Members:

Bolado, Aaron John C.
Mahayag, David Geisler M.
Mirandilla, Sir Derick M.
Narisma, Anaise Nicole M.
Placente, Yesa V.
Prado, Mariella Alleli O.
Soliven, Prince Bryll R.

I. Introduction

simpliCty cleverly combines *"Simplicity"* with *"C,"* perfectly capturing the essence of an accessible, beginner-friendly programming language inspired by the foundational C language. The name suggests clarity, minimalism, and ease of learning

Designed as a simplified version of C, ***simpliCty*** aims to provide a more accessible entry point for novice programmers, allowing them to grasp programming concepts with ease while fostering a sense of creative expression. The language emphasizes "Creation, Creativity, and Completion," embodying a commitment to clarity and usability. By minimizing complexities and enhancing readability, this programming language seeks to empower beginners to understand programming fundamentals more intuitively and focus on building their problem-solving skills. In essence, ***simpliCty*** is envisioned as a tool that not only supports coding proficiency but also nurtures a deeper, more conscious approach to programming, resonating with the natural cycles of learning and growth.

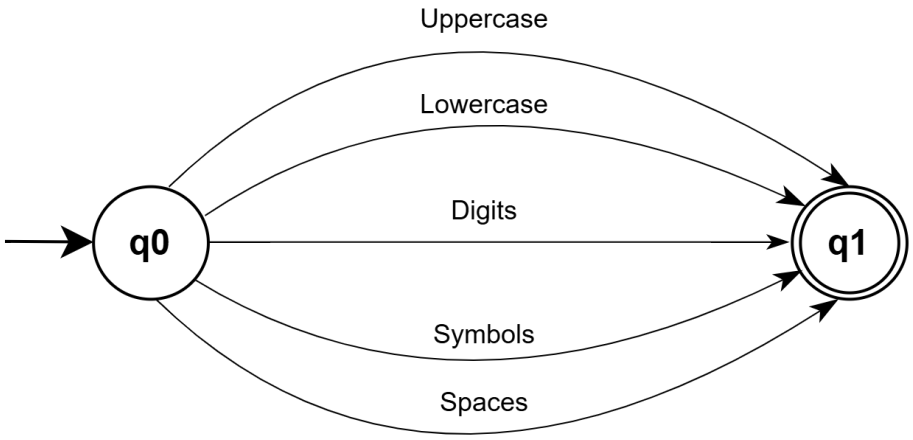
II. Syntactic Elements of Language

Character Set

This programming language accepts the following character set:

- **Characters** = {Letters, Digits, Symbols, Spaces}
- **Letters** = {Uppercase, Lowercase}
- **Uppercase** = {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}
- **Lowercase** = {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}
- **Digits** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- **Symbols** = {`, ~, !, @, #, $, %, ^, &, *, (,), -, _, [,], {, }, \, |, ;, :, ', ", ,, <, ., >, /, ?}
- **Spaces** = {Whitespace ' ', Tab '\t', Newline '\n'}

State Machine for Character Set:



Identifiers

In programming, an *identifier* is a name given to elements such as variables and functions. Identifiers allow programmers to assign readable, memorable labels to values, functions, and other entities in the code, making the program easier to understand and maintain. They must adhere to specific rules that ensure clarity and prevent conflicts with reserved keywords.

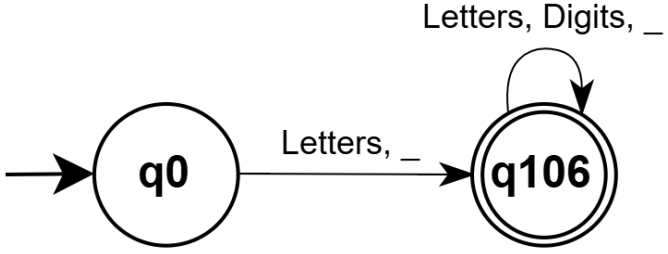
Rules for variables:

- A variable must start with an alphabet character (or underscores, **'_'**) followed by a combination of alphanumeric characters and underscores.
- It must be case-sensitive, meaning '*myVar*' and '*MyVar*' would be treated as different names.
- No other special character can be used other than the character underscore **'_'**.
- Must not match with any keyword and reserved words.

Rules for functions:

- A function must start with an alphabet (or underscores, **'_'**) followed by a combination of alphanumeric characters and underscores. It must be case-sensitive.
- No other special character can be used other than the character underscore **'_'.**
- Must not match with any keywords and reserved words.
- Does not support function overloading, so each function must have a unique name.

