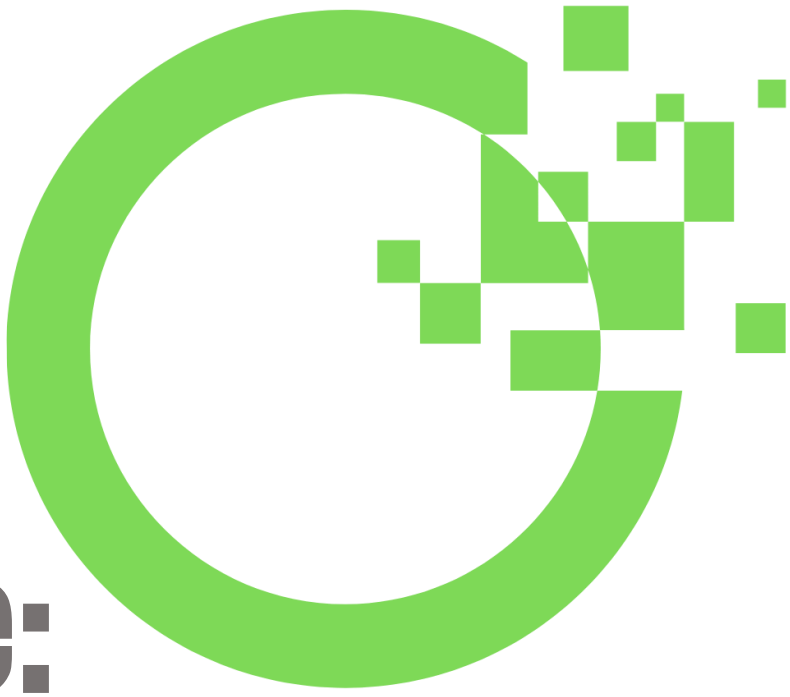


FirstByte:

Nurturing Novices, One Bit at a Time

Aquino | Bautista | Jugueta | Labinay
Masarque | Peña | Uy

Table of Contents

About……4

Introduction…… 5

Chapter 1: Lexical Analysis (Syntactic Elements of Language)

Character Set…… 6

Identifiers……...6

Operation Symbol……..…7

Keywords and Reserved Words……...12

Noise Words……21

Comments……23

Blanks…… 24

Delimiters and Brackets……...24

Free-and-Fixed Field Formats……25

Expression……...26

Statements……...27

Chapter 2: Syntax Analysis

Production Rule

Declaration Statement……34

Input / Output Statement……36

Assignment Statement……40

Assignment Operator……...43

Unary Operator……46

Boolean Logic……48
Boolean Relation……52
Conditional Statement……54
Iterative Statement……...59
New Principles……66
New Principle: Force……...68

About

This documentation, hereto entitled **“First Byte: Nurturing Novices, One Bit at a Time”** presented and submitted by Mark Joseph J. Aquino, Pauline Ann P. Bautista, Ashley Sheine N. Jugueta, Stefen V. Labinay, Andy D. Masarque, Ma. Charissa B. Peña, and Lord Allain B. Uy of BSCS 3-5 as a final requirement for the subject Principles of Programming Language.

With the guidance of the students’ professor, Mr. Montaigne G. Molejon, they succeeded in creating a programming language.

Introduction

Introducing "FirstByte", a versatile, user-friendly, functional and procedural programming language created for beginners and novice programmers. Featuring an easy-to-learn designed syntax that may serve as a foundation for those who are stepping into the world of coding, making it a less-difficult journey for newcomers to grasp the fundamentals and principles of programming. With FirstByte, learning to code becomes less intimidating and more enjoyable, turning it into an empowering experience. Also, this innovative programming language introduces a new feature to empower beginners by integrating math and physics equations into their coding experience. This accessible feature not only gives the user a new learning curve but also opens new possibilities for incorporating scientific principles into the world of programming.

"FirstByte" is derived from two English words, "first" and "byte." The term "first" represents the idea of the starting point, while "byte" is one of the fundamental units of digital information and is used to represent a wide range of data in computers. Hence, the combination of these two words signifies the very first step for beginners in the journey of programming. It implies that this language is designed for those people with no background in coding to easily grasp the fundamental concepts of programming.

The Goals of FirstByte revolves around providing accessibility and intuitiveness for users new to coding, enabling individuals without prior experience to delve into programming. It aims to empower users to create programs and solve problems using technology. To achieve these goals, the language aims to develop a simplified syntax that is easy to read and write, and it utilizes natural language constructs and terminology that are familiar to non-programmers.

FirstByte was inspired by the strengths of Python and C, driven by a vision to enhance accessibility, readability, and user-friendliness in programming. Our vision extends beyond mere accessibility; envisioning a language that fosters growth and encourages creativity among programmers of all levels. Programming should be an enjoyable and empowering experience, not a daunting obstacle. By streamlining the learning process and removing unnecessary barriers, this will cultivate a community of engaged and enthusiastic programmers.

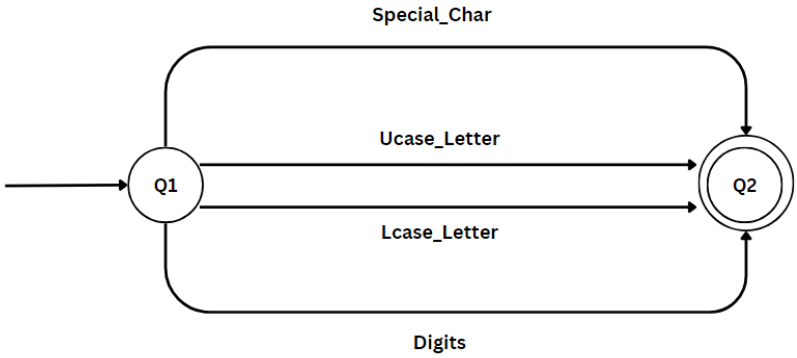
Chapter 1: Lexical Analysis

SYNTACTIC ELEMENTS OF LANGUAGE

1. Character Set

- **Characters** = {Alphabet, Digits, Special_Char}
- **Alphabet** = {Ucase_Letter, Lcase_Letter}
- **Ucase_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}
- **Lcase_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}
- **Digits** = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- **Integers** = Digits* | -Digits*
- **Special_Char** = {., +, -, *, /, %, >, <, =, “, ‘, ,, ;, |, \, (,), [,], _, ^, ~, &, <space>}

Machine for Character Set

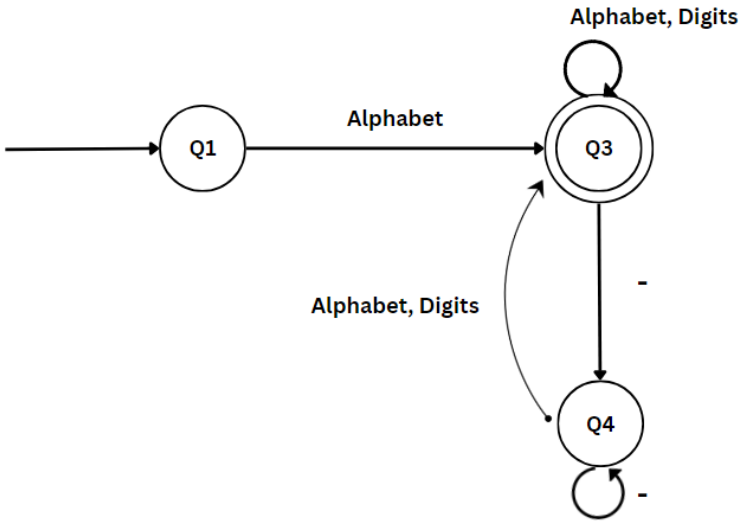


2. Identifiers

Rules

- An identifier must start with an alphabet, either uppercase or lowercase. It is case sensitive.
- Identifier can also start with an underscore (_), no other special characters is allowed.
- The character after the first character of an identifier can be an Alphabet, Digits, or Underscore.
- No keywords and reserved words can be used as an identifier

Machine for Identifiers



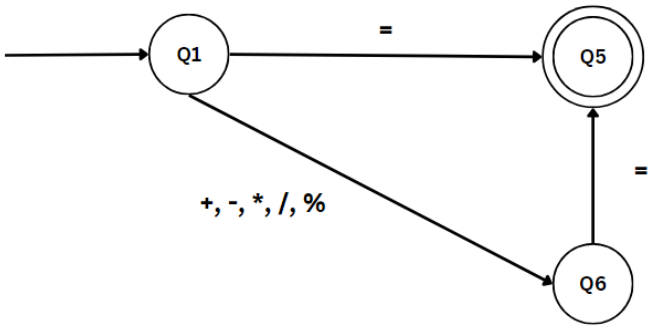
3. Operation Symbol

- **Assignment_Operator** = { =, +=, -=, *=, /=, %=}

Assignment_Operator	**Example Expression**	**Description**
= (Assignment Operator)	X = 3	Assign the value of the variable X to 3.
+= (Addition Assignment Operator)	X += 3	Adds 3 to the current value of X and returns the sum.
-= (Subtraction Assignment Operator)	X -= 3	Subtracts 3 to the current value of X and returns the difference.
*= (Multiplication Assignment Operator)	X *= 3	Multiplies the current value of X by 3 and returns the product

/= (Division Assignment Operator)	X /= 3	Divides the current value of X to 3 and returns the quotient.
%= (Modulo Assignment Operator)	X %= 3	Divides the current value of X to 3 and returns the remainder.

Machine for Assignment Operator

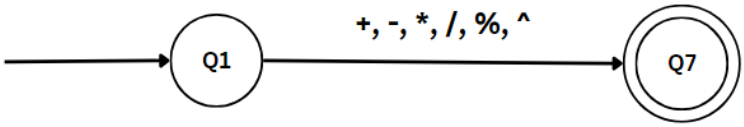


- **Arithmetic_Operator =** {+, -, *, /, %, pow}

Arithmetic_Operator	**Example Expression**	**Description**
+ (Addition Operator)	X + Y	Adds the value of X and Y.
- (Subtraction Operator)	X – Y	Subtracts the value of X by Y.
* (Multiplication Operator)	X * Y	Multiplies the value of X and Y.
/	X / Y	Divides the value of X by Y.

