Pamantasan ng Lungsod ng Maynila
University of the City of Manila
Intramuros, Manila Philippines

C-Grass PLUS

A Compiler Presented to the
Computer Science Department
College of Information System Technology and Management

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science in Computer Science (BSCS)

Submitted by:
TEAM HERBICODE

CANDO, Jhaime Jose O.
CALADIAO, Jerome Z.
GUMAWID, Reuel Augustus A.
LIM, Lance Daniel P.
TENIO, Jonald R.

Submitted to:
PROF. LEISYL M. MAHUSAY
PROF. JAMILLAH S. GUIALIL

June 2024

PAMANTASAN NG LUNGSOD NG MAYNILA

College of Information System Technology and Management

Bachelor of Science in Computer Science (BSCS)

S.Y. 2023-2024

CSC 0322-1 - COMPILER DESIGN

C-Grass PLUS

Group Leader:
Cando, Jhaime José O.

Group Members:

Caladiao, Jerome Z.

Gumawid, Reuel Augustus A.

Lim, Lance Daniel P.

Tenio, Jonald R.

Submitted by:
HerbiCode

Submitted to:
Prof. Leisyl M. Mahusay
Prof. Jamillah S. Guialil

Date: June 13, 2024

PAMANTASAN NG LUNGSOD NG MAYNILA

(University of the City of Manila)

Intramuros, Manila

College of Information System and Technology Management

Computer Science Department

APPROVAL SHEET

The proposed compiler entitled **"C – Grass PLUS"** presented and submitted by **HerbiCode** is hereby approved and accepted as partial fulfillment in CSC 0322 - Compiler Design for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance and approval for **FINAL DEFENSE**.

Leisyl M. Mahusay, MEM/SM
Adviser

Jamillah S. Guialil
Adviser

Accepted and approved in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

Raymund M. Dioses, MAEd
Chairperson, CS Department

Khatalyn E. Mata, DIT
Dean, CISTM

ACKNOWLEDGEMENT

We would like to pledge our gratitude to our professors, Professor Leisyl M. Mahusay, for teaching and guiding us to continue pushing through making all the progression first step to the other. Thank you for being our professor in (1) Automata Theory and Formal Languages, and (2) Compiler Design, as well as to Professor Jamillah S. Guialil for being part of our course in Compiler Design clearing out various things for the totality of the course. Additionally, Professor Jamillah S. Guialil handled us as one of the panelists with Sir Gabriel Hill for the defense in Automata Theory and Formal Languages. Handled us through a year of learning and mustering all the knowledge garnered studying Computer Science in designing a compiler.

We would also like to share our respect to those who helped us make through the struggles in creating the compiler system to pass to its point of a well grounded program to still by our stance for the series of defense we came through.

We would also like to clip a message of acknowledgement for this group as we make it work through all the hardships having to nourish each individual's potential to perform in their own ways and develop a fighting chance to defend the trailing chapters of this course.

And to you, Lord God, who guided us in keeping our spirits up and minds clear of all the struggles and down strokes of making a group together. We thank you for your guiding light towards every last bit of journey finishing this course as a whole.

For the future students of this course, it may be renamed for the years to come, or remain as Compiler Design, but we wanted for you to make it through the year ready to use all the wisdom you can pull up, grow and learn in developing your analysis, logics, and the system itself.

We all acknowledge you.

TABLE OF CONTENTS

GENERAL INFORMATION ABOUT C - GRASS PLUS

I. Language Overview

C-Grass PLUS is a programming language that holds "Python Language Utilization System" as it is inspired with how easier to read and understand Python language with the combination of C-language's coding format to help both parties of programming language creators and coding programmers a good agreement of maintaining all the important coding standard.

The bodies of water in the ocean and sea are vast. Under its depths is the tallest plant in the world collectively protecting and giving shelter to the smaller creatures such as any type of fishes, they're brushed together by the soft and supple breeze of these plants called seagrass. Seagrass gave the spark for this programming language project standing strong beneath the water surface against all the odds of the atmosphere out of the sea, the strong currents pulling the waves splash the earth's thin crust.

Sea, indeed, C, C-grass will continue to grow reaching the sun's rays with its brightly warm light to the tip of the silken cold-watered leaf. Against all the odds of its vulnerability and fragility are the illuminating smiles as the dolphins jump around up the white blue sky.

C-Grass PLUS will be a next-evolution step for a proactive environmental campaign with the power of programming and learning these non-living creatures, machines.

II. General Rules

1. All reserved words are case-sensitive and should be written in lowercase letters.
2. The program begins with the seed(start) reserved word, then ends with the plant(end) reserved word, and any code or statements written afterwards will not be read.
3. Comments are ignored by the compiler. To insert a single-line comment, the question mark (?) symbol is used. To insert a multi-line comment, the less than symbol (<) followed by two hyphens (-) declares the start of a comment (<--), then two hyphens (-) followed by a greater than symbol (>) declares the end of a comment (-->).
4. The floral(global) keyword is used for global variable declaration and is only allowed outside the garden(main) function and sub-functions.
5. Local variables are only accessible within a specific function. The body of each function is enclosed within an open parenthesis {(} and a closing parenthesis {)}.
6. Only one garden(main function) is allowed to exist within the program. While sub-functions (non-void function and void function) are written after.
7. All statements, functions and declarations should be terminated by a semicolon (;).
8. Only non-void functions use the regrow(return) reserved word for returning value to the main function, while void functions do not.
9. Function parameters possess a local scope within the function statement itself.
10. Subfunctions can have other subfunctions as parameters, arguments, or return value.
11. In naming all statements, functions and declarations, the hashtag(#) is used for identifiers that comprise up to fifty(50) characters.
12. There are two types of data:
    a. (1) Common data types: tint(integer), flora(float), chard(character), string(string) and bloom(boolean). And;
    b. (2) Sequence Data types: florist(list), tulip(tuple), dirt(dictionary), and stem(set).
    c. Typecasting is supported.
13. All sequence data types have a maximum of three dimensions.
14. Escape characters can be used in any part of the string.
15. Operators comprise arithmetic most useful for numeric; and relational, logical, comparison and assignment can be used for any type of data.

III. Structure of the Language

```
seed

?single line comment
<-- multi-line comment -->

floral <global declaration>;

garden()
(
   <statement/s>;
);

<function-datatype> <identifier> (<datatype> <identifier>)
(
   <statement/s>;

   regrow <statement/s>;
);

plant
```

Where:

<statement/s>	→	local variable statements : all datatypes; see <datatype>
	→	input-and-output statements : inpetal(input), mint(print)
	→	iterative statements : fern(for), willow(while)
	→	conditional statements : leaf(if), eleaf(elif), moss(else)
	→	assignment statements
	→	calling a function : #identifier and its arguments value
	→	clear terminal : clear(clear)
	→	stop terminal : break(break)

<function-datatype>	→	tint(integer)
	→	flora(float)
	→	chard(character)
	→	string(string)
	→	bloom(boolean)
	→	viola(void)
<datatype>	→	tint(integer)
	→	flora(float)
	→	chard(character)
	→	string(string)
	→	bloom(boolean)
	→	florist(list)
	→	tulip(tuple)
	→	dirt(dictionary)
	→	stem(set)
<identifier>	→	variable name
	→	function name
	→	parameter name
<parameter-type>	→	all datatypes; see <datatype>

IV. Reserved Words

Reserved Words	Python	Description
Main Function		
garden	-	Keyword used to represent the main function
Start and End		
seed	-	Used to start the program
plant	-	Used to end the program
Data Types		
tint	int	Used to declare a data type representing whole numbers.
flora	float	Used to declare a data type representing numbers with fractional parts.
chard	character	Used to declare a data type representing single alphanumeric character
string	string	Used to declare a data type representing a sequence of characters.
bloom	boolean	Used to declare a data type representing logical true or false.
florist	list	Used to declare a data type representing a mutable ordered collection of items that can contain elements or different data types.
tulip	tuple	Used to declare a data type representing data like lists but cannot be modified after creation.
dirt	dictionary	Used to declare a data type representing a collection of key-value pairs that are unique and immutable.
stem	set	Used to declare a data type representing an unordered collection of unique elements.

Input and Output Statements		
inpetal()	input()	A function that reads a line of text from the user as a string.
mint()	print()	A function that can display text, variables, and the results of expressions.
Conditional Statements		
leaf	if	Conditional statement that is used to execute a block of code only if a specified condition is true.
moss	else	Conditional statement that is used in conjunction with an if statement to specify a block of code that should be executed when the if condition is false.
eleaf	elif	Conditional statement that allows you to specify multiple conditions in a sequence.
tree	match	Conditional statement that compares the value, condition, or pattern of the value to subsequent branches.
branch	case	Executes a block of code inside a tree if the specified value, condition, or pattern is satisfied.
nut	not	Negates a conditional or logical statement.
true	true	Data value used to represent the truth value of an expression and is not equal to zero.
false	false	Data value used to represent the false value of an expression and is equal to zero.
break	break	Conditional statement used to terminate loops, statements, and the program itself.
Iterative Statements		
fern	for	Looping statement that is used for iterating over a sequence (such as a list, tuple, string) or other iterable objects.

willow	while	Looping statement that repeatedly executes a block of code as long as a specified condition is true.
Others		
*args	*args	Allows us to pass multiple arguments of non-keyword arguments to a function.
**kwargs	**kwargs	Allows us to pass the variable length of keyword arguments to the function.
at	in	Used to check if a particular value is present in a sequence
bare	None	Represents the absence of a value or a null value same with an empty data.
clear	-	Used to clean the output field.
floral	global	Used to declare the global variable.
getItems()	getitems()	Function for getting all of the items of a dictionary, returns a list of tuples.
getKeys()	keys()	Function for getting all of the keys of a dictionary, returns a list of keys
getValues()	values()	Function for getting all of the values of a dictionary, returns a list of values.
hard	-	Used to declare a constant value from a variable.
lent	len	Used to get the length of sequence data types.
regrow	return	Used to specify the return value of a function.
viola	void	Used to define a function that does not return a value.