<u>State Machine for Identifiers:</u>



Regular Expression: [(Letters + _) (Letters + Digits + _)*]

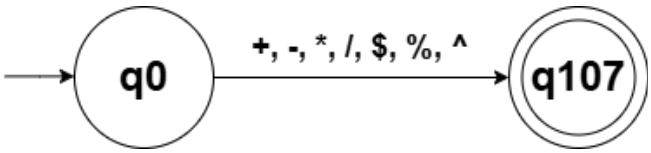
Operation Symbols

A. Arithmetic Operations

Arithmetic_Operators = {+,-,*,/,$,%,^}

OPERATOR	NAME	DESCRIPTION	EXAMPLE
+	Addition Operator	Adds two operands.	a + b
-	Subtraction Operator	Subtracts the second operand from the first.	a - b
*****	Multiplication Operator	Multiplies two operands.	a * b
/	Division Operator	Divides first operand by the second, keeping only the integer part.	a / b
$	Integer Division Operator	Divides two operands and keeps the decimal result.	a $ b
%	Modulo Operator	Returns the remainder when dividing two operands.	a % b
^	Exponentiation	Raises the first operand to the power of the second.	a ^ b

State Machine for Arithmetic Operators:



Regular Expression: (+ + - + * + / + $ + % + ^)

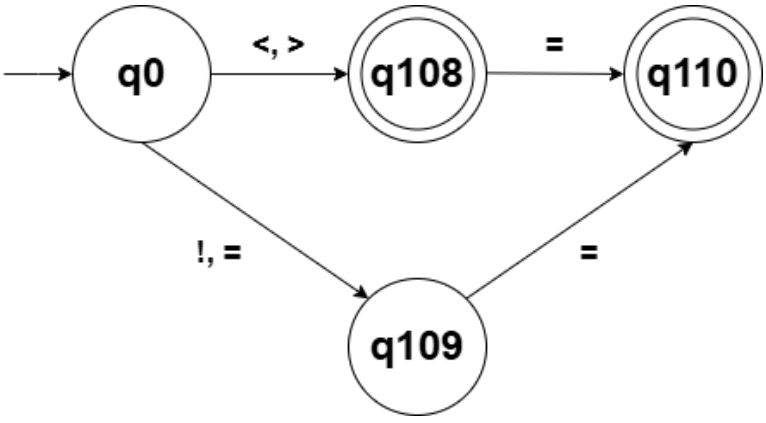
B. Boolean Operations

I. Relational Operators

Relational_Operators = {<,>,<=,>=,==,!=}

OPERATOR	NAME	DESCRIPTION	EXAMPLE
<	Less Than	True if the left operand is less than the right.	a < b
>	Greater Than	True if the left operand is greater than the right.	a > b
<=	Less Than or Equal To	True if the left operand is less than or equal to the right.	a <= b
>=	Greater Than or Equal To	True if the left operand is greater than or equal to the right.	a >= b
==	Equal To	True if both operands are equal.	a == b
!=	Not Equal	True if operands are not equal.	a != b

State Machine for Relational Operators:



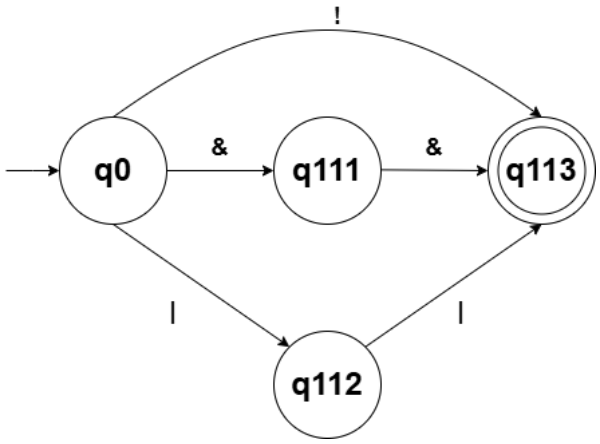
Regular Expression: ((< + >) =?) + (! + =) =

II. Logical Operators

Logical_Operators = {&&, ||, !}

OPERATOR	NAME	DESCRIPTION	EXAMPLE
&&	Logical AND	Returns true if both operands are true	a && b
||	Logical OR	Returns true if at least one operand is true.	a || b
!	Logical NOT	Inverts the truth value of the operand	!a

State Machine for Logical Operators:



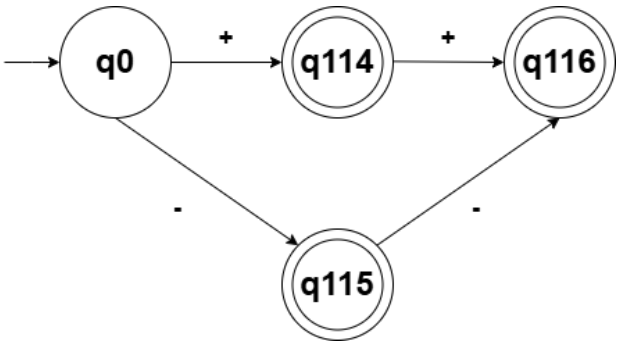
Regular Expression: ! + && + ||

C. Unary Operators

Unary_Operators = {+,-,++,--}

OPERATOR	NAME	DESCRIPTION	EXAMPLE
+	Unary Plus	Indicates a positive value	+a
-	Unary Negation	Negates the value (makes it negative)	-a
++	Increment	Increases the value of the variable by 1	a++
--	Decrement	Decreases the value of the variable by 1	a--

State Machine for Unary Operators:



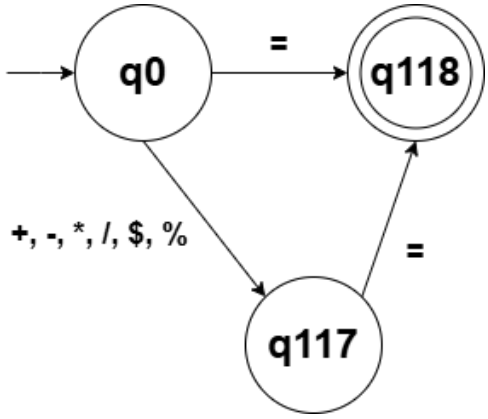
Regular Expression: (++?) + (--?)

D. Assignment Operators

Assignment_Operators = {=,+=, -=, *=, /=, $=, %=}

OPERATOR	NAME	DESCRIPTION	EXAMPLE
=	Assignment	Assigns the value of a variable	a = 2
+=	Addition Assignment	Adds the right operand to the left operand and assigns the result	a += 2
-=	Subtraction Assignment	Subtracts the right operand from the left operand and assigns the result	a -= 2
***=**	Multiplication Assignment	Multiplies the left operand by the right operand and assigns the result	a *= 2
/=	Division Assignment	Divides the left operand by the right operand and assigns the result	a /= 2
$=	Integer Division Assignment	Performs integer division and assigns the result	a $= 2
%=	Modulo Assignment	Performs modulo operation and assigns the result	a %= 2

State Machine for Assignment Operators:



Regular Expression: = + (+ + - + * + / + $ + %)=

Keywords and Reserved Words

A. Keywords

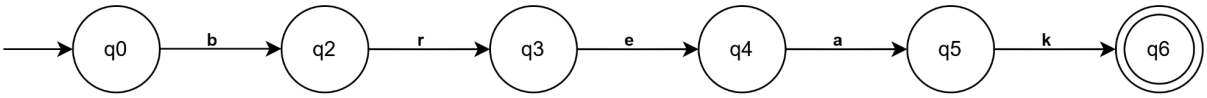
- Keywords define control flow and the core structure of the program. They hold specific syntactic roles and cannot be used as identifiers.

KEYWORD	DESCRIPTION
break	Exits the current loop, stopping any further iterations
continue	Skips to the next iteration in a loop, bypassing the remaining code in the current loop
default	Specifies a default case in switch-case statements
display *(printf in C)*	Outputs specified data to the console

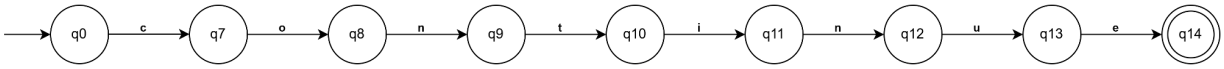
else	Specifies an alternative block if the if condition is false
for	Begins a loop with initialization, condition, and increment, iterating a set number of times
if	Starts a conditional block; executes the code if a condition is true
input *(scanf in C)*	Receives user input
main	Start of program execution
return	Exits a function and optionally returns a value
while	Begins a loop that continues while a specified condition is true

State Machine for Keywords:

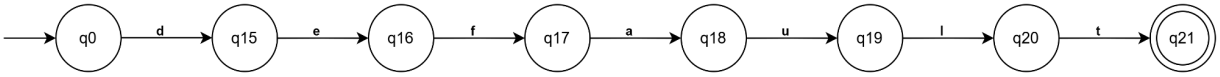
break



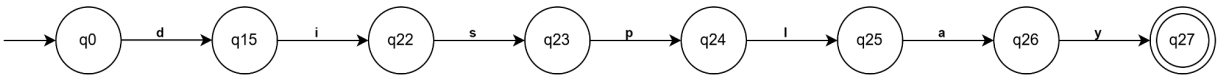
continue



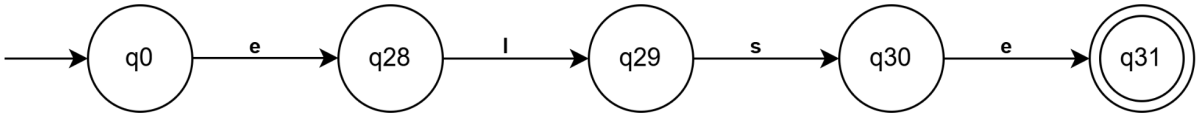
default



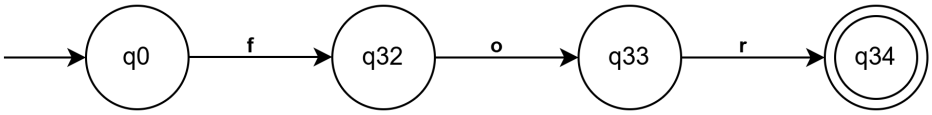
display



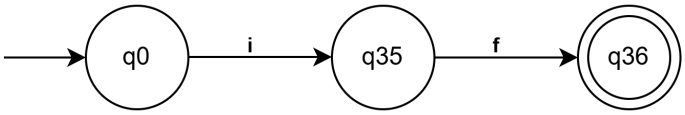
else



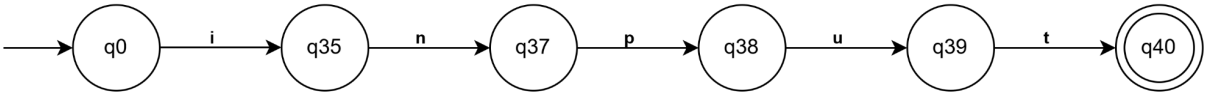
for



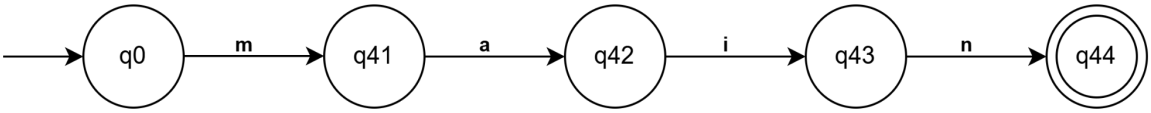
if



input

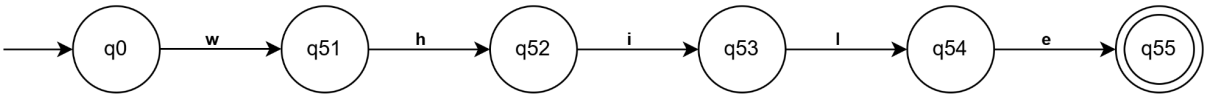


main



return

while



B. Reserved Words

- Reserved words typically represent data types or core constants and cannot be used as identifiers.

RESERVED WORD	DESCRIPTION
boolean *(bool in C)*	Data type for holding boolean data types (i.e. true and false)
character *(char in C)*	Data type for holding character values
constant *(const in C)*	Declares a constant that cannot be changed after initialization
false	Boolean literal holding a value equal to zero
float	Data type for holding floating-point values
integer *(int in C)*	Data type for holding integer values
null	Data type that represents absence of value
string *(char[] or str in C)*	Data type for holding texts, like words and sentences
true	Boolean literal holding a value not equal to zero
void	Function that returns nothing

State Machine for Reserved Words:

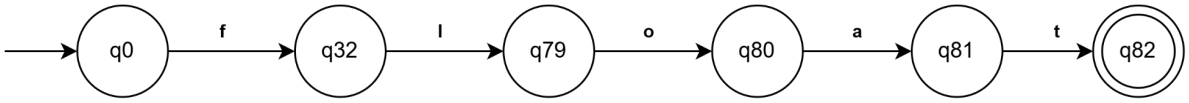
boolean

character

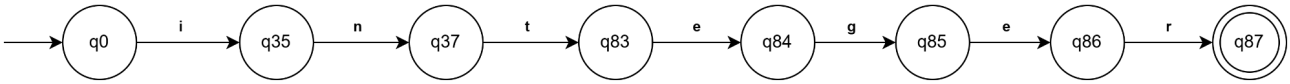
constant

false

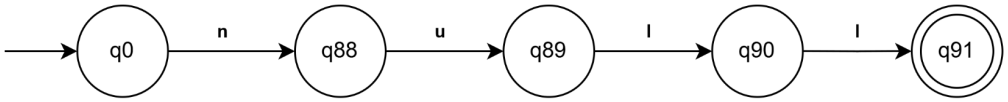
float



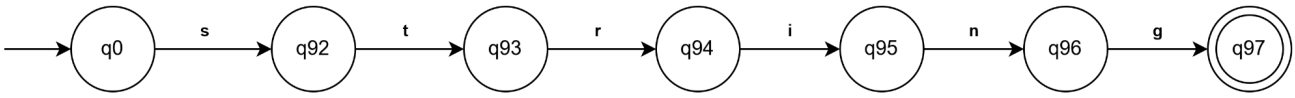
integer



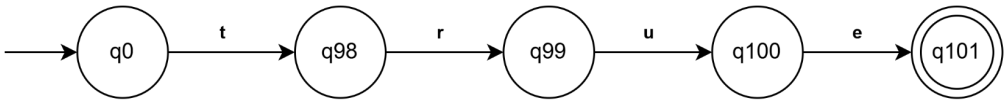
null



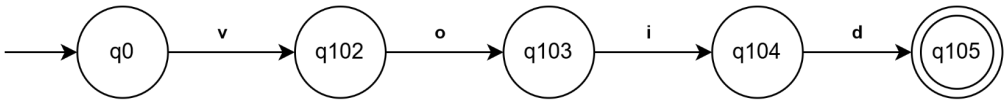
string



true



void



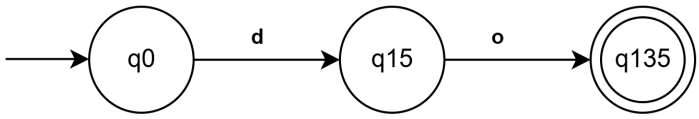
Noise Words

Noise words are reserved words that do not affect command execution but serve to make the code more readable and align it with natural language. These words are recognized by the compiler yet ignored in terms of execution, meaning they do not alter the functionality of the code.

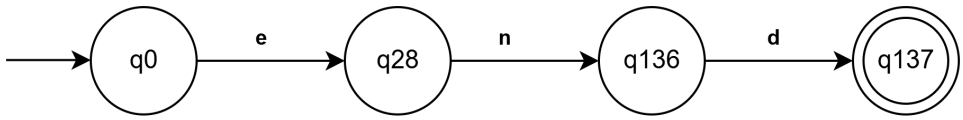
NOISE WORD	PURPOSE	EXAMPLE
do	Emphasizes action in loops, maintaining the traditional C structure but improving readability.	while (counter < 5) do display("Counting...");
end	Signifies the end of a block (function, loop, or conditional), which may help beginners identify where blocks start and stop without braces { }.	if (age > 18) then display("Adult"); end
let	Highlights variable declarations to make assignments more intuitive.	let age = 20; let name = "Juan";
then	Used to clarify the action following a condition. It adds flow to if and else if statements, making them read more like English sentences.	if (age > 18) then display("Adult");

State Machine for Noise Words:

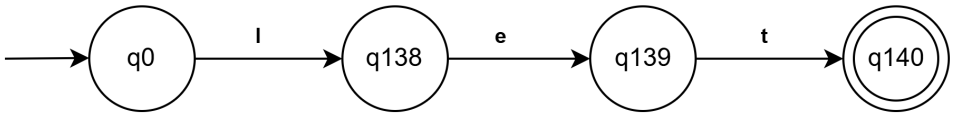
do



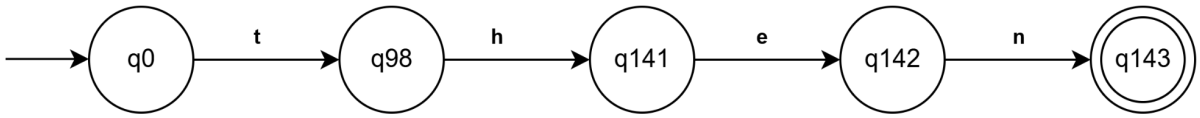
end



let



then



Comments

Comments are syntactic elements used to annotate code, providing explanations, or notes that are ignored by the compiler during execution. They serve multiple purposes, including improving code readability, aiding in debugging and documenting the functionality of code segments for future reference.

