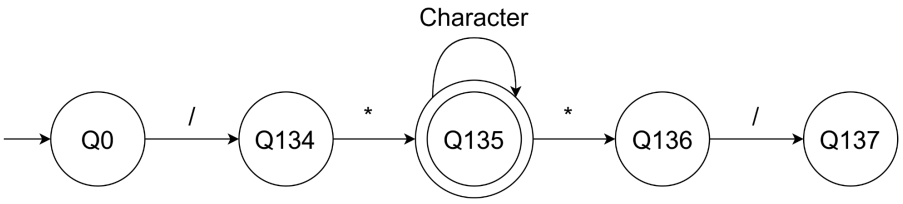


6. **Comments**

   a. **Single Line Comments: //**



   b. **Multiple Line Comments:** /* comments */



7. **Blanks**

These are whitespace characters such as spaces, tabs, and newlines. It separates tokens, organizes code, and makes it easier to read.

Element	Usage	Example
Blanks (Spaces)	Used to separate individual tokens such as identifiers, operators, and expressions.	`int total = a + b * c;`

Tabs	Used as additional whitespace for indentation, making nested blocks visually clear and easier to understand.	`start`
		`  for i 1 to 5`
		`  echo Count i`
		`  end for`
		`end`

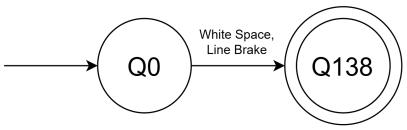
Newlines	Used to indicate the end of a statement. Each line is treated as a separate instruction unless a statement continues across multiple lines.	`number x = 5` `echo x`

8. Delimiters and Brackets

Delimiters are symbols that separate elements in code. Brackets are symbols that group or enclose code.

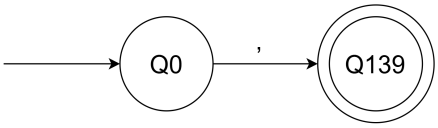
a. White space or line break (pressing the *Enter* key)

In Echo, we aim to design a simple and beginner-friendly programming language. To achieve this, we decided to eliminate the use of semicolon (;) as the standard statement terminator. Instead, code lines are separated using white spaces or line breaks (by pressing the Enter key).



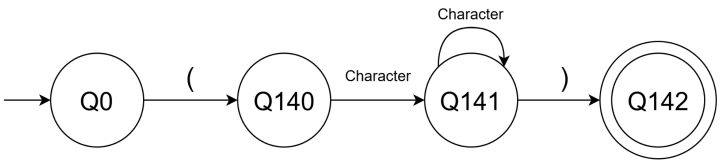
b. Comma (,)

Used to separate multiple identifiers or variables within a declaration, function call, or parameter list.



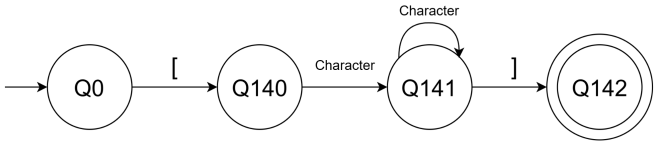
c. Parentheses ( )

Used to enclose parameters or arguments passed to functions and to define the order of operations in expressions or conditions.



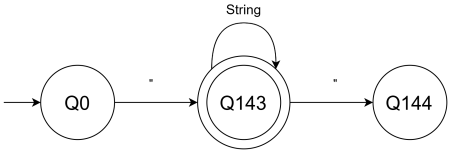
d. Brackets ([ ])

Used when declaring and accessing elements of arrays. Brackets also specify the index or position of an element within an array.



e. Quotation Marks (" ")

Used to enclose string literals, which represent sequences of characters.



9. Free-and-Fixed-Field Formats

In **Echo**, we aim to provide a simplified and user-friendly coding experience. To achieve this, Echo adopts a **free-field format**, which allows programmers to write code without being restricted by fixed column positions or alignment rules. By using a free-field format, Echo promotes a more flexible and readable style of programming that focuses on logic and understanding rather than strict formatting rules.

10. Expressions

It is a mix of variables, constants, operators, and functions that creates a value when evaluated.

a. Arithmetic Expressions

Precedence	Operator/s	Description	Associative
1	()	Parenthesis	N/A

2	* / %	Multiplication, Division, and Modulo	Left to Right
3	+ -	Addition and Subtraction	
4	=	Equal	

b. Unary Expressions

Precedence	Operator	Description	Associative
1	++ --	Postfix increment and decrement	Left to Right
2	++ --	Prefix increment and decrement	Right to Left

c. Boolean Expression

Precedence	Operator/s	Description	Associative
1	()	Parenthesis	N/A
2	!	NOT (Logical)	Right to Left
3	< <=	Less than and less than or equal to (Relational)	Left to Right
	> >=	Greater than and greater than or equal to (Relational)	
	== !=	Equal to and is NOT equal to (Relational)	
4	&&	AND (Logical)	
5	||	OR (Logical)	
6	=	Assignment	Right to Left
	*= /= %=	Multiplication, Division, and Modulo Assignment	
	+= -=	Addition and Subtraction Modulo	