V. Reserved Symbols

Reserved Symbol	Description
Arithmetic operators	
+	Used for addition.
-	Used for subtraction. It also signifies a negative number.
*	Used for multiplication.
/	Used for division
%	Used for modulo which computes for the remainder.
**	Used for exponential value.
//	Used for floor division to the nearest integer.
Assignment operators	
=	Used for equations and assigning values.
+=	Used for instantly equating added values.
-=	Used for instantly equating subtracted values.
*=	Used for instantly equating multiplied values.
/=	Used for instantly equating divided values.
%=	Used for instantly equating remainder of divided values.
**=	Used for instantly equating multiplied values by exponential values.
//=	Used for instantly equating divided values to the nearest integer.
Comparison operators	

==	Used for comparing equal values.
!=	Used for comparing not equal values.
>	Used for comparing greater than values.
<	Used for comparing less than values.
>=	Used for comparing greater than or equal values.
<=	Used for comparing less than or equal values.
Logical operators	
=&	Used as logical AND operator
=/	Used as logical OR operator
Others	
(	Used to initiate grouping expressions and invoking functions
)	Marks the end for grouping expressions and invoking functions
[	Indicates the beginning for crafting florists, accessing florists elements, and indexing sequences
]	Signals the completion for crafting florists, accessing florists elements, and indexing sequences
{	Used to start defining tulip, dirt, stem, and delineating code blocks in specific contexts
}	Denotes the conclusion for constructing tulip, dirt, stem, and delineating code blocks in specific contexts
“	Employed to declare strings enclosed in double quotes
‘	Employed to declare chards enclosed in single quotes
,	Used to separate elements in a florist, tulip, dirt, stem, or function arguments and parameters.

:	Marks the pair keys with values in dictionaries.
.	Used to connect whole numbers and floating-point decimal numbers.
;	Terminates statements
/	Escape characters used to prevent errors in strings when using certain special characters.
#	Used to emphasize that the given keyword is an identifier.
?	Declares a single-line comments
<--	Declares the start of multi-line comments
-->	Declares the end of multi-line comments
_	Used for creating a default case in a tree-branch statement.

VI. Specific Rules

Identifiers

Identifiers are names given to various program elements such as variables, functions, etc.

I. In naming identifiers, it must start with a hashtag(#) followed by a single letter. It will accept a combination of letters and numbers afterwards.

II. The first hashtag(#) is not counted among the character count.

III. Identifiers are required to have at least one(1) character up to fifty(50) characters.

IV. Special characters are not allowed except underscores(_).

FORMAT
#<let>(<numulet> | <null>)^49

Where:

let → letters

numulet → numbers, underscores, and letters

EXAMPLE	
Valid	Invalid
#b	#$@#(
#Bl0ck	#0(*&cent
#Stud_num	#S_t u de nt s_2
#tint	#stu%d()

Data Types

Data types are reserved words given to various program elements to control the type of data value it carries.

a. tint (int) - data type representing whole numbers only.

I. Negative tint numbers are preceded by a hyphen-minus sign(-).

II. Tint can hold a maximum number of six (6) whole numbers ranging from values -999999 to 999999.

III. The default value is 0.

IV. Spaces, commas, and any non-numeric are not allowed.

V. Leading zeros are ignored.

FORMAT
0 (- | <null>)<dig>(<num> | <null>)^5

Where:

dig → numbers from 1 to 9

num → numbers from 0 to 9

EXAMPLE	
Valid	Invalid
100000	100 0 0 0
-123456	537457385543
0	+1
000000010	112,3
121	534-3

b. flora (float) - data type representing real numbers.

I. Negative flora numbers are preceded by a hyphen-minus sign(-).

II. Flora can hold a maximum number of six (6) whole numbers and six (6) fractional digits ranging from values -999999.999999 to 999999.999999.

III. Spaces, commas, and any non-numeric characters are not allowed except for a period (.).

IV. Trailing zeros are ignored.

FORMAT
0 (- | <null>)<dig>(<num> | <null>)^5 . (<num> | <null>)^6

Where:

dig → numbers from 1 to 9

num → numbers from 0 to 9

EXAMPLE	
Valid	Invalid
100000.111111	100,111.213
-123456	3.2.4
0.10000000000	1.2A
10.0	127454121.52463839

c. chard (character) - data type representing a single display unit of information equivalent to one alphabetic symbol, digit, or letter.

I. Chards can encompass letters, numbers, symbols, or any other printable ASCII character.

II. Chards can only have a single character.

III. Chards are enclosed by single quotation marks (“)

FORMAT
’ASCII | <null>’

EXAMPLE	
Valid	Invalid
‘a’	‘a1’
‘C’	‘v_’

d. string (string) - data type representing a set of alphanumeric values.

I. Strings can encompass letters, numbers, symbols, or any other printable ASCII character.

II. Strings are enclosed with double quotation marks (“ ”).

III. Using quotation marks (“ ”) multiple times in a single line alone is not allowed. Labels with backslash (\) to all the quotation marks are considered as a character.

IV. In the string data type, There are a total of five(5) escape characters that can be used:

(a) \n = for new line

(b) \t = for tab character

(c) \' = for single quote character

(d) \" = for double quote character

(e) \\ = for backlash character

V. String supports interpolation.

(a) The curly braces ({ }) are used inside string literals to enclose variable identifiers.

(b) The values corresponding the variable will be included inside the string and be treated as a string literal.

<table>
<tr><th>FORMAT</th></tr>
<tr><td>“(<ASCII> (<escape> | <null>)^∞({<identifier>} | <null>)^∞ |<null>)^∞”</td></tr>
</table>

Where:

<escape> → Escape characters

<identifier> → variable name

EXAMPLE	
Valid	Invalid
“Hello!”	Hello!
“He said, \"Hey!\""	“He” said, "Hey!""
“Characters\n 123\n @#!&”	‘Characters 123 @#!&”’

e. bloom (boolean) - data type representing binaural values 0 or 1, and true-or-false.

I. One(1) is equivalent to true, while zero(0) is equivalent to false.

II. Bloom values are case sensitive.

FORMAT
true, false, 0, 1

EXAMPLE	
Valid	Invalid
1	Truer
0	t
true	fAlsify
false	FAls

Sequence Data Types

Sequence data types are reserved words given to various program elements to read a collection of data values.

a. florist (list) - a sequence data type representing a list of any data type value.

I. Can contain a mix of data types and sequence data types.

II. Florists are enclosed within square brackets ([ ]), and can be declared empty.

III. Florist elements are separated by comma (,). Duplicated elements are allowed.

IV. Florists can contain a maximum of 150 elements.

V. Florist is mutable.

VI. Florist elements are accessed using indexing, starting from index 0.

VII. For elements with string data type, using double quotation marks (“ ”) multiple times for a single element is not allowed. Label with backslash (\) to all the quotation marks considered as characters.

VIII. Operators are not allowed to be used between florists, except for plus (+) for concatenation.

FORMAT

[<null> | <datatype-value>(,<datatype-value> | <null>)^∞]

Where:

<datatype-value> → tint(integer) literal
→ flora(float) literal
→ chard(character) literal
→ string(string) literal
→ bloom(boolean) literal
→ florist(list)
→ tulip(tuple)
→ dirt(dictionary)
→ stem(set)
→ #identifier

EXAMPLE	
Valid	Invalid
[[‘a’,’b’], “cat”]	string1
[“string_a”, “string_b”, “string_c”]	[{{[{string_a, string_b, “string_c”]]

b. tulip (tuple) - an unmodifiable sequence data type representing a set of any printable values.

I. Can contain a mix of data types and sequence data types.

II. Tulips are enclosed within curly brackets ({ }), and it is not recommended to leave it empty.

III. Tulips can contain a maximum of 150 elements.

IV. Tulip elements are immutable once declared. The values are sustained, and any modifications are invalidated.

V. Tulip elements are separated by comma (,) and can be similar/duplicate elements.

VI. For elements with string data type, using double quotation marks (“ ”) multiple times for a single element is not allowed. Label with backslash (\) to all the quotation marks considered as characters.

FORMAT
{<null> | <datatype-value> (,<datatype-value> | <null>)^∞}

Where:

<datatype-value>	→	tint(integer) literal
	→	flora(float) literal
	→	chard(character) literal
	→	string(string) literal
	→	bloom(boolean) literal
	→	florist(list)
	→	tulip(tuple)
	→	dirt(dictionary)

→ stem(set)
→ #identifier

EXAMPLE	
Valid	Invalid
{"string1"}	{{{{{string1
{"string_a", "string_b", "string_c"}	[[[string_a, string_b, string_c]]]

c. dirt (dictionary) - a key-paired sequence data type representing a set of any printable values.