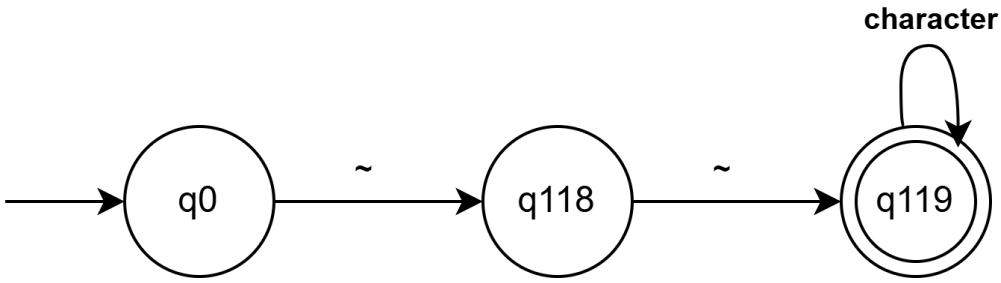
Rules

- Comments can be placed anywhere in the program, whether at the beginning, within, or at the end of code sections.

A. Single-line comments:

- ***Syntax:*** *~~single-line comment*
- ***Example:*** integer x; *~~x is for holding the value of age*

State Machine for Single-line comments:

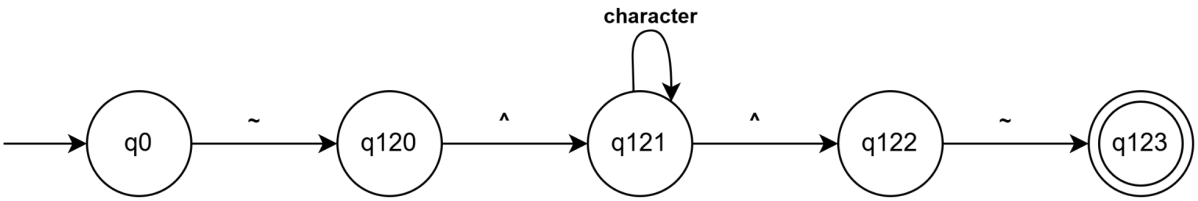


B. Multiple-line comments:

- ***Syntax:*** *~^ comments ^~*
- ***Example:***

~^ This is a multiple-line comment.
It can span multiple lines.
The compiler will ignore all this text. ^~

State Machine for Multiple-line comments:



Blanks (Spaces)

Whitespace characters, such as spaces (' '), tabs ('\t'), and newlines ('\n'), are generally ignored by the compiler outside of string literals. They are essential for separating tokens, improving code readability, and allowing flexible formatting.

Example:

- **Code with minimal spaces:** integer x=5;
- **Code with additional spaces:** integer x = 5;

Both examples yield the same result, as extra spaces outside strings are ignored.

However, whitespace inside quotation marks is preserved as written, allowing precise formatting within strings.

Example:

- **String literals:** “Hello World”

  Retains the exact spacing between "Hello" and "World" in the output.

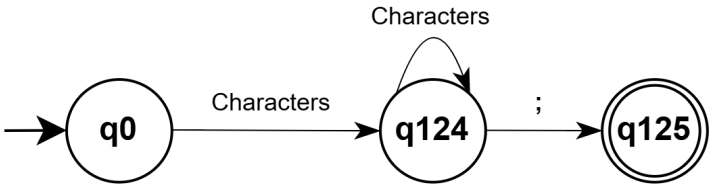
Delimiters & Brackets

Delimiters are symbols in programming that separate or group code elements, helping the compiler interpret code structure. They define boundaries, such as statement ends, code blocks, and function parameters. Common delimiters like ;, {}, (), and [] ensure code organization and readability.

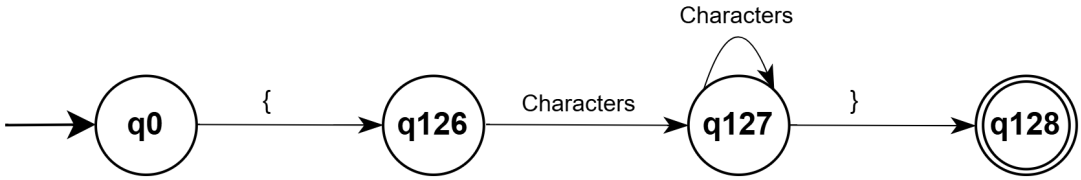
DELIMITER	DESCRIPTION	USES
;	Semicolon	Marks the end of a statement. It signals to the compiler where one statement finishes so that it can move on to the next statement.
{	Opening Curly Brace	Used to begin a block of code, such as in functions, loops, or conditional statements, indicating the start of grouped statements.
}	Closing Curly Brace	Used to end a block of code, matching with the opening curly brace to signal the end of grouped statements.
(	Opening Parenthesis	Used to call and define functions by enclosing parameters or to establish conditions in control structures.
)	Closing Parenthesis	Completes the function call or condition started by the opening parenthesis.
[	Opening Square Bracket	Used in arrays or lists to specify indices, indicating the start of an element reference or literal collection.
]	Closing Square Bracket	Completes the array or list element reference or literal collection, matching with the opening square bracket.