(Division Operator)		
% (Modulo Operator)	X % Y	Divides the value of X by Y and returns the remainder.
^ (Exponent Operator)	X ^(n)	Computes the power of base X to exponent n

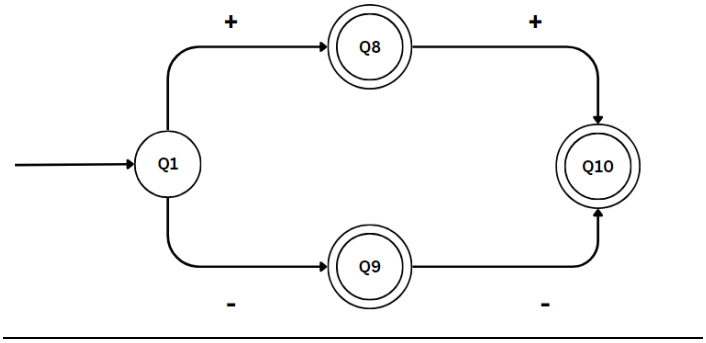
Machine for Arithmetic Operator



- **Unary_Operator** = {+, -, ++, -- }

Unary_Operator	**Example Expression**	**Description**
+ (Unary Plus Operator)	+X	Indicates that the value of X is positive.
- (Unary Minus Operator)	-X	Indicates that the value of X is negative.
++ (Increment Operator)	++X or X++	Increases the value of operand X by 1.
-- (Decrement Operator)	--X or X--	Decreases the value of operand X by 1.

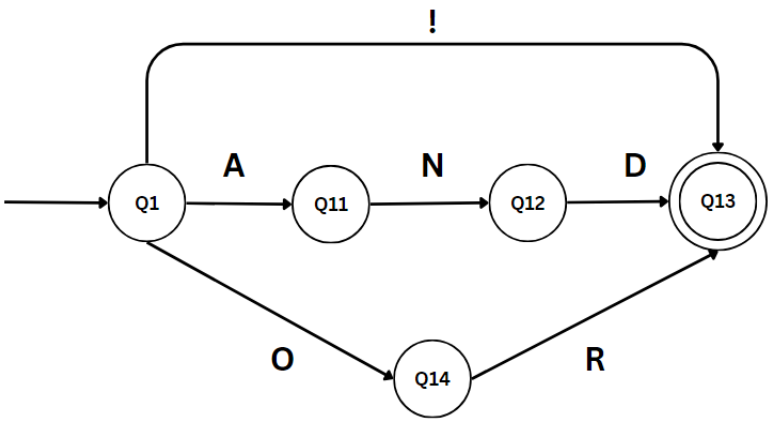
Machine for Unary Operator



- **Boolean_Logic** = {!, OR, AND}

Boolean_Logic	**Example Expression**	**Description**
! (Logical NOT Operator)	{X = 5} !(X)	Returns the opposite value of the expression. **!(X) returns FALSE.**
OR (Logical OR Operator)	{X = 5} (X > 0 OR X < 4)	Returns a value (true) if one or more statements in a condition are true, else return false. **(X>0 OR X<4) returns TRUE.**
AND (Logical AND Operator)	{X = 5} (X>0 AND X<4)	Returns a value (true) if both statements in a condition are true, else return false. **(X>0 AND X<4) returns FALSE.**

Machine for Boolean_Logic

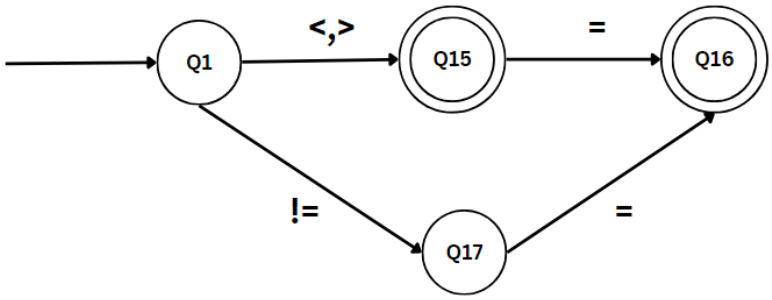


- **Boolean_Relation =** {==, !=, >, <=, >=}

Boolean_Relation	**Example Expression**	**Description**
== (Is Equal to Operator)	{X = 5, Y = 10} (X == Y)	Checks if the value of operands is equal then returns true, else returns false. **(X == Y) returns FALSE.**
!= (Is Not Equal to Operator)	{X = 5, Y = 10} (X != Y)	Checks if the value of operands is not equal then returns true, else returns false. **(X != Y) returns TRUE.**
> (Greater Than Operator)	{X = 5, Y = 10} (X > Y)	Checks if the value of the left operand is greater than the value of the right operand then returns true, else returns false. **(X > Y) returns TRUE**.
< (Less Than Operator)	{X = 5, Y = 10} (X< Y)	Checks if the value of the left operand is less than the value of the right operand then returns true, else returns false. **(X < Y) returns FALSE.**

<= (Less Than or Equal to Operator)	{X = 5, Y = 10} (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, else returns false. **(X <= Y) returns TRUE.**
>= (Greater Than or Equal to Operator)	{X = 5, Y = 10} (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, else returns false. **(X >= Y) returns FALSE.**

Machine for Boolean Relation



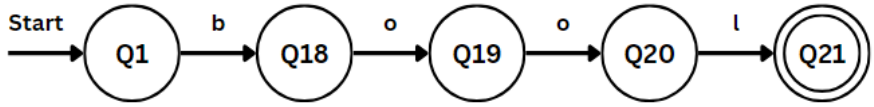
4. Keywords and Reserved Words

Keywords	Definition
bool (boolean)	a data type that can only have one of two values: true or false.
def (default)	is executed if no case constant expression value is equal to the value of the expression.
str (string)	consists of one or more characters, which can include letters, number, and other types of characters.
int (integer)	are whole numbers that can have both zero, positive, and negative values but no decimal values (0,-1,1).
deci (float and deci)	float and double data type combined.
char (character)	is used for declaring character type variables.

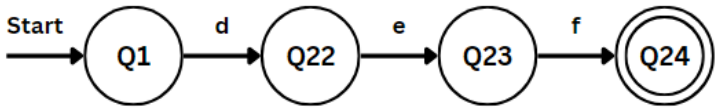
when (if)	a conditional statement that is used to check a condition and execute if the condition holds true.
otherwise (else)	an alternative statement that contains the block of code that executes if the conditional expression in the if statement resolves to a false value
loop (for)	an iterative statement that is used to check for certain conditions and then repeatedly execute a block of code as long as conditions are met.
to	a statement used together with the "loop" keyword to indicate the range of values
input (scan)	a function that takes an input from the user.
output (print)	a function that prints an output on the screen.
stop (break)	terminates and exits a loop.
jump (GOTO)	statement that is used to jump to a specific location or label within the source code.
arithSeq (Arithmetic Sequence)	a function that calculates the nth term of an arithmetic sequence where an is the nth term, a1 is the first term, n is the position of the term in the sequence, and d is the common difference between terms.
arithSer (Arithmetic Series)	a function that calculates the sum of the first *n* terms of an arithmetic sequence where Sn is the sum of the first n terms, n is the number of terms, a1 is the first term, and an is the nth term.
geoSeq (Geometric Sequence)	a function that calculates the nth term of a geometric sequence where an is the nth term, a1 is the first term, n is the position of the term in the sequence, and r is the common ratio between terms.
geoSer (Geometric Series)	a function that calculates the sum of the first *n* terms of a geometric sequence where Sn is the sum of the first n terms, n is the number of terms, a1 is the first term, an is the nth term, and r is the common ratio between terms.
distance	a function that calculates the distance between two points given the distance and coordinates of the two points.
slope	a function that calculates the slope of a given line given the coordinates of two points on the line.

pythagorean	a function that calculates the length of a missing side of a triangle where a and b are the lengths of the other two sides and c is the length of the hypotenuse.
quadratic	a function that calculates the solution of the quadratic equation given a, b, and c where a and b are coefficients and c is a constant.
force	a function that calculates the force given the mass and acceleration.
work	a function that calculates the work given the force and distance.
acceleration	a function that calculates acceleration given the force of the object and its mass.
power	a function that calculates the power given the work and time.
momentum	a function that calculates the momentum given the mass of the object and its velocity.
potential	a function that calculates the potential energy given the mass of the object, acceleration due to gravity, and the height in meters.
kinetic	a function that calculates the kinetic energy given the mass of the object and its velocity.
toInt (to integer)	a function that converts decimal data type to integer data type.
toDeci (to decimal)	a function that converts integer data type to decimal data type.
toStr (to string)	a function that converts integer or decimal data type to string data type
pi	a constant variable with a constant value of PI: 3.141592653589793.
accGrav	a constant variable with a constant value of the magnitude of the acceleration due to gravity: 9.8 m/s2.
euler	a constant variable commonly used in mathematics with a value of 2.718281828459045
goldenRatio	a constant variable with a value of 1.618033988749895