I. Can contain a mix of data types and sequence data types.
II. Dirts are enclosed within curly brackets ({ }), and can be declared empty.
III. Duplicate keys are not allowed.
IV. Dirt can contain a maximum of 150 key-value pair elements.
V. Dirt key-paired elements are separated by comma (,) and the values can be similar/duplicate.
VI. Keys only follow string data type and are immutable.
VII. Empty Declaration of dirts are not
VIII. For elements with string data type, using double quotation marks (" ") multiple times for a single element is not allowed. Label with backslash (\) to all the quotation marks considered as a character.

<table>
<tr><th>FORMAT</th></tr>
<tr><td>{"(<ASCII>)^∞" : <datatype-value>((,"(<ASCII>)^∞" : <datatype-value>) | <null>)^∞}</td></tr>
</table>

Where:

<datatype-value> → tint(integer) literal
→ flora(float) literal
→ chard(character) literal
→ string(string) literal

→ bloom(boolean) literal
→ florist(list)
→ tulip(tuple)
→ dirt(dict)
→ stem(set)
→ #identifier

EXAMPLE	
Valid	Invalid
{“key1” : [1, 2, 3], “key2” : [“string_a”, “string_b”, “string_c”]}	[[key1 {1, 2, 3}, key2 {string_a, string_b, string_c}]]
{“key_a” : tulip_petal1}	[key-a] {[tulip_petal1]}
{“keyA” : florist_name1}	(keyA [:] {florist_name1})

d. stem (set) - a key-paired sequence data type representing a set of any unique printable values.

I. Can contain a mix of data types and sequence data types.
II. Stems are enclosed within curly brackets ({ }), and can be declared empty.
III. Stem elements are immutable once declared. Only unique values are sustained, and any follow-up actions are ignored except for sorting elements.
IV. Stems can contain a maximum of 150 elements.
V. The default sort arrangement is in ascending order.
VI. Stem elements are separated by comma (,).
VII. For elements with string data type, using double quotation marks (“ ”) multiple times for a single element is not allowed. Label with backslash (\) to all the quotation marks considered as a character.

FORMAT
{<null> | <datatype_value> (, <datatype_value> | <null>)^∞}

Where:

<datatype-value> → tint(integer) literal
→ flora(float) literal
→ chard(character) literal
→ string(string) literal
→ bloom(boolean) literal
→ florist(list)
→ tulip(tuple)
→ dirt(dict)
→ stem(set)
→ #identifier

EXAMPLE	
Valid	Invalid
{ 5,8,3,4,7,2,1 }	[[5,7,8,3,4,4,7,5,2,1}}
{“string3”, “string2”, “string1”}	string3, string2, string3, string1, string2

Indexing

This is the way for accessing the elements of a sequence data type (florist, tulip, etc.) and for selecting characters in a string.

For florist (list), tulip (tuple), stem (set), and string (string):

1. To get individual elements, indexing is done by using square brackets ([ ]) with the index number inside after stating the identifier. The index of the first element is zero (0).
2. Only positive integers are allowed for indexing and slicing.

3. Indexing does not work with the dirt sequence data type since it is a key-value pair. On the other hand, when the value of a key-value pair in a dirt is another sequence data type or a string, indexing and slicing can be used.

FORMAT FOR INDEXING
<sqnc-type>[<index>];

Where:

<sqnc-type> → string, florist, stem, tulip

<index> → tint literal, identifier

EXAMPLE	
Valid	Invalid
#list[2]	#tuple[:]
#str[#a]	#dict[3]

Typecasting

Is the process of converting the value of a single data type (such as a tint, flora, etc.) into another data type.

Rules for Typecasting:

1. All data types can be type casted except dirt.
    a. Can only typecast common data type into another common data type, likewise, sequence data type can only be type casted into another sequence data.
    b. A dirt data type cannot be type casted but a dirt value can.
2. Cannot typecast a common data type into a sequence data type, and vice versa.

Supported Typecasting:

1. Common data type → Common data type
    a. tint → flora
    b. flora → tint
    c. tint or flora → string
    d. string → tint or flora
        i. Not all string can be converted to tint or flora e.g., “abc” cannot be typecasted to tint or flora but “123” can.
    e. chard → string
    f. string → chard
        i. Not all string can be converted to chard e.g., “abc” cannot be typecasted to chard but “a” can.
    g. bloom → string
    h. string → bloom
        i. Only “true” or “false” can be type casted to bloom
    i. bloom → tint or tint → bloom
        i. 0 and 1 are the only ones that can be type casted to tint, because 0 is false and 1 is true.
    j. chard → tint
    k. chard → flora
        i. Not all chards can be converted to tint or flora e.g., ‘a’ cannot be typecasted to tint or flora but ‘65’ can.

2. Sequence data type → Sequence data type
    a. florist → tulip
    b. florist → stem
    c. stem → tulip
    d. tulip → stem
    e. tulip → florist
    f. stem → florist

FORMAT
<datatype>(<datatype-value>)

Where:

<datatype-value>	→	tint
	→	flora
	→	chard
	→	string
	→	bloom
	→	florist
	→	tulip
	→	dirt
	→	stem

<datatype-value>	→	tint(integer) literal
	→	flora(float) literal
	→	chard(character) literal
	→	string(string) literal
	→	bloom(boolean) literal
	→	florist(list)
	→	tulip(tuple)
	→	dirt(dict)
	→	stem(set)
	→	#identifier
	→	#identifier[index]
	→	#identifier[key]

EXAMPLE	
Valid	Invalid
tint(3.4)	tint(3)
florist(#tulip_var)	stem(records)

Variables

Variables hold data values, which can change or vary as the program runs.

Rules for Declaring and Initializing Variables:

1. A variable declaration begins with the data type, followed by the identifier or variable name.
2. An equal sign (=) is used after the variable name to assign a data value.
3. Variable declarations without initialization are allowed.
4. All variable declarations must have unique identifiers within the same scope.
    a. Global variable identifiers must be unique and not have any duplicate identifiers in any part of the program.
5. In declaring variables, any unmatching data type and data values in a single line are not allowed. Other than that, each variable declared in a single line is separated with a comma (,).
6. Variable initialization and assignment without a declaration are not allowed.
7. Variables defined outside any functions without the floral(global) keyword are not allowed.
8. In order to declare a constant value from a variable, the keyword hard(constant) must be used before the data type.
9. A tulip(tuple) cannot be declared as a constant with hard(constant) as it is already immutable.
10. Swap notation and sequence unpacking cannot be used for declaring variables, only for instantiating or assigning values.
11. Creating a constant variable with multiple variable declarations in a single line will only require one hard(constant) keyword at the beginning of the line before the data type of the variables.
12. Declaring and instantiating variables at the same line are allowed. Using arithmetic operations for instantiating variables or declaring and instantiating variables are allowed.

FORMAT
<datatype> <identifier> = <value>; <datatype> <identifier> ; <datatype> (<identifier> = <value> | <null>)^∞; hard <hdatatype> <identifier> = <value>; hard <hdatatype> <identifier> ; hard <hdatatype> (<identifier> = <value> | <null>)^∞;

Where:

<datatype>	→	tint(integer)	→	flora(float)
	→	chard(character)	→	string
	→	florist(list)	→	tulip(tuple)
	→	dirt(dict)	→	stem(set)
<identifier>	→	variable name		
<value>	→	data value		
<hdatatype>	→	tint(integer)	→	flora(float)
	→	chard(character)	→	string
	→	florist(list)	→	dirt(dict)
	→	stem(set)		

EXAMPLE	
Valid	Invalid
string #itsString = “Hello, World!”;	string %foo hard = “Hello, Word”
floral hard flora #itsFlora = 18.24;	stem foo bar hard = #18
hard tint #foo = 18, #itsTint = 19;	dirt hard = “hello”, bar = 19;
florist #emptyFlorist;	tulip #foo hard =;

garden()(
    tint #a = 4;
    tint #b = 90;
)
	garden()(
	    tint #num = 4;
	    tint #num = 90;
	)

seed
 	
floral flora #pi = 3.14;
 	
garden()(
    flora #r = 3;
    flora #circle_area = 2 * #pi * r;
)
 	
plant
	seed
	
	floral flora #pi = 3.14;
	
	garden()(
	    flora #pi = 1.64;
	)
	
	plant

Conditionals

Conditionals are statements which execute parts of the program, given that conditions are met.

1. A leaf(if) is not always followed by an eleaf(else if) or moss(else).
2. An eleaf(else if) and moss(else) always comes after a leaf(if).
3. In a tree-branch(switch-case), the maximum number of branch(case) statements that can be used in a tree(switch) is 150.
4. A branch(case) with no given condition is considered as the default branch(case), wherein a default branch(case) will be executed if the input does not meet the conditions of any other branch(case) statements.
5. All branches(cases), including the default branch(case), must end with a break(break).