State Machine for Delimiters:

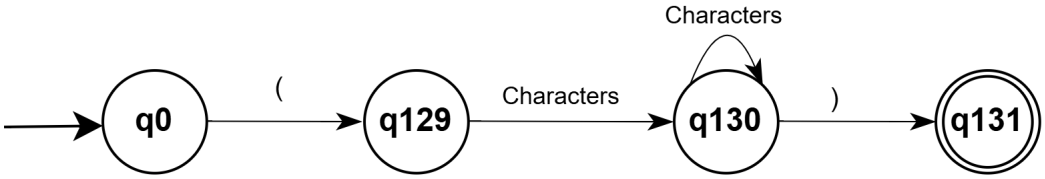
Semicolon



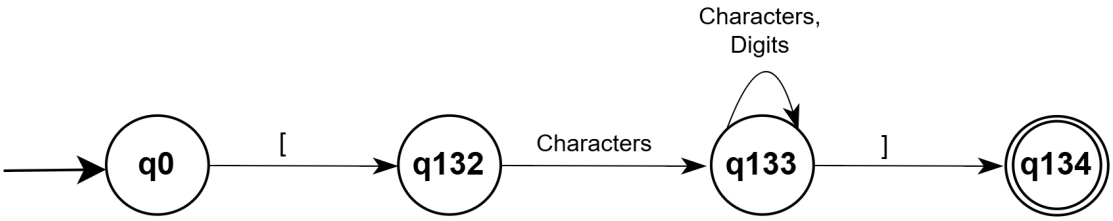
Curly Braces



Parenthesis



Square Brackets



Free-Field Format

This is a free-field programming language, code can be placed anywhere on a line, allowing flexible arrangement. Programmers have freedom in organizing code blocks for readability as long as the logical parameters are strictly followed.

Expressions

- Arithmetic Expression

LEVEL OF PRECEDENCE	OPERATOR	DESCRIPTION	ASSOCIATIVITY
1	()	Parenthesis	Left-to-right
2	^	Exponential Operator	Left-to-right
3	*, /, %, $	Multiplication, Division, Modulus, Integer Division	Left-to-right
4	+, -	Addition, Subtraction	Left-to-right

- Unary Expression

LEVEL OF PRECEDENCE	OPERATOR	DESCRIPTION	ASSOCIATIVITY
1	++x, --x	Prefix Incrementation, Prefix Decrementation	Right-to-left
2	x++, x--	Postfix Incrementation, Postfix Decrementation	Left-to-right

- Boolean Expression

LEVEL OF PRECEDENCE	OPERATOR	DESCRIPTION	ASSOCIATIVITY
1	!	Logical NOT negates the truth	Right-to-left

		value of a boolean expression. If the expression evaluates to true, it returns false, and if it evaluates to false, it returns true.	
2	<, <= >, >=	Checks if the left value is less than the right value; Checks if the left value is less than or equal to the right value Checks if the left value is greater than the right value; Checks if the left value is greater than or equal to the right value	Left-to-right
3	==, !=	Checks for equality between values, if equal or not equal	Left-to-right
4	&&	Logical AND returns true if both of its operands are true. If either operand is false, the result is false. If the first operand is false, the second operand is not evaluated.	Left-to-right
5	||	Logical OR returns true if at least one of its operands is true. If both operands are false, the result is false. If the first operand is true, the second operand is not evaluated.	Left-to-right
6	= +=, -=	Direct Assignment Assignment by sum, difference	Right-to-left

	*=, /=, %=, $=	Assignment by product, quotient, remainder, and floating-point	

Statements

A. Declaration Statements

– A ***declaration*** statement is used to declare a variable with a specific data type. It specifies an identifier, such as the name of a variable or a function. Each declaration must be terminated with a semicolon (;).

– Syntax:

```
data_type identifier;
```

– Example:

```
integer num;
float grade, weight;
```

B. Assignment Statements

– Syntax:

```
data_type = expression;
identifier assignment _operator expression;
```

– Example:

```
integer num = 10;
float grade = num * 1.75;
num += 5;
boolean on = true, off = false;
```