11. Statements

A statement is a self-contained instruction that executes an action and may or may not produce a value.

a. Declaration Statements

A. data_type = {number, decimal, string, boolean, list}

B. Type must be specified for first declaration.

C. Multiple variables of the same type can be declared in one statement.

D. Initialization is optional.

Syntax	Example
<data_type> <identifier>	number age
<data_type> <identifier> <assignment_operator> <constant>	boolean isCs = true
<data_type> <identifier> <assignment_operator> <identifier>	string a = b
<data_type> <identifier>, <identifier>	string one, two
<data_type> <identifier> <assignment_operator> <constant>, <identifier> <assignment_operator> <constant>	decimal num1 = 20.5, num2 = 40.75
list <identifier>	list scores
list <identifier>[<index>]	list scores[5]
list <identifier> = [<values>]	list scores = [85, 90, 78, 92]

b. Input Statements

i. Syntax:

1. <identifier> = input(<data_type>)
2. <identifier> = input(<data_type>, <prompt_message>)

ii. Rules:

1. Variables must be declared before input.
2. Type in input() specifies expected data type.
3. Optional prompt message can be included.

A. Example:

```
start
string name
      name = input(string, "Input a name:")
      echo name
end
```

Output:

```
Input a name: John Doe
John Doe
```

c. Output Statements

A. *Echo* outputs the value of any expression.

B. String Insertion System (SIS) uses @ for variable interpolation.

C. SIS only works within double-quoted strings.

D. Escape @ with backslash (\@) for literal @ character.

Syntax	Example	Output
echo "<string>"	`Echo "Hello, World!"`	Hello, World!

echo "<string>"
echo "<string>"
echo "<string>"
	`echo "Line 1"`
	`echo "Line 2"`
	`echo "Line 3"`
		Line 1
		Line 2
		Line 3

echo <identifier>	`boolean isStudent = true` `echo isStudent`	true
echo <expression>	`number x = 5, y = 3` `echo x + y`	8
echo "<string> @<identifier>"	`number a = 10` `echo "Your number is @a"`	Your number is 10
echo "<string> @<identifier> <string>..."	`number age = 21` `echo "You are @age years`	You are 21 years old.

	old.”	

d. Assignment Statements

A. assignment_op = {=, +=, -=, *=, /=, %=}

B. Variable must be declared before assignment

C. Assignment type must match variable's declared type

Syntax	Definition	Example
<data_type> <identifier> <assignment_op> <value>	Assigns a literal value directly to the identifier.	boolean isStudent = true string name = “John Doe”
<identifier> <assignment_op> <identifier>	Assigns the value stored in another identifier to the left identifier.	number a *= b number c = 10
<identifier> <assignment_op> <expression>	Assigns the result of an expression to the right identifier.	c = a + b

e. Conditional Statements

Type	Syntax	Definition	Example
if	if <condition> <statements> end if	Executes statements only if the condition is true.	number num = 5 if num > 0 echo “positive num” end if
			Output: Positive num

if else	if <condition>
	<statements>
	else
	<statements>
	end if
		Runs the first block if the condition is true; otherwise runs the else block	number score = 68
			if score >= 75
			echo “passed”
			else
			echo “failed”
			end if

			Output: Failed
If else if else	if <condition> <statements> else if <condition> <statements>	It evaluates each condition in order, executing the first block whose	number score = 85 if score >= 90 echo "Grade: A" else if score >= 80

	else if
	  <statements>
	else
	  <statements>
	end if
		condition evaluates to true, and skipping the rest.	echo "Grade: B"
			else if score >= 70
			  echo "Grade: C"
			else
			  echo "Grade: F"
			end if
			Output:
			Grade: B

nested if-else	if <condition>
	  <statements>
	  if <-nested-condition>
	  <nested-statements>
	  else
	  <nested-statements>
	  end if
	else
	  <statements>
	end if
		Places one if-else statement inside another, where The inner if statement is executed only if the outer if condition is true.	number num = 5
			
			if num > 0
			  echo "Positive num"
			
			  if num > 10
			  echo "Large num"
			  else
			  echo "Small num"
			  end if
			
			else
			  echo "Negative num"
			end if
			Output:
			Positive num
			Small num

switch	switch <variables>
	case <value>
	  <statements>
	case<value>
	  <statements>
	default
	  <statements>
	end switch
		Compares a variable to multiple values and executes the matching case. Cases do NOT fall through (implicit break after each case).	string grade = "B"
			switch grade
			  case "A"
			  echo "Excellent"
			  case "B"
			  echo "Good"
			  case "C"
			  echo "Average"
			  default
			  echo “Improvement"
			endswitch
			Output:
			Good

f. Loop Statements

a. for loop

```
for <variable> = <initial_value> to <final_value>
  <statements>
end for
```

```
for <variable> = <initial_value> to <final_value> by <step>
   <statements>
end for
```

- **Single Statement –** Runs a set of instructions a certain number of times until the loop variable reaches its limit.

```
start
  for i = 1 to 5
    echo "Count: @i”
  end for
end
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

- **Multiple Statements –** Executes a block of multiple statements for each iteration until the loop variable reaches its limit.