FORMAT

```
leaf (<condition>)(
<statement/s>;
);
eleaf (<condition>)(
<statement/s>;
);
moss (
<statement/s>;
);
tree(<identifier>)(
    branch ‘1’: <statement/s>; break;
    branch ‘2’: <statement/s>; break;
    branch: <statement/s>; break;
);
```

Where:

<condition>	→	runs while #identifier is true
	→	#identifier < tint literal | #identifier
	→	#identifier > tint literal | #identifier
	→	#identifier >= tint literal | #identifier
	→	#identifier <= tint literal | #identifier
	→	#identifier != tint literal | #identifier
	→	#identifier == tint literal | #identifier
	→	#identifier =& tint literal | #identifier
	→	#identifier =/ tint literal | #identifier
<statement/s>	→	local variable statements : all datatypes; see <datatype>
	→	input-and-output statements : inpetal(input), mint(print)
	→	iterative statements : fern(for), willow(while)
	→	conditional statements : leaf(if), eleaf(elif), moss(else)
	→	assignment statements
	→	calling a function : #identifier and its arguments value

→ clear terminal : clear(clear)

→ stop terminal : break(break)

<identifier> → variable name

Examples	
Valid	Invalid
leaf (#score >= 20) (mint("Passed"););	leaf (score + 20) (mint("Final Score"););

leaf (#score >= 90) (
mint("A");
);
eleaf (#score >=80) (
mint("B");
eleaf (#score >=70) (
mint("C");
);
moss (
mint("D or F");
);
	eleaf (#score >=80) (
	mint("B");
	eleaf (#score >=70) (
	mint("C");
	);
	moss (
	mint("D or F");
	);
	leaf (#score >= 90) (
	mint("A");
	);

string #weekday(tint #n)(tree(#n)(branch 0: regrow "Monday"; break; branch 1: regrow"Tuesday"; break; branch 2: regrow "Wednesday"; break; branch 3: regrow "Thursday"; break; branch 4: regrow "Friday"; break; branch 5: regrow "Saturday"; break; branch 6: regrow "Sunday"; break;	string #weekday(tint #n)(tree(#n)(branch 0: regrow "Monday"; branch 1: regrow"Tuesday"; break; branch 2: regrow "Wednesday"; break; branch 3: regrow "Thursday"; break; branch 4: regrow "Friday"; break; branch 5: regrow "Saturday"; break; branch 6: regrow "Sunday"; break;

branch _: regrow "Invalid day number";
    break;
);
);
mint(#weekday(3));
mint(#weekday(6));
mint(#weekday(7));
	…
	    branch 160: regrow "Invalid day
	    number";
	);
	);
	mint(#weekday(3));
	mint(#weekday(6));
	mint(#weekday(7));

Iteratives

Iteratives are statements denoting iteration or repeating process until it finishes.

1. Only the fern(for) and willow(while) keywords work with iteratives.
2. You can only use either a boolean value or an integer value for its parameters.
3. Statements written inside an iterative are the statements only read repeatedly by the loop of its process.
4. For a breaking-point, either a comparator or force-close by using the break(break) keyword is used.

FORMAT
fern (<counter>; <condition>; <update>) (<statement/s>); willow (<condition>) (<statement/s>);

Where:

<counter> → tint #identifier = tint literal

<condition> → runs while #identifier is true
→ #identifier < tint literal | #identifier
→ #identifier > tint literal | #identifier
→ #identifier >= tint literal | #identifier
→ #identifier <= tint literal | #identifier
→ #identifier != tint literal | #identifier

	→	#identifier == tint literal | #identifier
	→	#identifier =& tint literal | #identifier
	→	#identifier =/ tint literal | #identifier
<update>	→	#identifier += tint literal
	→	#identifier -= tint literal
<statement/s>	→	local variable statements : all datatypes; see <datatype>
	→	input-and-output statements : inpetal(input), mint(print)
	→	iterative statements : fern(for), willow(while)
	→	conditional statements : leaf(if), eleaf(elif), moss(else)
	→	assignment statements
	→	calling a function : #identifier and its arguments value
	→	clear terminal : clear(clear)
	→	stop terminal : break(break)

Examples	
Valid	Invalid
fern(i=1;i<5;i+=1) (mint(i););	fern(i=”a”;i<5;i-=1) (mint(i);)
fern(i=10;i>0;i-=2) (mint(i););	fern(i=100;i>0;5) (mint(i);)

willow(true)
(
i += 1;
if(i==5)
(
break;
);
);
	willow(“hello”)
	(
	i += 1;
	j += 1;
	)

bloom #isFalse = false;
willow(#isFalse == false)
	bloom #willAlwaysBeTrue = true;
	willow(willAlwaysBeTrue = ‘true’)

(
  i += 1;
  if(i==5)
  (
    #isFalse = true;
  );
);
	(
	  i += 1;
	  j += 1;
	)

Functions

Functions are program instructions that perform a specific task, packaged as a unit.

1. When calling a function, the number of arguments should match the number of parameters as well as its order.
2. Non-void functions can only have tint(integer), flora(float), bloom(boolean), chard(character), and string(string) as its data type and must always regrow(return) a value.
3. Void functions do not have a regrow(return) keyword to return any values. The viola(void) is used for declaring a void function.
4. Function parameter elements are separated by a comma(**,**).
5. Every function must contain at least one statement within its body. Even the regrow(return) statement is counted as one.
6. Sub-functions can have other sub-functions as parameters, arguments, or return value.
7. First-class functions are supported. Any sub-function can be treated as a first-class function. First-class functions can be:
    a. Assigned to variables
    b. Passed as arguments to other functions
    c. Returned as values from other functions
8. Higher-order functions are supported. Any sub-function can be treated as a higher-order function. Higher-order functions are used for:
    a. Abstraction – Higher-order functions allow you to abstract over actions, behaviors, or operations. They enable you to define generic

functions that can operate on a variety of functions, making your code more flexible and reusable.

b. Functional Composition – Higher-order functions facilitate function composition, which involves combining two or more functions to produce a new function. Function composition allows you to create pipelines of transformations, making your code more declarative and expressive.

NOTE: First-class functions are functions that are being returned, being used as parameters for another function, or are being assigned to a variable. Higher-order functions are functions that return a function and/or use a function as a parameter

9. Variable length parameters:
    a. *args - for non-keyworded arguments.
        i. The *args syntax is used to pass a non-keyworded, variable-length argument list to a function.
        ii. The args name is a convention, but it can be any valid variable name preceded by an asterisk (*).
        iii. Inside the function, args is a tuple containing all the extra positional arguments passed to the function.
    b. **kwargs - for keyword arguments.
        i. The **kwargs syntax is used to pass a variable-length keyword argument dictionary to a function.
        ii. The kwargs name is a convention, but it can be any valid variable name preceded by a double asterisk (**).
        iii. Inside the function, kwargs is a dictionary containing all the keyword arguments passed to the function.
    c. There can only be one *args and **kwargs parameter for every sub-function.
    d. When both *args and **kwargs are present at the same time, *args always comes first and followed ny **kwargs.
    e. Positional parameters always comes first before *args and **kwargs.

FORMAT (Main Function)
garden() (<statement/s>;)

FORMAT (Non-Void Function)
<function-datatype> <identifier> (<datatype> <identifier>) (<statement/s>; regrow <statement/s>;)

FORMAT (Void Function)
viola <identifier> () (<local declaration>; <statement/s>;)

Where:

<statement/s>	→	all datatypes in <datatype> for local variable declaration
	→	inpetal(input) and mint(print) for input-and output statements
	→	iterative statements : fern(for), willow(while)
	→	conditional statements : leaf(if), eleaf(elif), moss(else)
	→	assignment statements with the use of assignment operations
	→	#identifier and its arguments value is used in calling a function
	→	clear(clear) to clear terminal
	→	break(break) to stop terminal
<function-datatype>	→	tint(integer)

	→	flora(float)
	→	chard(character)
	→	string(string)
	→	bloom(boolean)
	→	viola(void)
<datatype>	→	tint(integer)
	→	flora(float)
	→	chard(character)
	→	string(string)
	→	bloom(boolean)
	→	florist(list)
	→	tulip(tuple)
	→	dirt(dictionary)
	→	stem(set)
<identifier>	→	variable name
	→	function name
	→	parameter name

Example (Sub-Function)	
Valid	Invalid
tint #add (tint #num1, tint #num2) (#tint #adding = #num1 + #num 2; regrow #adding;)	add (num1, num2) (adding = num1 + num 2 regrow adding)
viola #violet () (mint (“the fragrant lavender”);)	viola () (regrow regrow regrow)

	Example (First-Class Function)	
	Valid	Invalid
Variables Assignment	tint #a = #sqrt(5);	tint #a = func(c)

Function Passing	tint #apply_func(tint #operation, tint #x, tint #y)(
	    regrow #operation(#x, #y);
	);
	
	tint #add(tint #x, tint #y)(
	    regrow #x + #y;
	);
	
	tint #result = apply_func(add, 5, 3);
		tint #apply_func(tint #operation, tint #x, tint #y)
		(
		    regrow #operation(#x, #y);
		);
		
		tint #result = apply_func(3,1);

Function Return	tint #multiply_by(tint #n)(
	    tint #multiplier(tint #x)(
	        regrow #x * #n;
	    );
	    regrow multiplier;
	);
	
	tint #multiply_by_5 = #multiply_by(5);
	
	tint #result = #multiply_by_5(10);
	
	?Output: 50
		tint #multiply_by(tint #n)(
		    tint #multiplier(tint #n)(
		        regrow #n;
		    );
		    regrow multiplier;
		);
		
		tint #multiply_by_5 = #multiply_by(5);
		
		tint #result = #multiply_by_5(10);