C. Array Declaration and Assignment

– Syntax:

```
data_type identifier[size];
identifier[index] = value;
```

– Example:

```
integer numbers[5];
numbers[0] = 10;
numbers[1] = 20;
```

D. Conditional (if, if else, if else if else)

TYPES	SYNTAX	EXAMPLE	NESTED EXAMPLE
If	if (condition) { ~~ statements }	if (num >= 11 && num <= 99) { display("Number is between 11 and 99\n"); }	if (num > 10) { if (grade < 100) { display("Number is between 11 and 99\n"); } }
if-else	if (condition) { ~~ statements } else { ~~ statements }	if (grade < 75) { display("Failed\n"); } else { display("Passed\n"); }	if (grade < 75) { display("Failed\n"); } else { if (grade >= 75 && grade < 100) { display("Passed\n"); } else if (grade == 100) { display("Perfect Score: Passed\n"); } }
if-else if-else	if (condition1) { ~~ statements } else if {condition2) { ~~ statements } else (~~ statements }	if (age < 13) { display("Child\n"); } else if (age >= 13 && age < 20) { display("Teenager\n"); } else { display("Adult\n");	if (age < 13) { display("Child\n"); if (age < 6) { display("Young Child\n"); } else { display("Older

		}	Child\n"); } } else if (age >= 13 && age < 20) { display("Teenager\n"); if (age <= 15) { display("Young Teen\n"); } else { display("Older Teen\n"); } } else { display("Adult\n"); if (age < 65) { display("Young Adult\n"); } else { display("Senior\n"); } }

E. Iterative Statements

LOOP TYPE	SYNTAX	DESCRIPTION	EXAMPLE	NESTED EXAMPLE
while	while (condition) {	Repeats a block of code as long as	while (i < 5) { display("%d	while (i < 3) { while (j < 2) {

	~~ statements }	the condition is true. The condition is evaluated before each iteration, and if it's initially false, the loop doesn't execute.	",i); i++; }	display("i = %d, j = %d\n", i, j); j++; } i++; }
for	for (initialization; condition; update) { ~~ statements }	A concise loop structure that initializes a variable, evaluates a condition, and updates in each iteration. Commonly used when the number of iterations is known.	for (integer i = 0; i < 5; i++) { display("%d ",i); }	for (integer i = 0; i < 3; i++) { for (integer j = 0; j < 2; j++) { display("i=%d, j=%d\n", i, j); } }

F. Function Statements

FUNCTION TYPE	SYNTAX	DESCRIPTION	EXAMPLE
Function Definition	return_type identifier(parameter_list) { ~~ statements }	Defines a function with a specific return type, name, and parameters. The function body contains the statements executed when the function is called.	integer add(integer a, integer b) { return a + b; }
Function Call	identifier(argument_list);	Calls a function by its name, passing arguments if	integer result = add(2, 3);

		required. The function executes and returns a value if applicable.	*~~ Output = 5*
Return Statement	return expression;	Returns a value from a function to the caller. The type of expression should match the function's return type.	integer add(integer a, integer b) { return a + b; } *~~ Note: In `main`: integer result = add(2, 3);*
Void Function	void identifier(parameter_list) { ~~ statements }	A function with a void return type, meaning it does not return any value. Often used for functions performing actions rather than calculations.	void greet() { display("Hello, World!"); } *~~ Note: Call in `main`: greet();*
Parameter Passing	data_type parameter_name	Specifies the type and name of each parameter in the function definition. Arguments must match the defined parameters in type and order.	integer add(integer a, integer b) { return a + b; } *~~ Note: In `main`: integer result = add(2, 3);*
Prototype Declaration	return_type identifier(parameter_list);	Declares a function's name, return type, and parameters at the beginning of the program to inform the compiler about	integer add(integer a, integer b); *~~ Note: Prototype at the top, then*

		its existence. Useful for organizing code.	*defined later in the code*

G. Output Statement (display)

The **display** function simplifies output by automatically handling data types and combining strings with variables. No format specifiers are needed, and multiple items can be printed in a single statement.

- Syntax:

```
display("message”);
display(variable_name);
display("Message", variable_name, "additional message");
```

- Examples:

Basic String Output	display("Hello, welcome!") ~~ Output: Hello, welcome!
Variable Output	name = "Juan" display("Your name is ", name) ~~ Output: Your name is Juan
Multiple Variables and Text	name = "Juan" age = 21 display("Name:", name, "| Age:", age) ~~ Output: Name: Juan | Age: 21
Combining Expressions	a = 10 b = 5 display("Sum of a and b is", a + b) ~~ Output: Sum of a and b is 15

H. Input Statement (input)

The **input** function takes a string prompt and displays it to the user, storing the entered data directly into the specified variable. This function doesn’t require format specifiers, allowing for straightforward assignment of user input to variables.

- Syntax:

  identifier = input("message")

- Examples:

Simple Input	name = input("What is your name?") *~^ If user enters: Juan* *Stored Value: name = "Juan" ^~*
Numeric Input	age = input("How old are you?") *~^ If user enters: 21* *Stored Value: age = 21 ^~*