```
start
  for i = 1 to 3
    echo "Iteration: @i”
    echo "Running loop..."
  end for
end
```

Output:

```
Iteration: 1
Running loop...
Iteration: 2
Running loop...
Iteration: 3
Running loop…
```

- **Nested for loops –** A for loop that contains one for loop inside another. The outer loop controls how many times the inner loop runs.

  Syntax:

```
for <outer_variable> = <initial_value1> to <final_value1>
  for <inner_variable> = <initial_value2> to <final_value2>
    <statements>
  end for
end for
```

```
for <outer_variable> = <initial_value1> to <final_value1> by <step1>
  for <inner_variable> = <initial_value2> to <final_value2> by <step2>
    <statements>
  end for
end for
```

Example:

```
start
  for x = 1 to 3
    for y = 1 to 2
      echo "x = @x, y = @y"
    end for
  end for
end
```

Output:

```
x = 1, y = 1
x = 1, y = 2
x = 2, y = 1
x = 2, y = 2
x = 3, y = 1
x = 3, y = 2
```

b. while loop

```
while <condition>
  <statements>
end while
```

- **Single statement –** Executes a single statement repeatedly as long as the condition remains true.

```
start
  number x = 1
```

```
  while num <= 3
    echo "Number: @x”
    num += 1
  end while
```

Output:

```
Number: 1
Number: 2
Number: 3
```

- **Multiple statements –** Executes all statements in the block repeatedly while the condition is true.

```
start
  number x = 1
  while num <= 3
    echo "Current: @x”
    num += 1
    echo "Loop continues..."
  end while
end
```

Output:

```
Current: 1
Loop continues...
Current: 2
Loop continues...
Current: 3
Loop continues...
```

- **Nested while loop –** A while loop that places one while loop inside another. The outer loop controls how many times the inner loop runs.

  Syntax:

```
while <condition1>
   while <condition2>
      <statements>
   end while
```

  Example:

```
start
  number x = 1
  while x <= 3
    number y = 1
    while y <= 2
      echo "x = @x, y = @y"
      y += 1
    end while
    x += 1
  end while
end
```

Output:

```
x = 1, y = 1
x = 1, y = 2
x = 2, y = 1
x = 2, y = 2
x = 3, y = 1
x = 3, y = 2
```

c. do-while loop

```
do
  <statement>
while <condition>
end do
```

- **Single Statement –** Executes the statement once before checking the condition, then repeats while it remains true.

```
start
  number x = 1
  do
    echo "Value: @x”
    x += 1
  while x <= 3
  end do
end
```

Output:

```
Value: 1
```

```
Value: 2
Value: 3
```

- **Multiple Statements —** Executes a block of multiple statements at least once, then continues looping while the condition remains true.

```
start
  number x = 1
  do
    echo "Round: @x”
    x += 1
    echo "Repeating..."
  while x <= 3
  end do
end
```

Output:

```
Round: 1
Repeating...
Round: 2
Repeating...
Round: 3
Repeating...
```

- **Nested do-while loops –** A do-while loop that places one do-while loop inside another. The outer loop executes at least once, and within each of its iterations, the inner loop also runs at least once.

  Syntax:

```
do
   do
      <statements>
   while <condition2>
   end do
while <condition1>
end do
```

Example:

```
start
  number x = 1
  do
    number y = 1
    do
      echo "x = @x, y = @y"
      y += 1
    while y <= 2
    end do
    x += 1
  while x <= 3
  end do
end
```

Output:

```
x = 1, y = 1
x = 1, y = 2
x = 2, y = 1
x = 2, y = 2
x = 3, y = 1
x = 3, y = 2
```

g. Functions Definition and Calls

i. Function names must follow identifier rules.

ii. Functions must not shadow built-in functions (mean, median, mode, etc.)

iii. Parameters must have explicit type declarations

iv. Return type appears before function name (if function returns value)

v. Functions with return type MUST include a return statement.

vi. Functions without return type CANNOT use return statements.

a. Function Definition Syntax

Type	Syntax

Function with Return Value and Parameters	function <return_type> <identifier> (<data_type> <param>, <data_type> <param>, …) <statements> return <expression> end function
Function without Return Value (with Parameters)	function <identifier> (<data_type> <param>, <data_type> <param>, …) <statements> end function
Function with Return Value and No Parameters	function <return_type> <identifier>() <statements> return <expression> end function
Function without Return Value and No Parameters	function <identifier>() <statements> end function

b. Function Call Syntax

<identifier>(<arg>, <arg>, ...)
<identifier>()

c. Examples:

Function Type	Example	Output
Function with Return	function number add(number x, number y) return x + y end function number result = add(5, 3) echo result	8
Function without return	function greet(text name, number age) echo "Hello, @name" echo "You are @age years old" end function greet("Alice", 25)	Hello, Alice You are 25 years old
Function with No Parameters	function displayWelcome() echo "Welcome to ECHO" echo "Data analysis made simple" end function	Welcome to ECHO Data analysis made simple

	displayWelcome()	