	Example (Higher-order Function)	
	Valid	Invalid

Abstraction	tint #apply_twice(tint #func, tint #arg)(
	regrow #func(#func(#arg));
	);
	
	tint #double(tint #x)(regrow #x * 2;);
	
	tint #result = #apply_twice(#double, 2);
	
	?Output: 8 (2 * 2 * 2)
		tint #apply_twice(tint #func, tint #arg)(
		regrow #func(#func(#arg));
		);
		
		viola #hello()(
		mint("Hello, World!");
		);
		
		tint #result = #apply_twice(#hello, 2);
		
		<--Error: function hello is not a tint return type function–>

Functional Composition	tint #compose(tint #f, tint #g)(
	regrow #f(#g(#x));
	);
	tint #x(tint # value)(regrow #value;)
	tint #add_one(tint #x)(regrow x + 1;);
	tint #double(tint #x)(regrow x * 2;)
	
	tint #add_one_then_double = compose(double, add_one);
	
	tint #result = add_one_then_double(3) ;
	
	?Output: 8 (double(add_one(3)))
		tint #compose(tint #f, tint #g)(
		regrow #f(#g(#x));
		);
		
		tint #result = compose(3,1);
		
		<--Error: 3 and 1 are not functions-->

Example (*args and **kwargs)	
Valid	Invalid

tint #sum(tint *#num)
(
#total = ;0
fern(#i in #num)(
total += #i;
);
regrow #total;
)
	string #display(string *#str, string #name)(
	fern(tint #i=0; #i<lent(#str);#i+=1)(
	mint(#str);
	mint(#name);
	);
	);

Operators

A symbol that performs specific mathematical, relational, or logical operations.

1. All operators can be used for any data type and sequence data type except for arithmetic operators.
2. Comparison and logical operators always produce boolean values.

a. **Arithmetic operators** – used for mathematical operations.

Operator	Description	Format
+	Adds two operands.	<num>+ <num> #identifier + #identifier
	Can be used in strings for concatenation.	<numlet> + <numlet> <let> + <let> #identifier + #identifier
-	Subtracts two operands.	<num> - <num>
*	Multiples two operands.	<num> * <num>

/	Divides two operands.	<num> / <num>
%	Returns the remainder when the first operand is divided by the second.	<num> % <num>
**	Returns the first operand raised to the power of the second operand.	<num> ** <num>
//	Divides two operands and rounded the quotient to the nearest integer.	<num> // <num>

b. **Assignment operators** – used for assigning values.

Operator	Description	Format
=	Assigns the value of the right operand to the left operand.	(#identifier | <numlet> | <num> | <let>) = (#identifier | <numlet> | <num> | <let>)
+=	Adds the right operand with left and then assigns the result in the left operand.	(#identifier | <numlet> | <num> | <let>) += (#identifier | <numlet> | <num> | <let>)
-=	Subtracts the right operand with left and then assigns the result in the left operand.	(#identifier | <num>) -= (#identifier | <num>)
*=	Multiplies the right operand with left and then assigns the result in the left operand.	(#identifier | <num>) *= (#identifier | <num>)
/=	Divides the right operand with left and then assigns the result in the left operand.	(#identifier | <num>) /= (#identifier | <num>)
%	Takes the remainder and assigns it to the left operand.	(#identifier | <num>) %= (#identifier | <num>)
**	Calculates exponential value using	(#identifier | <num>) **=

<table>
<tr><td></td><td>operands and assigns value to the left operand.</td><td>(#identifier | <num>)</td></tr>
<tr><td>//</td><td>Calculates the rounded-off quotient to the nearest integer and assigns it to the left.</td><td>(#identifier | <num>) //= (#identifier | <num>)</td></tr>
</table>

c. **Comparison operators** – used to compare values in conditional, input-and-output and iterative statements.

<table>
<tr><th>Operator</th><th>Description</th><th>Format</th></tr>
<tr><td>==</td><td>Compare if the left and right operand are equal.</td><td>(#identifier | <numlet> | <num> | <let>) == (#identifier | <numlet> | <num> | <let>)</td></tr>
<tr><td>!=</td><td>Compare if the left and right operand are not equal.</td><td>(#identifier | <numlet> | <num> | <let>) != (#identifier | <numlet> | <num> | <let>)</td></tr>
<tr><td>></td><td>Compare if the left operand is greater than the right operand.
Can only be used in numeric values.</td><td>(#identifier | <num>) > (#identifier | <num>)</td></tr>
<tr><td><</td><td>Compare if the left operand is less than the right operand.
Can only be used in numeric values.</td><td>(#identifier | <num>) < (#identifier | <num>)</td></tr>
<tr><td>>=</td><td>Compares if the left operand is greater than or equal to the right operand.
Can only be used in numeric values.</td><td>(#identifier | <num>) >= (#identifier | <num>)</td></tr>
<tr><td><=</td><td>Compares if the left operand is less than or equal to the right operand.</td><td>(#identifier | <num>) <= (#identifier | <num>)</td></tr>
</table>

	Can only be used in numeric values.	

d. **Logical operators** – used for conditional, input-and-output and iterative statements.

<table>
<tr><th>Operator</th><th>Description</th><th>Format</th></tr>
<tr><td>=&</td><td>Returns true if both operands are true.</td><td>(#identifier | <numlet> | <num> | <let>) =& (#identifier | <numlet> | <num> | <let>)</td></tr>
<tr><td>=/</td><td>Returns true if one of the operands is true.</td><td>(#identifier | <numlet> | <num> | <let>) =/ (#identifier | <numlet> | <num> | <let>)</td></tr>
</table>

e. **Precedence of All Operators**

Precedence	**Operator**	**Associativity**
1	( )	Innermost first
2	*, /, %	Left to right
3	+, -	Left to right
4	<, >, <=, >=	Left to right
5	*=, /=, %=	Left to right
6	=, +=, -=	Right to left
7	==, !=, =&, =/	Left to right

C - GRASS PLUS LANGUAGE STRUCTURE

VII. Regular Definitions

Name	Definition
zero	{0}
dig	{1,2,3,4,5,6,7,8,9}
let	{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}
num	{zero, dig}
null	{λ}
dot	{.}
ascii	{any printable characters}
newline	{enter}
numulet	{num, let, underscore}
underscore	{_}
sep	{,}
ht	{#}
compar	{<, >, =, !}
arith	{+, -, *, /, %}
delimi	{compar, arith, space, ;,), (, sep, [,], dot}
delimtf	{compar, newline, space, arith, ;,),], }, sep, :}
delims	{”}
delimc	{’}

delimb	{space, compar, ;,),]}
delim1	{space, ;}
delim2	{newline, space}
delim3	{space, (,],), ;}
delim4	{space}
delim5	{num, space, (, “, h, [}
delim6	{let, num, space, newline, ht, (, [, {, },],), “, ‘, :, -}
delim7	{newline, dig, space, ht, “, ‘, (, [, {}
delim8	{let, num, ht, space, “, (, [, {}
delim9	{space, newline}
delim10	{;, space, newline, sep}
delim11	{;, +, space, sep,), },]}
delim12	{newline, space, =, ;,), }, sep,]}
delim13	{dig, space, (, ht}
delim14	{ascii}
delim15	{newline, space, dig, let, “,),], [}
delim16	{newline, let, ht, space,)}
delim17	{num, newline, space, arith, ht, =,],), }, sep, ;, ‘, (, dot}
delim18	{;, sep,],), }, dot}
delim19	{num, space, ht, (}
delim20	{let}

delim21	{space, :, (}
delim22	{num, space, (, ht}
delim23	{newline, space, “, }, (, ‘, dig}
delim24	{space, (}
delim25	{newline,), space, sep}
delim26	{newline,), space}
delim27	{space, num, ht, sep, underscore, [,), :}
delim28	{let, ht}

VIII. Regular Expressions

a. Reserved Words

Reserved Words	Regular Expressions	Tokens
Main Function		
garden	(g)(a)(r)(d)(e)(n)	garden
Start and End		
seed	(s)(e)(e)(d)	seed
plant	(p)(l)(a)(n)(t)	plant
Data Types		
tint	(t)(i)(n)(t)	tint
flora	(f)(l)(o)(r)(a)	flora
chard	(c)(h)(a)(r)(d)	chard
string	(s)(t)(r)(i)(n)(g)	string
bloom	(b)(l)(o)(o)(m)	bloom
florist	(f)(l)(o)(r)(i)(s)(t)	florist
tulip	(t)(u)(l)(i)(p)	tulip
dirt	(d)(i)(r)(t)	dirt
stem	(s)(t)(e)(m)	stem
Input and Output Statement		
inpetal	(i)(n)(p)(e)(t)(a)(l)	inpetal
mint	(m)(i)(n)(t)	mint