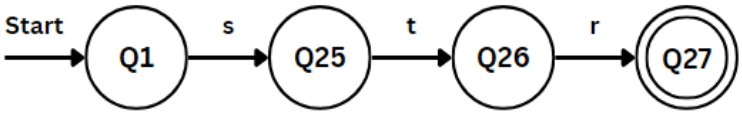
bool



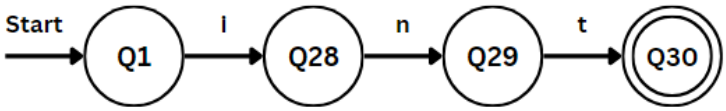
def



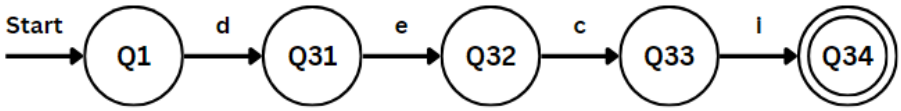
str



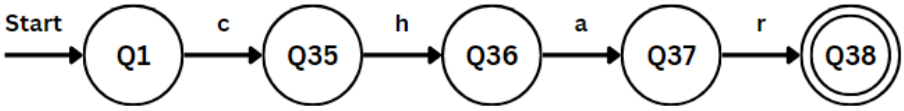
int



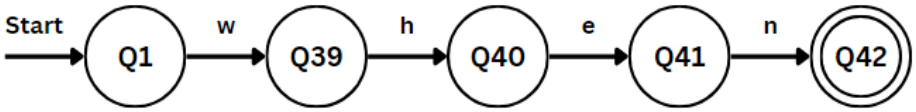
deci



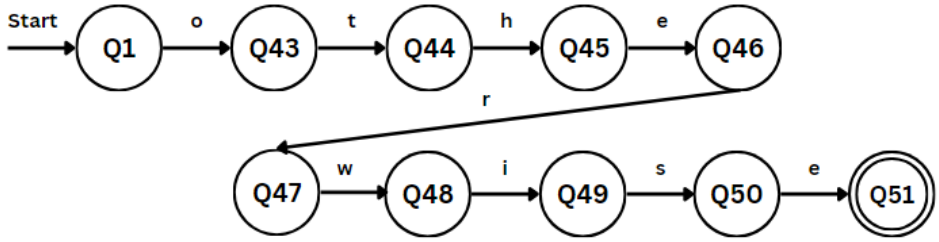
char



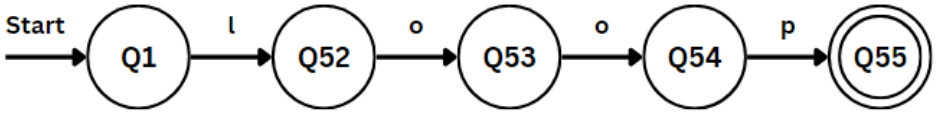
when



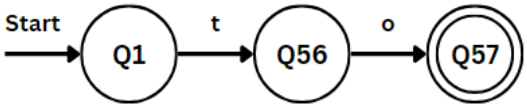
otherwise



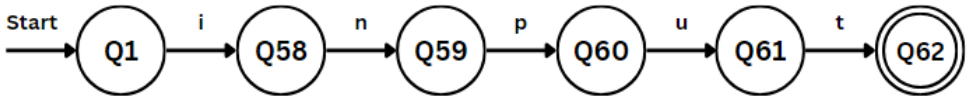
loop



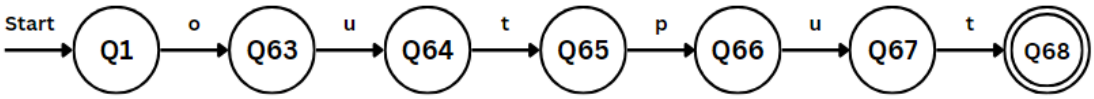
to



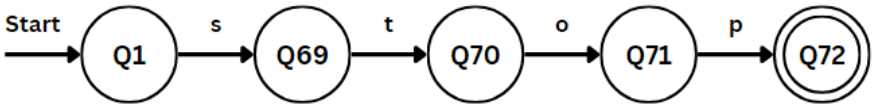
input



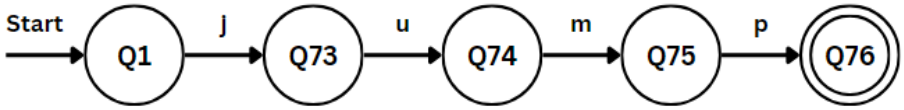
output



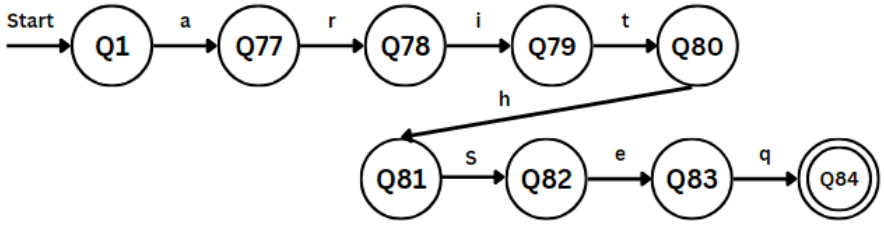
stop



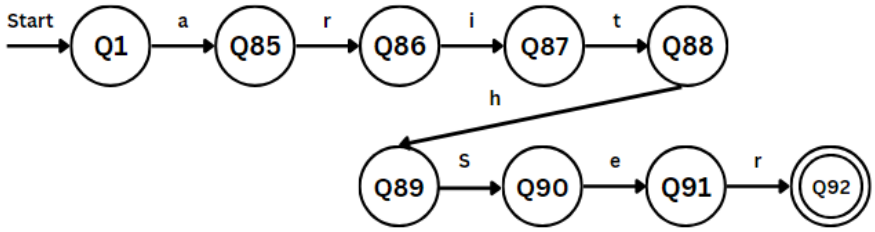
jump



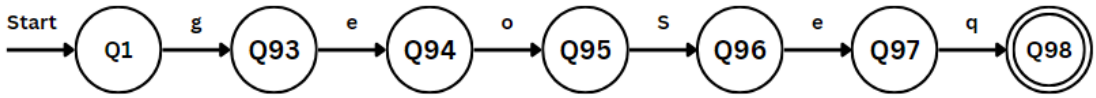
arithSeq



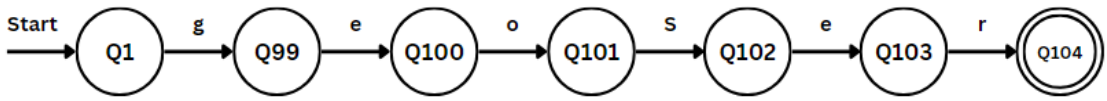
arithSer



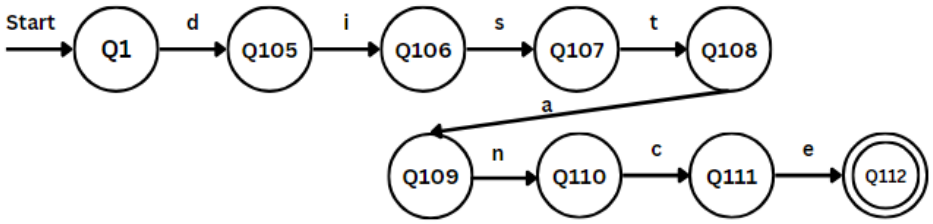
geoSeq



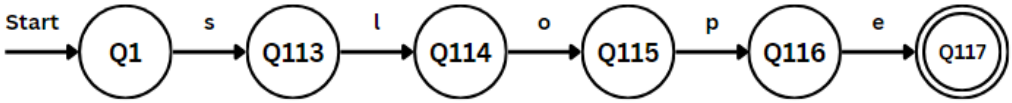
geoSer



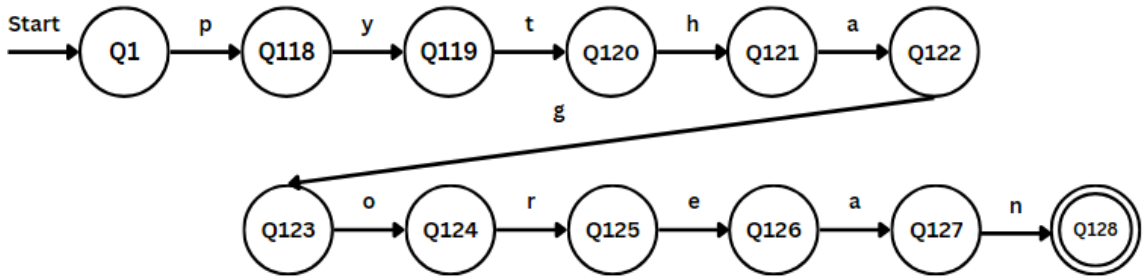
distance



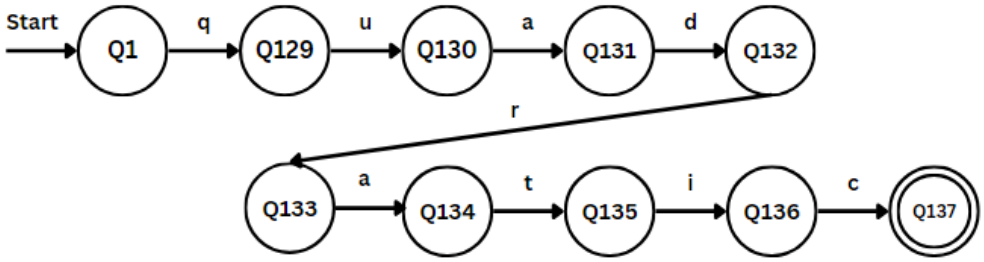
slope



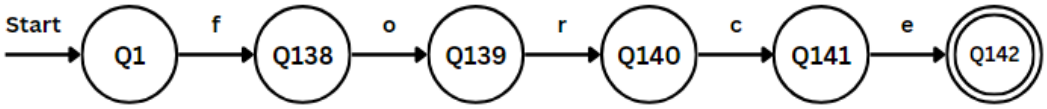
pythagorean



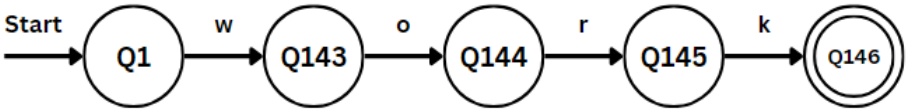
quadratic



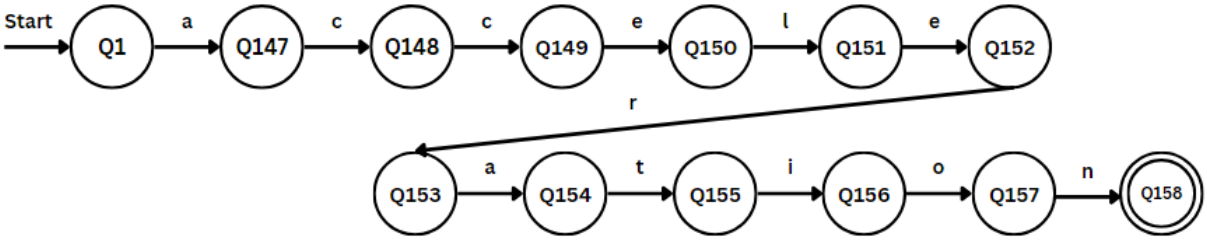
force



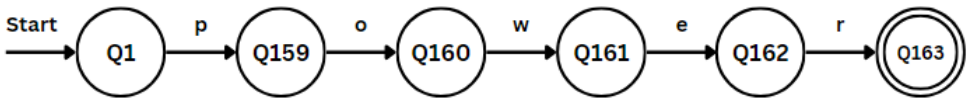
work



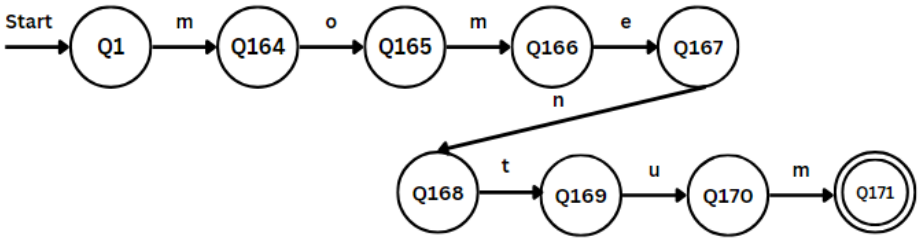
acceleration



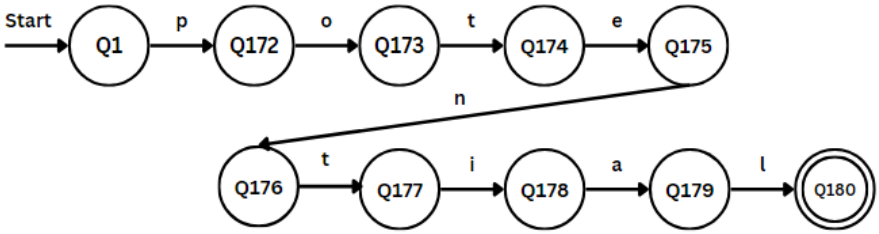
power



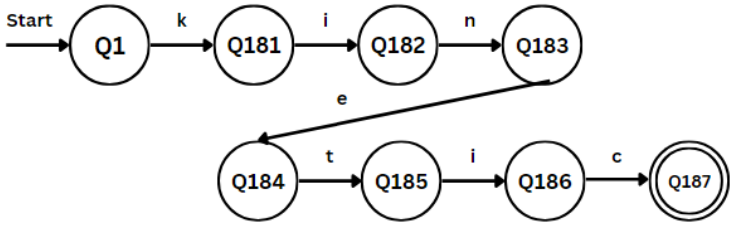
momentum



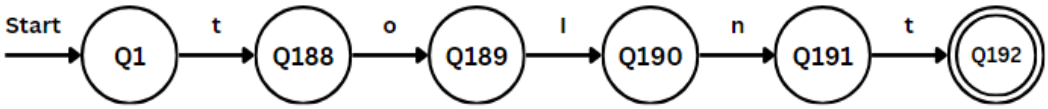
potential



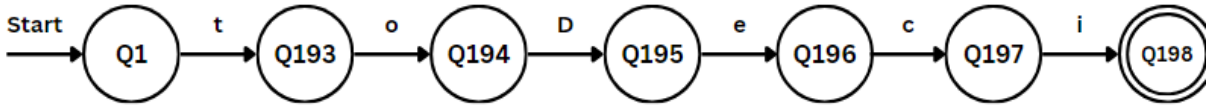
kinetic



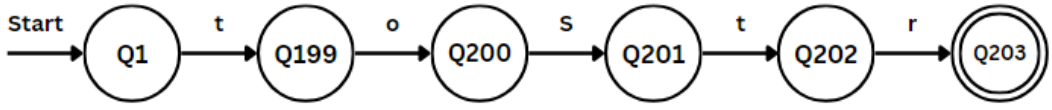
toInt



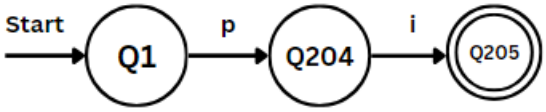
toDeci



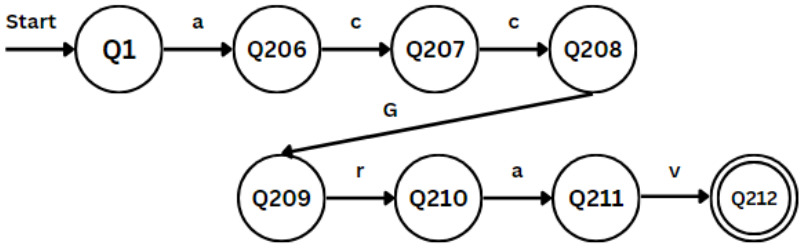
toStr



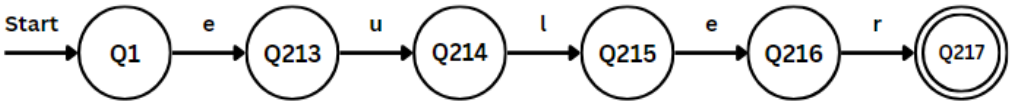
pi



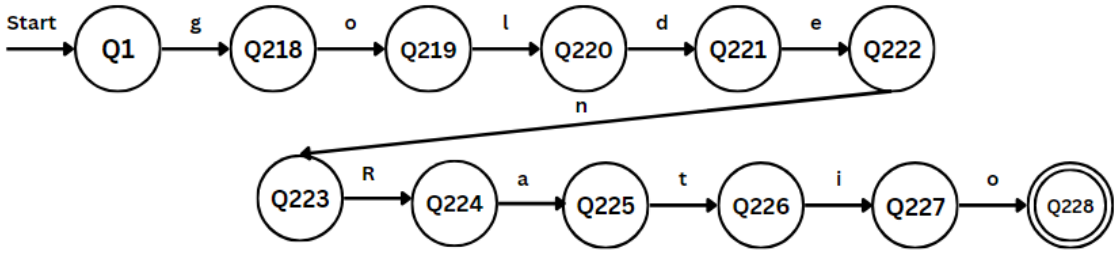
accGrav



euler