Conditional Statements		
leaf	(l)(e)(a)(f)	leaf
moss	(m)(o)(s)(s)	moss
eleaf	(e)(l)(e)(a)(f)	eleaf
nut	(n)(u)(t)	nut
true	(t)(r)(u)(e)	true
false	(f)(a)(l)(s)(e)	false
break	(b)(r)(e)(a)(k)	break
Iterative Statements		
fern	(f)(e)(r)(n)	fern
willow	(w)(i)(l)(l)(o)(w)	willow
Others		
args	(a)(r)(g)(s)	args
kwargs	(k)(w)(a)(r)(g)(s)	kwargs
at	(a)(t)	at
bare	(b)(a)(r)(e)	bare
branch	(b)(r)(a)(n)(c)(h)	branch
clear	(c)(l)(e)(a)(r)	clear
floral	(f)(l)(o)(r)(a)(l)	floral
getItems	(g)(e)(t)(I)(t)(e)(m)(s)	getItems
getKeys	(g)(e)(t)(K)(e)(y)(s)	getKeys

getValues	(g)(e)(t)(V)(a)(l)(u)(e)(s)	getValues
hard	(h)(a)(r)(d)	hard
lent	(l)(e)(n)(t)	lent
regrow	(r)(e)(g)(r)(o)(w)	regrow
tree	(t)(r)(e)(e)	tree
viola	(v)(i)(o)(l)(a)	viola

b. Reserved Symbols

Reserved Symbols	Regular Expressions	Tokens
Arithmetic operators		
+	(+)	+
-	(-)	-
*	(*)	*
/	(/)	/
%	(%)	%
**	(*)(*)	**
//	(/)(/)	//
Assignment operators		
=	(=)	=
+=	(+)(=)	+=
-=	(-)(=)	-=
*=	(*)(=)	*=
/=	(/)(=)	/=
%=	(%)(=)	%=
**=	(*)(*)(=)	**=
//=	(/)(/)(=)	//=
Comparison operators		

==	(=)(=)	==
!=	(!)(=)	!=
>	(>)	>
<	(<)	<
>=	(>)(=)	>=
<=	(<)(=)	<=
Logical operators		
=&	(=)(&)	=&
=/	(=)(/)	=/
=!	(=)(!)	=!
Others		
()	(()())	()
[]	([)(])	[]
{}	({)(})	{}
“	(“)	“
‘	(‘)	‘
,	(,)	,
:	(:)	:
.	(.)	.
;	(;)	;
#	(#)	#

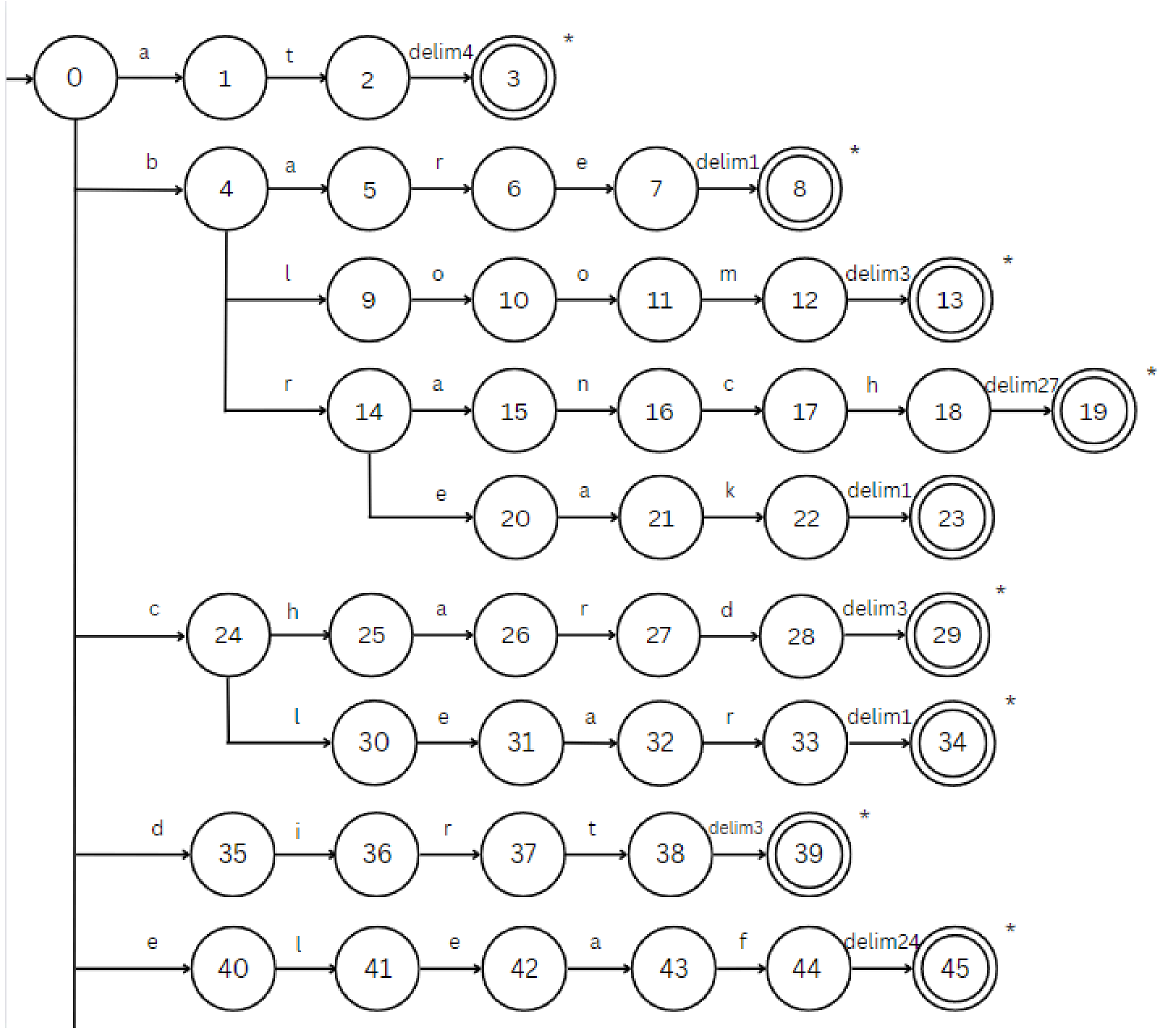
\	(\)	\
?	(?)	?
<--	(<)(-)(-)	<--
-->	(-)(-)(>)	-->
_	(_)	_

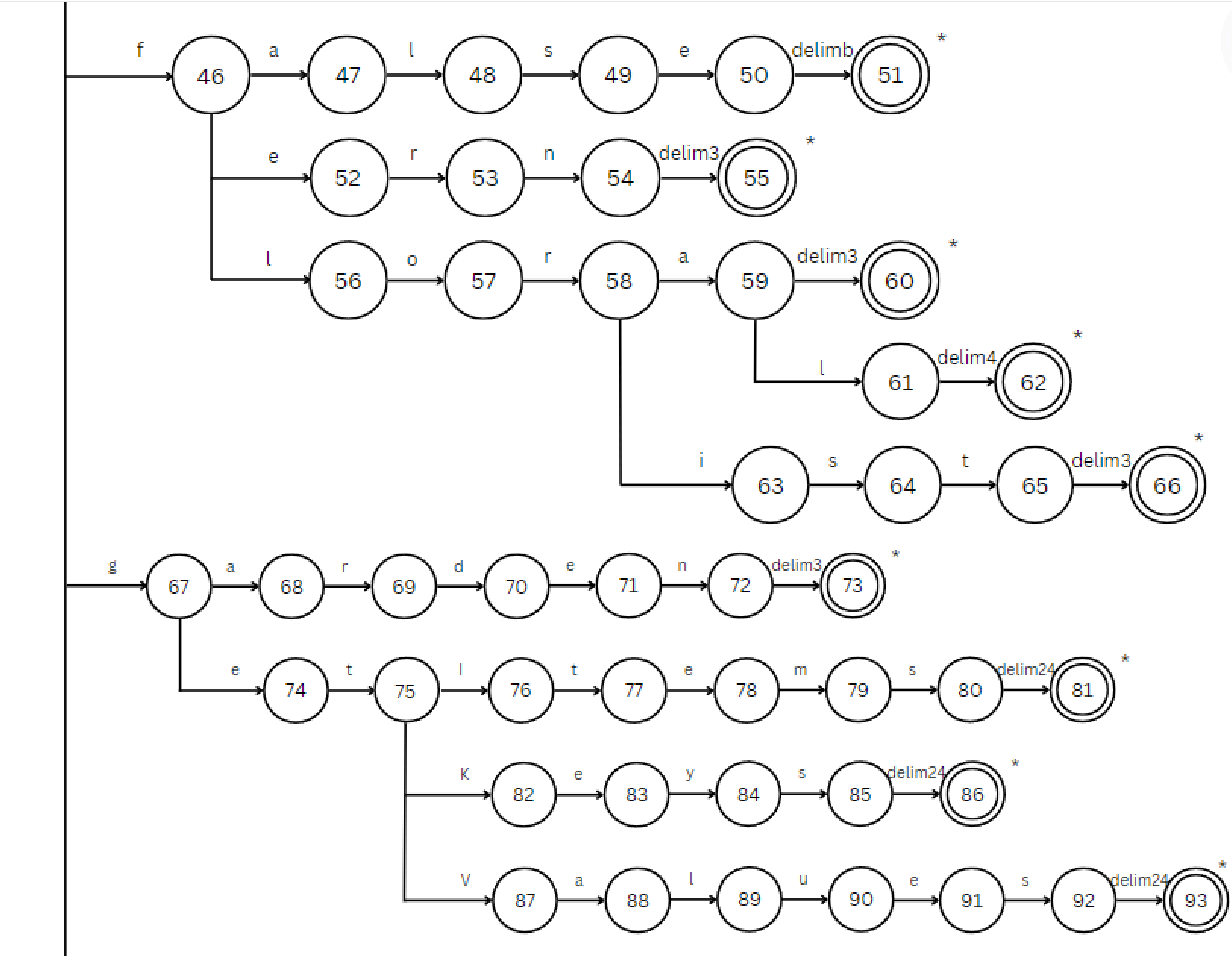
c. Literals

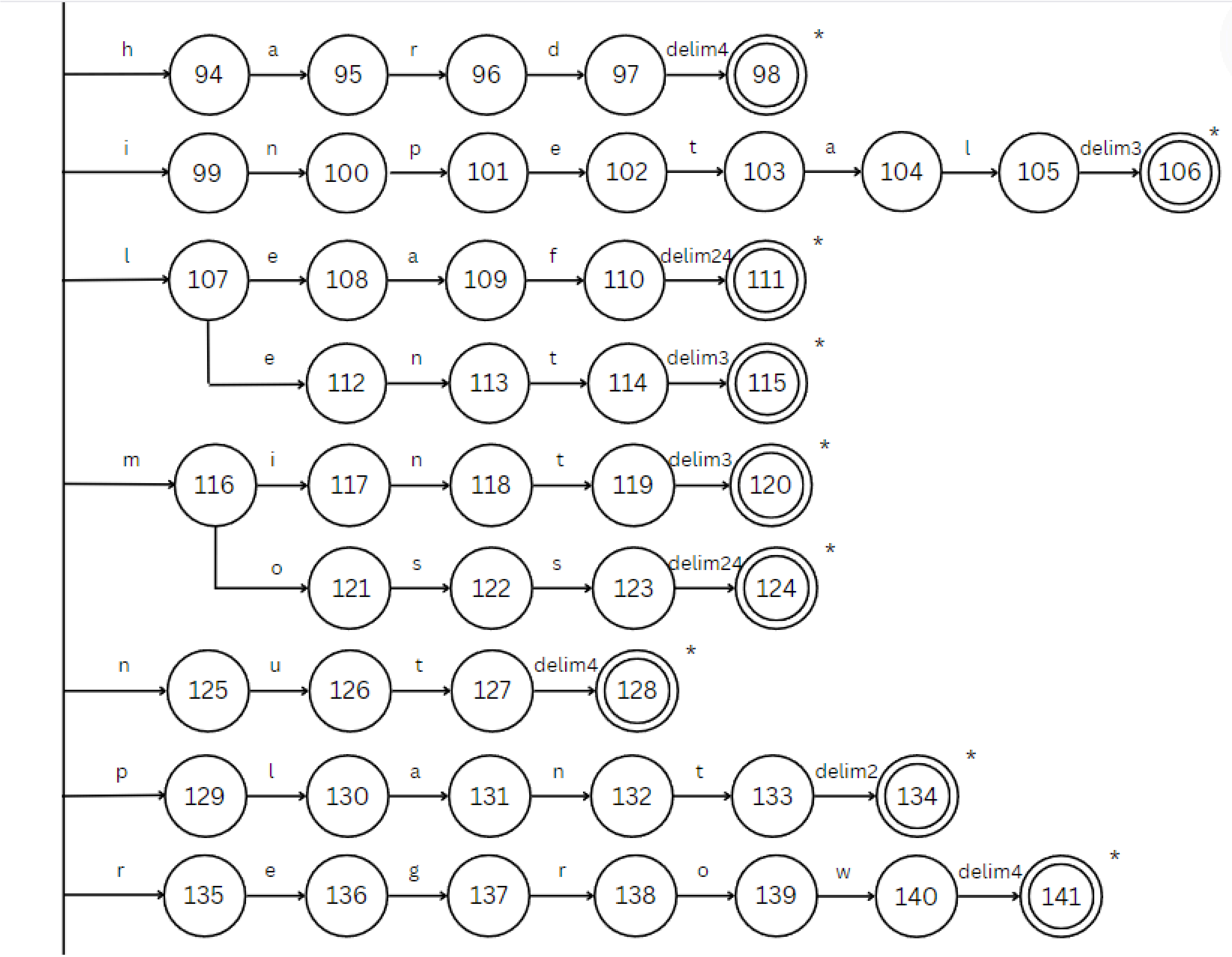
<table>
<tr><th>Literals</th><th>Regular Expressions</th><th>Token</th></tr>
<tr><td>identifier</td><td>(#)(let)(numulet | null)^49</td><td>Identifier</td></tr>
<tr><td>single-line comment</td><td>(?)(ascii)^∞</td><td>Single-line comment</td></tr>
<tr><td>multi-line comment</td><td>(<)(-)(-)(ascii)^∞(-)(-)(>)</td><td>Multi-line comments</td></tr>
<tr><td>tint literal</td><td>((-) | null) num(num | null)^5</td><td>tint literal</td></tr>
<tr><td>flora literal</td><td>((-) | null)num(num | null)^5 . (num | null)^6</td><td>flora literal</td></tr>
<tr><td>chard literal</td><td>(’)(ASCII | null)</td><td>chard literal</td></tr>
<tr><td>string literal</td><td>(“)(ASCII | null)$^+$</td><td>string literal</td></tr>
<tr><td>bloom literal</td><td>(t)(r)(u)(e) | (f)(a)(l)(s)(e) | (0) | (1)</td><td>bloom literal</td></tr>
</table>