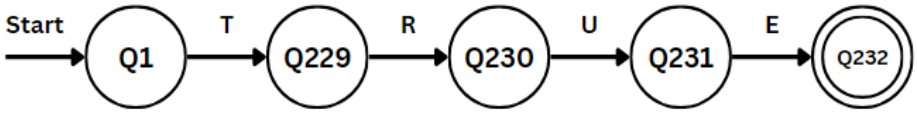


goldenRatio

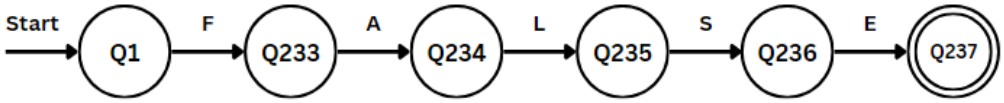


Reserved Words	Definition
TRUE (true)	a boolean value that represents truth.
FALSE (false)	a boolean value that represents false.
main (main)	entry-point of a program execution.
cont (continue)	skips the current ireation of the loop and continues with the next iteration.

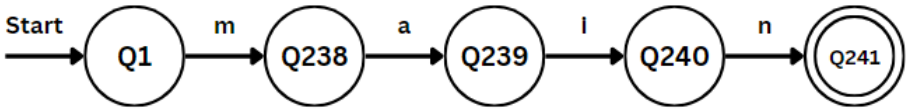
TRUE



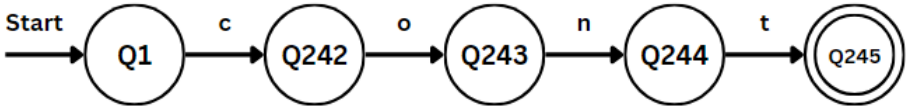
FALSE



main



cont

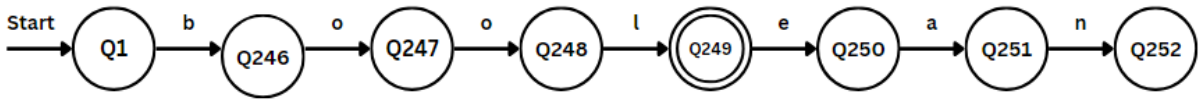


5. Noise Words

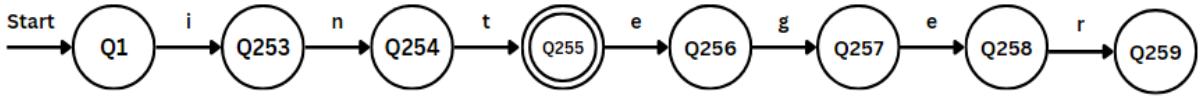
Noise Words	Shorthand	Original Notation	Definition
ean	bool	boolean	Represents the complete notation for the data type "boolean," with "ean" acting as a noise word. It is shortened to "bool" for clarity.
eger	int	integer	Represents the complete notation for the keyword "integer." The term "eger" acts as a noise word and is shortened to "int" for conciseness.
mal	deci	decimal	Represents the complete notation for the data type "decimal." The term "mal" serves as a noise word and is shortened to "deci" to simplify.

ing	str	string	Represents the complete notation for the keyword "string." The term "ing" acts as a noise word and is shortened to "str" for conciseness.
acter	char	character	Represents the complete notation for the data type "character," with "acter" acting as a noise word. It is abbreviated as "char" for conciseness.
ault	def	default	Represents the complete notation for the keyword "default." The term "ault" acts as a noise word and is shortened to "def" for conciseness.
inue	cont	continue	Represents the complete notation for the keyword "continue." The term "inue" acts as a noise word and is shortened to "cont" for conciseness.

bool



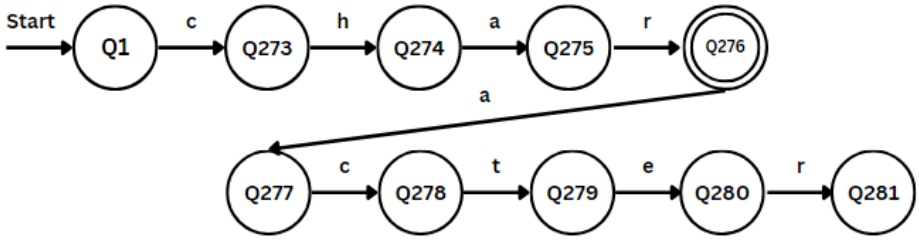
int



deci

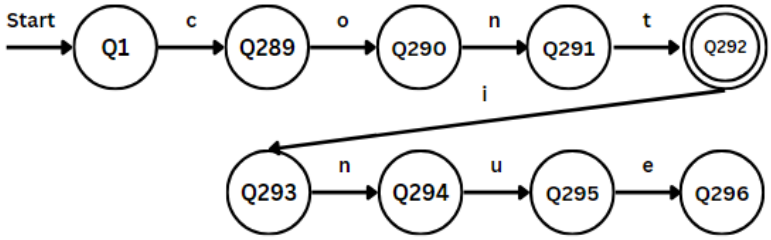
str

char



def

cont



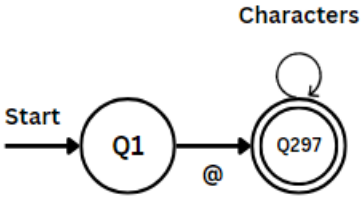
6. Comments

a. Single-line Comments

@This is a single-line comment

- A single-line comment always starts with @.
- In a single line, the statements after @ will not be executed.
- A single-line comment can be alone in one line:

@This is a single line comment
Output("Hello World!")

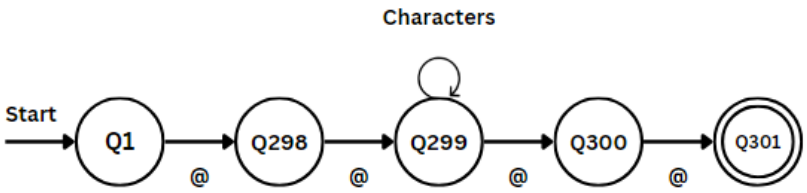


b. Multi-line Comments

@@
This is a
multi-line comment
@@

- A multi-line comment always starts with @@ and ends with @@.
- In a multi-line, any statement between @@ and @@ will not be executed.
- It will still be considered as a comment even if the statement is placed beside the start and end symbol without a space:

@@This is a
multi-line comment@@



7. Blanks

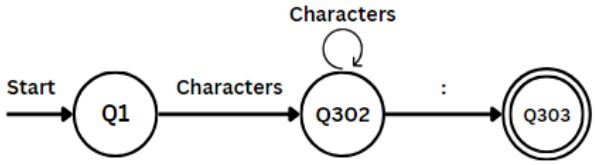
- A white space is necessary after using identifiers, operators, keywords, reserved words, noise words, and delimiters and brackets.
- FirstByte generally ignores white space or an entire blank line**.**
- Multiple variable declarations can be written in one line.

8. Delimiters and Brackets

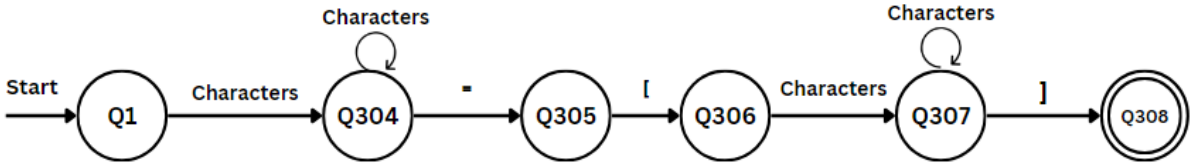
- **Colon** (:) – used to indicate the start of a new block of code, such as loop, function, or conditional statement. (e.g., def greet(name):)
- **Square Brackets** ([ ]) – used to create and access lists and dictionaries. (e.g., my_list = [ ])

- **Braces** ({ }) – used to create and access dictionaries, and to format strings. (e.g, my_dict = {“name”: “alice”, “age” : 25})
- **Parenthesis** (( )) – used to enclose expressions, functions, and methods. (e.g., toStr(number))

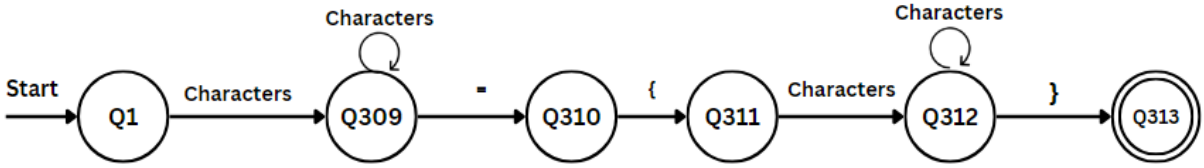
Colon



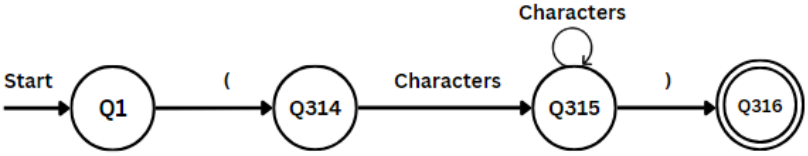
Square Brackets



Braces



Parenthesis



9. Free-and-Fixed-Field Formats

FirstByte is based on Python which is a free-field format language. This means that FirstByte is relatively flexible in terms of code formatting and indentation. Unlike languages like Java and C, which use braces or other explicit markers to define code blocks, FirstByte relies on indentation to indicate the scope and structure of the code. In this way, the language will be more flexible and can handle different types of data structures.

10. Expression

Rules for evaluation expressions:

i. Mathematical/Arithmetic Expressions

Precedence	Operator	Description	Associativity
1	* / %	Multiplication, Division, Remainder, and Bitwise	left-to-right
2	+ -	Addition and Subtraction	left-to-right

ii. Unary Expression

Precedence	Operator	Description	Associativity
1	+	Unary Plus	
2	-	Unary Minus	
3	!	Logical Not	right
4	Is	Identity Operator	
5	Is not	Identity Negation	
6	-(in set difference)	Set Difference	
7	+(string concat)	String Concatenation	

iii. Boolean Expression (Relational and Logical)

Precedence	Operator	Description	Associativity
1	!	Logical Not	right-to-left
2	< <= > >=	The relational operators include "less than," "less than	left-to-right

		or equal to," "greater than", and "greater than or equal to."	
3	== !=	The relational operators encompass "equal to" and "not equal to."	left-to-right
4	AND	Logical AND	
5	OR	Logical OR	
6	= += -= *= /= %=	Direct Assignment Assignmentt by sum and difference Assignment by product, quotient, remainder, and integer quotient.	right to left

11. Statements

- **Declaration Statements**

  data_type = {int, str, char, deci, bool}

Syntax	**Example**
<data_type><identifier>	**int** a **str** name **deci** float
<data_type><identifier>, <Assignment_Operator><value>	**int** a=5 **str** name=”andy” **deci** float =3.5

<data_type><identifier>, <identifier>, …,<identifier>	**int** a, b, c **str** name, color, gender **deci** n1, n2, n3
<data_type><identifier><Assignment_Operator><value>, <identifier><Assignment_Operator><value>,<identifier> <Assignment_Operator><value>	**int** a=5, b=3, c=10 **str** name=”andy”, color=”blue”, gender=”male” **deci** n1=2.5, n2=1.5, n3=.3

- **Input/Output Statements**

  i. Input Statements

Syntax	Example	Output
<data_type><identifier> = input	output(“Enter your age: “) int age = input() output(“Your Age: “, age)	Enter your age: 30 Your Age: 30

  ii. Output Statements

Syntax	Example	Output
output(“<statement>”)	output(“Hello World!”)	Hello World!
output(<identifier>)	int x = 12 output(x)	12
output(“<statement>”, <identifier>)	str name =”Tipen” output(“Hello”, name)	Hello! Tipen

- **Assignment Statements**

  Assignment_Operator = {=, +=, -=, *=, /=, %=}

Syntax	Definition	Example

<data_type> <identifier> <Assignment_Operator><value>	**Assignment by value** – assigns the value to the identifier.	**int** count = 2 **int** number = 3
<data_type> <identifier> <Assignment_Operator><identifier>	**Assignment by identifier –** assigns the value of the right identifier to the left identifier.	count += number
<data_type> <identifier> <Assignment_Operator><expression>	**Assignment by expression** – assigns the value of the expression to an identifier	**int** total = count + number

- **Conditional Statement**

  i. when (if)

Syntax	Example	Output
when (expression): <statement>	**int** x = 10 **int** y = 5 when (x >= y): output(“Accepted”)	Accepted

  ii. when otherwise (if else)

Syntax	Example	Output

when (expression):
  <statement>
otherwise:
  <statement>
	int num1 = 2
	when (num1 == 5):
	  output(“Hello World”)
	otherwise:
	  output(“Goodbye World”)
		Goodbye World

iii. when-otherwise when-otherwise

Syntax	Example	Output

when (expression):
  <statement>
otherwise when (expression):
  <statement>
otherwise:
  <statement>
	int x = -5
	when (x<0):
	  output("x is a negative num")
	otherwise when (x>0):
	  output("x is a positive num")
	otherwise:
	  output("invalid number")
		x is a negative num

iv. nested when (nested if)

Syntax	Example	Output

when (condition 1):
  when (condition 2):
  <statement>
  otherwise:
  <statement>
otherwise:
  <statement>
	deci grade = 2.25
	when (grade <= 3.0):
	  when (grade == 1.00):
	  output("excellent!:)
	  otherwise:
	  output("passed!")
	otherwise:
	  output("failed")
		x is a negative num

v. nested when otherwise (nested if else)

Syntax	Example	Output
when (condition 1): when (condition 2): <statement> otherwise: <statement> otherwise: when (condition 3): <statement> otherwise: <statement>	**deci** gwa = 1.15 when (gwa >= 1.0000 AND gwa <= 1.3500): when (gwa <= 1.15): output("Summa Cum Laude") otherwise: output("Magna Cum Laude") otherwise: when (gwa >= 1.3501 AND gwa <= 1.6000): output("Cum Laude") otherwise: output("Congrats")	Summa Cum Laude

vi. nested when otherwise when (nested if else if)

Syntax	Example	Output
when (condition 1): when (condition 2): <statement> otherwise when(condition 3): <statement> otherwise: <statement> otherwise when (condition 4): when (condition 5): <statement> otherwise: <statement> otherwise: <statement>	int n1 = 7 int n2 = 3 int n3 = 10 when (n1 <= n2): when (n2 <= n3): output("Asc order: n1, n2, n3") otherwise when (n1 <= n3): output("Asc order: n1, n3, n2") otherwise: output("Asc order: n2, n1, n3")| otherwise when (n2 <= n3): when (n1 <= n3): output("Asc order: n2, n1, n3") otherwise: output("Asc order: n2, n3, n1") otherwise: output("Asc order: n3, n2, n1")	Ascending order: n2, n1, n3

- **Iterative Statements**

  <initialization> ::= <identifier> “=” <value>

  <range> ::= “to” <value>

  i. Loop

Syntax	Example	Output
loop <initialization> <range> (incrementation) <statement>	int i loop i = 0 to 4 (i++): output(i)	0 1 2 3

  ii. Nested Loop

Syntax	Example	Output

loop <initialization> <range>(incrementation)
    <statement>
    loop
<initialization><range>(incrementation)
        <statement>
	int i,j
	loop i=0 to 3 (i++):
	    output(i)
	    loop j=0 to 2 (j++):
	        output(j)
		0
		0
		1
		1
		0
		1
		2
		0
		1

Chapter 2: Syntax Analysis

Syntactic Elements of Language

Declaration Statement

Production Rule:

<DEC_STATEMENT> =

<DATA_TYPE><WHITESPACE><IDENTIFIER>

<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

<DATA_TYPE><WHITESPACE><IDENTIFIER>,<WHITESPACE> <IDENTIFIER>, <WHITESPACE><IDENTIFIER> ……<WHITESPACE> <IDENTIFIER>

<DATA_TYPE><IDENTIFIER><ASSIGNMENT_OPERATOR><VALUE>, <IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE> , <IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

<DATA_TYPE> = int | deci | str | char | bool

<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _<DIGIT> ::= 0 | <NONZERO>

<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ALPHABET> ::= 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 | 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

Example:

Leftmost Derivation

ex: **int var**

<DEC_STATEMENT> ::= <DATA_TYPE><WHITESPACE><IDENTIFIER>

::= int<WHITESPACE><IDENTIFIER>

::= int <IDENTIFIER>

::= int <ALPHABET><ALPHABET><ALPHABET>

::= int v,<ALPHABET><ALPHABET>

::= int va,<ALPHABET>

::= int var

Rightmost Derivation

ex: **int var**

<DEC_STATEMENT> ::= <DATA_TYPE><WHITESPACE><IDENTIFIER>

::= <DATA_TYPE><WHITESPACE><ALPHABET><ALPHABET><ALPHABET>

::= <DATA_TYPE><WHITESPACE><ALPHABET><ALPHABET>,r

::= <DATA_TYPE><WHITESPACE><ALPHABET>,ar

::= <DATA_TYPE><WHITESPACE>var

::= <DATA_TYPE> var

::= int var

Input /Output Statement

Production Rule:

<**DATA_TYPE**> ::= int

<**WHITESPACE**> ::= ' '

<**IDENTIFIER**> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

<**IDENTIFIER_CHARS**> ::= <ALPHABET> | <DIGIT> | _

<**ALPHABET**> ::= 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 | 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

<**DIGIT**> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+

<**NONZERO**> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<**EQUAL**> ::= =

<**KEYWORD**> ::= input | output

<**DELIMITER**> ::= (|)

<**STRING**> ::= <SPECIAL_CHARS><STRING_CHARS>*<SPECIAL_CHARS>

<**STRING_CHARS**> ::= <ALPHABET> | <DIGIT> | <SPECIAL_CHARS>

<**SPECIAL_CHARS**> ::= ! | @ | # | $ | % | ^ | & | * | (|) | _ | - | + | = | [|] | { | } | | | \ | ; | : | , | < | > | . | / | ? | ~ | “

Example: Input Statement

Leftmost Derivation

ex: **int x = input()**

<INPUT_STATEMENT> ::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><KEYWORD><DELIMITER><DELIMITER>

::=
int<WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><KEYWORD><DELIMITER><DELIMITER>

::= int
<IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><KEYWORD><DELIMITER>
<DELIMITER>

::= int
<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><EQUAL><WHITESPACE><K
EYWORD><DELIMITER><DELIMITER>

::= int
x<WHITESPACE><EQUAL><WHITESPACE><KEYWORD><DELIMITER><DELIMITER
>

::= int x <EQUAL><WHITESPACE><KEYWORD><DELIMITER><DELIMITER>

::= int x =<WHITESPACE><KEYWORD><DELIMITER><DELIMITER>

::= int x = <KEYWORD><DELIMITER><DELIMITER>

::= int x = input<DELIMITER><DELIMITER>

::= int x = input(<DELIMITER>

::= int x = input()

Rightmost Derivation

ex: **int x = input()**

<INPUT_STATEMENT> ::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPAC
E><KEYWORD><DELIMITER><DELIMITER>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPAC
E><KEYWORD><DELIMITER>)

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPAC
E><KEYWORD>()

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPAC
E>input()

::= <DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL> input()

::= <DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE>= input()

::= <DATA_TYPE><WHITESPACE><IDENTIFIER> = input()

::= <DATA_TYPE><WHITESPACE><ALPHABET>+<IDENTIFIER_CHARS>* = input()

::= <DATA_TYPE><WHITESPACE>x = input()

::= <DATA_TYPE> x = input()

::= int x = input()

Example: Output Statement

Leftmost Derivation

ex: output("xd")

<OUTPUT_STATEMENT> ::= <KEYWORD><DELIMITER><STRING><DELIMITER>

::= output<DELIMITER><STRING><DELIMITER>

::= output(<STRING><DELIMITER>

::= output(<SPECIAL_CHARS><STRING_CHARS>*<SPECIAL_CHARS><DELIMITER>

::= output(“<STRING_CHARS>*<SPECIAL_CHARS><DELIMITER>

::= output(“<ALPHABET><SPECIAL_CHARS><DELIMITER>

::= output(“x<STRING_CHARS>*<SPECIAL_CHARS><DELIMITER>

::= output(“x<ALPHABET><SPECIAL_CHARS><DELIMITER>

::= output(“xd<SPECIAL_CHARS><DELIMITER>

::= output(“xd”<DELIMITER>

::= output(“xd”)

Rightmost Derivation

ex: **output("xd")**

<OUTPUT_STATEMENT> ::= <KEYWORD><DELIMITER><STRING><DELIMITER>

::= <KEYWORD><DELIMITER><STRING>)

::=
<KEYWORD><DELIMITER><SPECIAL_CHARS><STRING_CHARS>*<SPECIAL_CHAR
S>)

::= <KEYWORD><DELIMITER><SPECIAL_CHARS><STRING_CHARS>*")

::= <KEYWORD><DELIMITER><SPECIAL_CHARS><ALPHABET>")

::= <KEYWORD><DELIMITER><SPECIAL_CHARS><STRING_CHARS>*d")

::= <KEYWORD><DELIMITER><SPECIAL_CHARS><ALPHABET>*d")

::= <KEYWORD><DELIMITER><SPECIAL_CHARS>xd")

::= <KEYWORD><DELIMITER>"xd")

::= <KEYWORD>("xd")

::= output("xd")

Assignment Statement

Production Rule:

<DATA_TYPE> = int | deci | str | char | bool

<VALUE> ::= <INTEGER> | <DECIMAL> | <STRING> | <DATA_CHARACTER>

<WHITESPACE> ::= ' '

<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _

<ALPHABET> ::= 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 | 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

<DIGIT> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+

<NONZERO> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ASSIGNMENT_OP> ::= = <ASSIGNMENT_OPERATOR> ::=
<DATA_TYPE><WHITESPACE>
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

Example:

Leftmost Derivation

ex**: int num1 = 10**

<ASSIGNMENT_OPERATOR> ::= <DATA_TYPE><WHITESPACE>
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= <DATA_TYPE><WHITESPACE>
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int<WHITESPACE>
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int
<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><ASSIGNMENT_OPERATOR>
<WHITESPACE><VALUE>

::= int
n<Alphabet>*<WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int nu, <WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int
nu,<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><ASSIGNMENT_OPERATO
R><WHITESPACE><VALUE>

::= int
num,<IDENTIFIER_CHARS>*<WHITESPACE><ASSIGNMENT_OPERATOR><WHITESP
ACE><VALUE>

::= int
num,<DIGIT><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int
num,<NONZERO><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VAL
UE>

::= int num1<WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int num1<WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int num1 <ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::= int num1 =<WHITESPACE><VALUE>

::= int num1 = <VALUE>

::= int num1 = <INTEGER>

::= int num1 = <NONZERO><DIGIT>*

::= int num1 = 10

Rightmost Derivation

ex**: int num1 = 10**

<ASSIGNMENT_OPERATOR> ::= <DATA_TYPE><WHITESPACE>
<IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><VALUE>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERA
TOR><WHITESPACE><VALUE>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERA
TOR><WHITESPACE><INTEGER>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE><NONZERO><DIGIT>*

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR><WHITESPACE>10

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><ASSIGNMENT_OPERATOR> 10

::= <DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE>= 10

::= <DATA_TYPE><WHITESPACE><IDENTIFIER> = 10

::= <DATA_TYPE><WHITESPACE><ALPHABET>+<IDENTIFIER_CHARS>* = 10

::= <DATA_TYPE><WHITESPACE>n,<Alphabet>* = 10

::= <DATA_TYPE><WHITESPACE>nu, = 10

::= <DATA_TYPE><WHITESPACE>nu,<ALPHABET>+<IDENTIFIER_CHARS>* = 10

::= <DATA_TYPE><WHITESPACE>num,<DIGIT>* = 10

::= <DATA_TYPE><WHITESPACE>num,<NONZERO> = 10

::= <DATA_TYPE><WHITESPACE>num1 = 10

::= <DATA_TYPE> num1 = 10

::= int num1 = 10

Arithmetic Operator

Production Rule:

<ARITHMETIC_OPERATOR> = + | - | * | / | % | ^

<VALUE> ::= <INTEGER> | <DECIMAL> | <STRING> | <DATA_CHARACTER>

<INTEGER> ::= <NONZERO><DIGIT>*

<DIGIT> ::= 0 | <NONZERO>

<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Example:

Leftmost Derivation

ex**: 5 + 3 – 2**

<ARITHMETIC_OPERATOR> ::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::=
<INTEGER><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::=
<NONZERO><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::=
5<WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5
<ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5
+<WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 +
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 +
<INTEGER><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 +
<NONZERO><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 + 3<WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 + 3 <ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::= 5 + 3 -<WHITESPACE><VALUE>

::= 5 + 3 - <VALUE>

::= 5 + 3 - <INTEGER>

::= 5 + 3 - <NONZERO>

::= 5 + 3 - 2

Rightmost Derivation

ex**: <u>5 + 3 – 2</u>**

<ARITHMETIC_OPERATOR>::=<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE>

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><INTEGER>

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><NONZERO>

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE>2

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE><ARITHMETIC_OPERATOR> 2

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE><WHITESPACE>- 2

::= <VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><VALUE> - 2

::= <VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><INTEGER>
- 2

::=
<VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE><NONZERO> -
2

::= <VALUE><WHITESPACE><ARITHMETIC_OPERATOR><WHITESPACE>3 - 2

::= <VALUE><WHITESPACE><ARITHMETIC_OPERATOR> 3 - 2

::= <VALUE><WHITESPACE>+ 3 - 2

::= <VALUE> + 3 - 2

::= <INTEGER> + 3 - 2

::= <NONZERO> + 3 - 2

::= 5 + 3 - 2

Unary Operator

Production Rule:

<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUALS><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

<UNARY_OPERATOR_VALUE> ::= + | -

<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

Example:

Leftmost Derivation

ex: **int x = - 2**

<UNARY_OPERATOR> ::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::=
int<WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int
<IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int
<ALPHABET><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int
x<WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int x
<EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int x =<WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int x = <UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::= int x = -<WHITESPACE><VALUE>

::= int x = - <VALUE>

::= int x = - <INTEGER>

::= int x = - <NONZERO>

::= int x = - 2

Rightmost Derivation

ex: **int x = - 2**

<UNARY_OPERATOR> ::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><VALUE>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><INTEGER>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE><NONZERO>

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE><WHITESPACE>2

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE><UNARY_OPERATOR_VALUE> 2

::=
<DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL><WHITESPACE>- 2

::= <DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE><EQUAL> - 2

::= <DATA_TYPE><WHITESPACE><IDENTIFIER><WHITESPACE>= - 2

::= <DATA_TYPE><WHITESPACE><IDENTIFIER> = - 2

::= <DATA_TYPE><WHITESPACE><ALPHABET> = - 2

::= <DATA_TYPE><WHITESPACE>x = - 2

::= <DATA_TYPE> x = - 2

::= int x = - 2

Boolean Logic

Production Rule:

<BOOLEAN_LOGIC> = LOGICAL_NOT | LOGICAL_OR | LOGICAL_AND

<BOOLEAN_RELATION> = EQUAL_TO | NOT_EQUAL | GREATER_THAN | LESS_THAN | GREATER_THAN_EQUAL | LESS_THAN_EQUAL

<VALUE> ::= <INTEGER> | <DECIMAL> | <STRING> | <DATA_CHARACTER>

<WHITESPACE> ::= ' '

<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _

<ALPHABET> ::= 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 | 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

<DIGIT> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+

<NONZERO> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ASSIGNMENT_OP> ::= =

Example:

Leftmost Derivation

ex: x < 5 || x > 10

<BOOLEAN_LOGIC> ::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::=
<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::=
x,<WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><WHITESPACE><BOOLEAN_LOGIC><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x
<BOOLEAN_RELATION><WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x
<,<WHITESPACE><VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x <
<VALUE><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x <
<INTEGER><WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x <
<NONZERO><DIGIT>*<WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x <
5,<WHITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5
<BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5
||<WHITESPACE><IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5 ||
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5 ||
<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5 || x,<WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5 || x <BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x < 5 || x >,<WHITESPACE><VALUE>

::= x < 5 || x > <VALUE>

::= x < 5 || x > <INTEGER>

::= x < 5 || x > <NONZERO><DIGIT>*

::= x < 5 || x > 10

Rightmost Derivation

ex: x < 5 || x > 10

<BOOLEAN_LOGIC> ::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BO
OLEAN_RELATION><WHITESPACE><VALUE>

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BO
OLEAN_RELATION><WHITESPACE><VALUE>

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BO
OLEAN_RELATION><WHITESPACE><NONZERO><DIGIT>*

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BO
OLEAN_RELATION><WHITESPACE>,10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE><BO
OLEAN_RELATION> 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER><WHITESPACE>,> 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><IDENTIFIER> > 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE><ALPHABET>+<IDENTIFIER_CHAR
S>* > 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC><WHITESPACE>,x > 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE><BOOLEAN_LOGIC> x > 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE><W
HITESPACE>|| x > 10

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>
|| x > 10

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><NONZERO>
<DIGIT>* || x > 10

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE>,5 || x > 10

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION> 5 || x > 10

::= <IDENTIFIER><WHITESPACE>,< 5 || x > 10

::= <IDENTIFIER> < 5 || x > 10

::= <ALPHABET>+<IDENTIFIER_CHARS>* < 5 || x > 10

::= x < 5 || x > 10

Boolean Relation

Production Rule:

<BOOLEAN_RELATION> ::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

<INTEGER> ::= <NONZERO><DIGIT>*

<DIGIT> ::= 0 | <NONZERO>

<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<WHITESPACE> ::= ' '

<BOOLEAN_RELATION> ::= not_equal

<VALUE> ::= <INTEGER>

Example:

Leftmost Derivation

Ex**. x != 5**

<BOOLEAN_RELATION> ::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::=<ALPHABET>+<IDENTIFIER_CHARS>*<WHITESPACE><BOOLEAN_RELATION><
WHITESPACE><VALUE>

::= x,<WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x <BOOLEAN_RELATION><WHITESPACE><VALUE>

::= x !=<WHITESPACE><VALUE>

::= x != <VALUE>

::= x != <INTEGER>

::= x != <DIGIT>

::= x != <NONZERO>

::= x != 5

Rightmost Derivation

Ex**. x != 5**

<BOOLEAN_RELATION> ::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><VALUE>

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><INTEGER>

::=
<IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE><NONZERO>

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION><WHITESPACE>,5

::= <IDENTIFIER><WHITESPACE><BOOLEAN_RELATION> 5

::= <IDENTIFIER><WHITESPACE>!= 5

::= <IDENTIFIER> = 5

::= <ALPHABET>+<IDENTIFIER_CHARS>*!= 5

::= x != 5

Conditional Statement

Production Rules

<KEYWORD> ::= when | output
<WHITESPACE> ::= ' '
<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+
<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _
<ALPHABET> ::= 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 | 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
<DIGIT> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+
<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<DELIMITER> ::= (|) | :
<**RELATIONAL_OP**> ::= >
<NEWLINE> ::=
<INDENT> ::= \t
<**OUTPUT_STMT**> ::= <KEYWORD> <DELIMITER> <STRING> <DELIMITER>
<STRING> ::= <SPECIAL_CHARS><STRING_CHARS>*<SPECIAL_CHARS>
<STRING_CHARS> ::= <ALPHABET> | <DIGIT> | <SPECIAL_CHARS>
<SPECIAL_CHARS> ::= "

Example:

Leftmost Derivation
Ex. **when (x > 5):**
output(“hi”)

<COND_STMT> ::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when<WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (<IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (<ALPHABET>+<IDENTIFIER_CHARS>* <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x<WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT>
<DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER>
<DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x ><WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE>
<INDENT> <OUTPUT_STMT>

::= when (x > <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT>
<OUTPUT_STMT>

::= when (x > <NONZERO> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT>
<OUTPUT_STMT>

::= when (x > 5<DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x > 5)<DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x > 5):<NEWLINE> <INDENT> <OUTPUT_STMT>

::= when (x > 5):
<INDENT> <OUTPUT_STMT>

::= when (x > 5):
    <OUTPUT_STMT>

::= when (x > 5):
    <KEYWORD> <DELIMITER> <STRING> <DELIMITER>

::= when (x > 5):
    output<DELIMITER> <STRING> <DELIMITER>

::= when (x > 5):
    output(<STRING> <DELIMITER>

::= when (x > 5):
    output(<SPECIAL_CHARS> <STRING_CHARS>* <SPECIAL_CHARS>
<DELIMITER>

::= when (x > 5):
    output(“<STRING_CHARS>* <SPECIAL_CHARS> <DELIMITER>

::= when (x > 5):
output(“<ALPHABET> <SPECIAL_CHARS> <DELIMITER>

::= when (x > 5):
output(“h<STRING_CHARS>* <SPECIAL_CHARS> <DELIMITER>

::= when (x > 5):
output(“h<ALPHABET> <SPECIAL_CHARS> <DELIMITER>

::= when (x > 5):
output(“hi<SPECIAL_CHARS> <DELIMITER>

::= when (x > 5):
output(“hi”<DELIMITER>

::= when (x > 5):
output(“hi”)

Rightmost Derivation
Ex. **when (x > 5):**
output(“hi”)

<COND_STMT> ::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <STRING> <DELIMITER>

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <STRING>)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <SPECIAL_CHARS> <STRING_CHARS>* <SPECIAL_CHARS>)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE> <RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <SPECIAL_CHARS> <STRING_CHARS>*”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <SPECIAL_CHARS>
<ALPHABET>")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <SPECIAL_CHARS>
<STRING_CHARS>*i")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD>
<DELIMITER><SPECIAL_CHARS><ALPHABET>i")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD> <DELIMITER><SPECIAL_CHARS>hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD> <DELIMITER>"hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT> <KEYWORD>("hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> <INDENT>output("hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
<NEWLINE> output("hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER> <DELIMITER>
output("hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT> <DELIMITER>:
output("hi")

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <DIGIT>):

```
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE> <NONZERO>):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> <WHITESPACE>5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>
<RELATIONAL_OP> 5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> <WHITESPACE>> 5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER> <IDENTIFIER> > 5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER>
<ALPHABET>+<IDENTIFIER_CHARS>* > 5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE> <DELIMITER>x > 5):
        output(”hi”)

::= <KEYWORD> <WHITESPACE>(x > 5):
        output(”hi”)

::= <KEYWORD> (x > 5):
        output(”hi”)

::= when (x > 5):
        output(”hi”)
```

Iterative Statement

Production Rule:

<KEYWORD> ::= loop | to | output

<WHITESPACE> ::= ' '

<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+

<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _

<ALPHABET> ::= 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 | 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

<DIGIT> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+

<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ASSIGNMENT_OP> ::= =

<DELIMITER> ::= (|) | :

<UNARY> ::= ++

<NEWLINE> ::= \n

<INDENT> ::= \t

<OUTPUT_STMT> ::= <KEYWORD> <DELIMITER> <IDENTIFIER> <DELIMITER>

Example:

Leftmost Derivation
Example:

loop i = 0 to 4 (i++):
output(i)

<ITERATIVE_STMT> ::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop<WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop <ALPHABET>+<IDENTIFIER_CHARS>* <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i<WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i =<WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0<WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to <NONZERO> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4<WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY>
<DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER>
<DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 (<IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE>
<INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 (<ALPHABET>+<IDENTIFIER_CHARS>* <UNARY> <DELIMITER>
<DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 (i<UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT>
<OUTPUT_STMT>

::= loop i = 0 to 4 (i++<DELIMITER> <DELIMITER> <NEWLINE> <INDENT>
<OUTPUT_STMT>

::= loop i = 0 to 4 (i++)<DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>
::= loop i = 0 to 4 (i++):<NEWLINE> <INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 (i++):<INDENT> <OUTPUT_STMT>

::= loop i = 0 to 4 (i++):
    <OUTPUT_STMT>

::= loop i = 0 to 4 (i++):
    <KEYWORD> <DELIMITER> <IDENTIFIER> <DELIMITER>

::= loop i = 0 to 4 (i++):
    output<DELIMITER> <IDENTIFIER> <DELIMITER>

::= loop i = 0 to 4 (i++):
    output(<IDENTIFIER> <DELIMITER>

::= loop i = 0 to 4 (i++):
    output(<ALPHABET>+<IDENTIFIER_CHARS>* <DELIMITER>

::= loop i = 0 to 4 (i++):
    output(i<DELIMITER>

::= loop i = 0 to 4 (i++):
    output(i)

Rightmost Derivation

Example:

loop i = 0 to 4 (i++):
    output(i)

<ITERATIVE_STMT> ::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <OUTPUT_STMT>

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <IDENTIFIER> <DELIMITER>

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <IDENTIFIER>)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER> <ALPHABET>+<IDENTIFIER_CHARS>*)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD> <DELIMITER>i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT> <KEYWORD>(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> <WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY> <DELIMITER> <DELIMITER> <NEWLINE> <INDENT>output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE> <ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>

<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY>
<DELIMITER> <DELIMITER> <NEWLINE> output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY>
<DELIMITER> <DELIMITER>
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY>
<DELIMITER>:
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER> <UNARY>):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER> <IDENTIFIER>++):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER>
<ALPHABET>+<IDENTIFIER_CHARS>*++):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE> <DELIMITER>i++):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> <WHITESPACE>(i++):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> (i++):
    output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <DIGIT> (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE> <NONZERO> (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD>
<WHITESPACE>4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> <KEYWORD> 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> <WHITESPACE> to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE> <DIGIT> to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> <WHITESPACE>0 to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>
<ASSIGNMENT_OP> 0 to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> <WHITESPACE>= 0 to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <IDENTIFIER> = 0 to 4 (i++):
	output(i)

::= <KEYWORD> <WHITESPACE> <ALPHABET>+<IDENTIFIER_CHARS>* = 0 to 4
(i++):
	output(i)

::= <KEYWORD> <WHITESPACE>i = 0 to 4 (i++):

```
        output(i)

::= <KEYWORD> i = 0 to 4 (i++):
        output(i)

::= loop i = 0 to 4 (i++):
        output(i)
```

NEW PRINCIPLES

Production Rule:

<FUNCTION_STATEMENTS> ::= <ARITHSEQ >| <ARITHSER> | <GEOSEQ> | <GEOSER> | <PYTHAGOREAN> | <QUADRATIC> | <FORCE> | <WORK> | <POWER> | <MOMENTUM> | <POTENTIAL> | <KINETIC> | <TOINT> | <TODECI> | <TOSTR> | <SLOPE> | <DISTANCE> |<ACCELERATION>

<ARITHSEQ> ::=
arithSeq(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
arithSeq(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<ARITHSER> ::=
arithSer(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
arithSer(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<GEOSEQ> ::=
geoSeq(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
geoSeq(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<GEOSER> ::= geoSer(<DIGIT>+,<WHITESPACE> <DIGIT>+,
<WHITESPACE><DIGIT>+) |
geoSer(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<DISTANCE> ::=
distance(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
distance(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<SLOPE> ::= slope(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
slope(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<PYTHAGOREAN> ::= pythagorean(<DIGIT>+,<WHITESPACE><DIGIT>+) |
pythagorean(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<QUADRATIC>::=
quadratic(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
quadratic(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<FORCE> ::= force(<DIGIT>+,<WHITESPACE><DIGIT>+) |
force(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<WORK> ::= work(<DIGIT>+,<WHITESPACE><DIGIT>+) |
work(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<ACCELERATION> ::=
acceleration(<DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+,<WHITESPACE><DIGIT>+) |
acceleration(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<POWER> ::= power(<DIGIT>+,<WHITESPACE><DIGIT>+) |
power(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<MOMENTUM> ::= momentum(<DIGIT>+,<WHITESPACE><DIGIT>+) |
momentum(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<POTENTIAL> ::= potential(<DIGIT>+,<WHITESPACE><DIGIT>+) |
potential(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<KINETIC> ::= kinetic(<DIGIT>+,<WHITESPACE><DIGIT>+) |
kinetic(<IDENTIFIER>,<WHITESPACE><IDENTIFIER>)

<TOINT> ::= toInt(<DIGIT><WHITESPACE>+<WHITESPACE><DIGIT>+) |
toInt(<DIGIT>) | toInt(<IDENTIFIER>)

<TODECI> ::= toDeci(<DIGIT>+) | toDeci(<IDENTIFIER>)

<TOSTR> ::= toStr(<DIGIT><WHITESPACE>+<WHITESPACE><DIGIT>+) |
|toStr(<DIGIT>+) | toStr(<IDENTIFIER>)

<DIGIT> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<IDENTIFIER> ::= <ALPHABET>+<IDENTIFIER_CHARS>* | _<IDENTIFIER_CHARS>+
<IDENTIFIER_CHARS> ::= <ALPHABET> | <DIGIT> | _
<ALPHABET> ::= 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 |
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

NEW PRINCIPLE: FORCE

Production Rule:

<FORCE> ::= force(<DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT>)
<KEYWORD> ::= force
<DELIMITER> ::= (|)
<SPECIAL_CHARS> ::= ,
<DIGIT> ::= 0 | <NONZERO> | <NONZERO><DIGIT>+
<NONZERO> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<WHITESPACE> ::= ' '

Example:

Leftmost Derivation

force(8, 10)
<FORCE> ::= force(<DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT>)

::=
<KEYWORD><DELIMITER>><DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>
::=
force<DELIMITER><DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>
::= force(<DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>
::= force(<NONZERO><SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>
::= force(8<SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>
::= force(8,<WHITESPACE><DIGIT><DELIMITER>
::= force(8, <DIGIT><DELIMITER>
::= force(8, <NONZERO><DIGIT>+<DELIMITER>
::= force(8, 10<DELIMITER>
::= force(8, 10)

Leftmost Derivation
Example:
force(8, 10)

<FORCE> ::= force(<DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT>)

::=
<KEYWORD><DELIMITER><DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT><DELIMITER>

::= <KEYWORD><DELIMITER><DIGIT><SPECIAL_CHARS><WHITESPACE><DIGIT>)
::=
<KEYWORD><DELIMITER><DIGIT><SPECIAL_CHARS><WHITESPACE><NONZERO>
<DIGIT>+)
::= <KEYWORD><DELIMITER><DIGIT><SPECIAL_CHARS><WHITESPACE>10)
::= <KEYWORD><DELIMITER><DIGIT><SPECIAL_CHARS> 10)
::= <KEYWORD><DELIMITER><DIGIT>, 10)
:= <KEYWORD><DELIMITER><NONZERO>, 10)
::= <KEYWORD><DELIMITER>8, 10)
::= <KEYWORD>(8, 10)
::= force(8, 10)