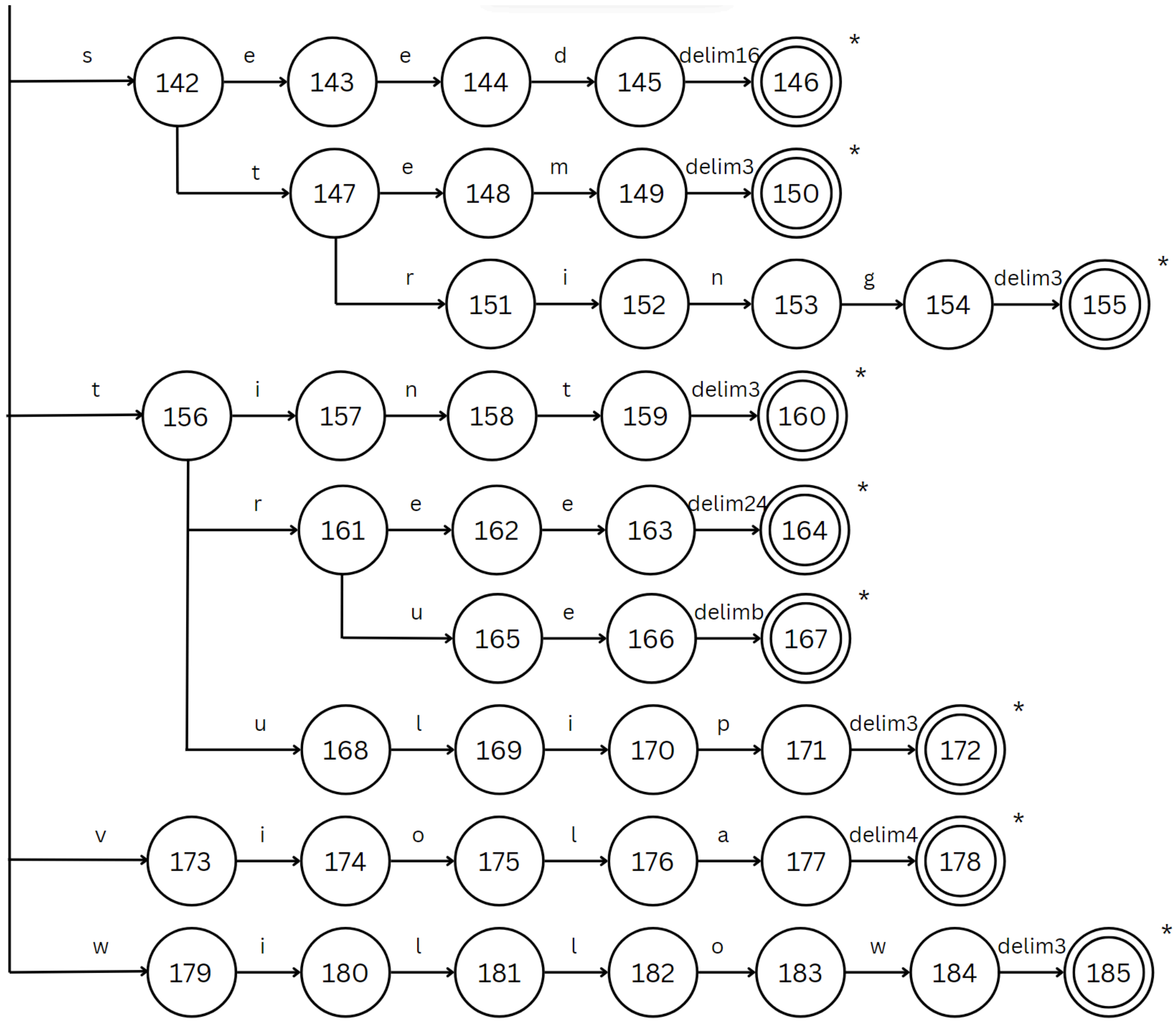
IX. Transition Diagram

a. Reserved Words

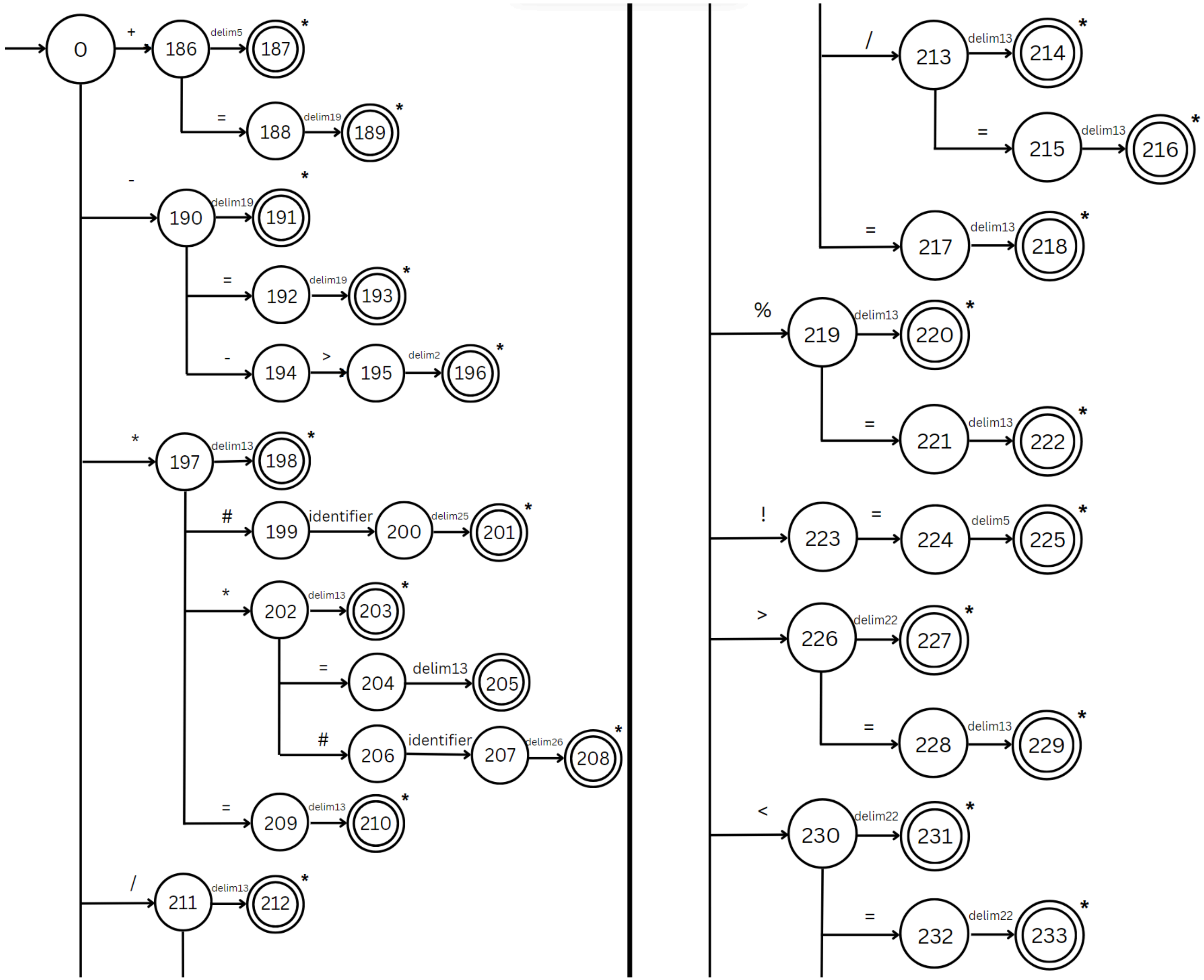


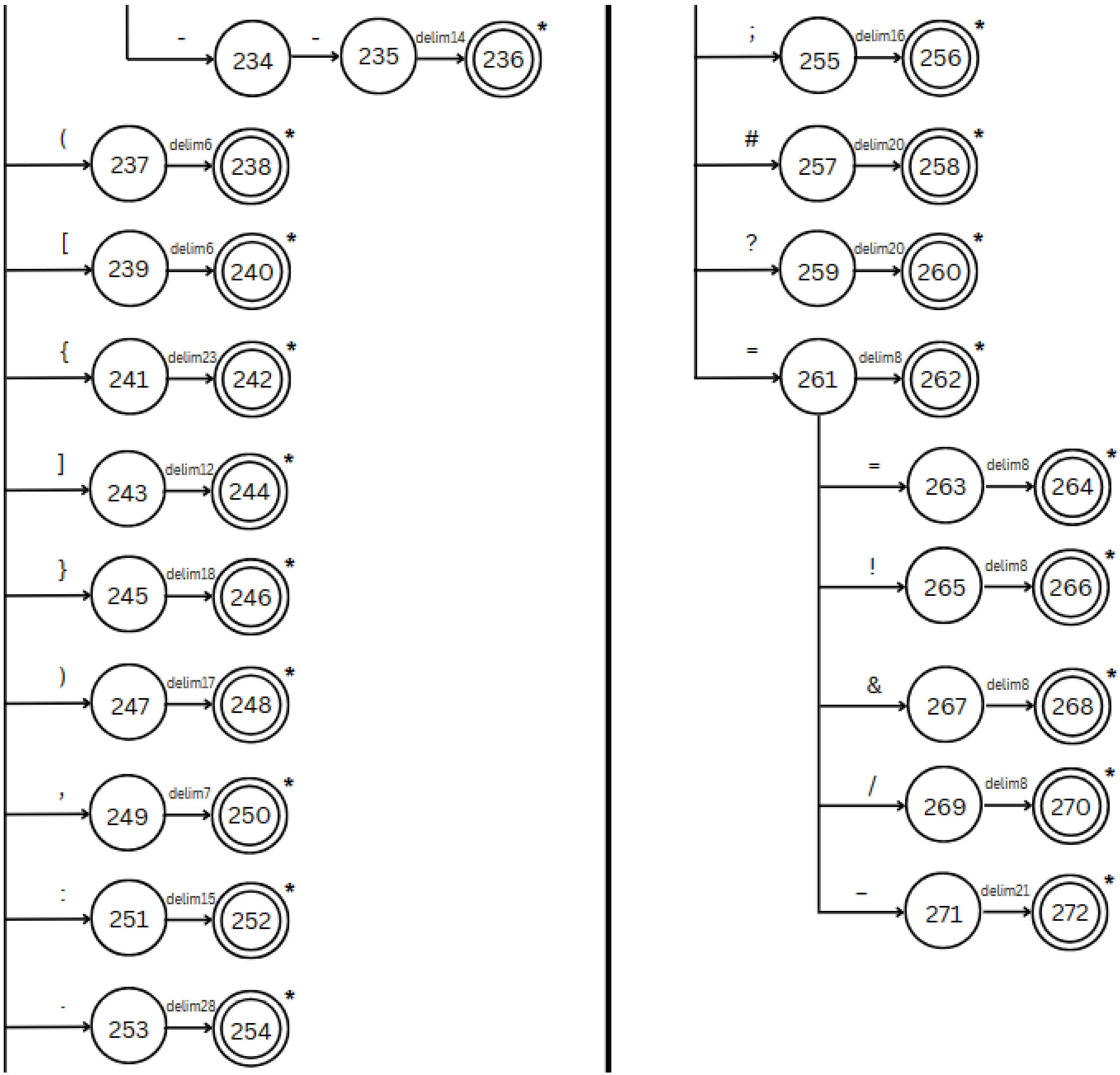

f
46
a
47
l
48
s
49
e
50
delimb
51
*
e
52
r
53
n
54
delim3
55
*
l
56
o
57
r
58
a
59
delim3
60
*
l
61
delim4
62
*
i
63
s
64
t
65
delim3
66
*
g
67
a
68
r
69
d
70
e
71
n
72
delim3
73
*
e
74
t
75
l
76
t
77
e
78
m
79
s
80
delim24
81
*
K
82
e
83
y
84
s
85
delim24
86
*
V
87
a
88
l
89
u
90
e
91
s
92
delim24
93
*


h
94
a
95
r
96
d
97
delim4
98
*
i
99
n
100
p
101
e
102
t
103
a
104
l
105
delim3
106
*
l
107
e
108
a
109
f
110
delim24
111
*
e
112
n
113
t
114
delim3
115
*
m
116
i
117
n
118
t
119
delim3
120
*
o
121
s
122
s
123
delim24
124
*
n
125
u
126
t
127
delim4
128
*
p
129
l
130
a
131
n
132
t
133
delim2
134
*
r
135
e
136
g
137
r
138
o
139
w
140
delim4
141
*


s
142
e
143
e
144
d
145
delim16
146
*
t
147
e
148
m
149
delim3
150
*
r
151
i
152
n
153
g
154
delim3
155
*
t
156
i
157
n
158
t
159
delim3
160
*
r
161
e
162
e
163
delim24
164
*
u
165
e
166
delimb
167
*
u
168
l
169
i
170
p
171
delim3
172
*
v
173
i
174
o
175
l
176
a
177
delim4
178
*
w
179
i
180
l
181
l
182
o
183
w
184
delim3
185
*

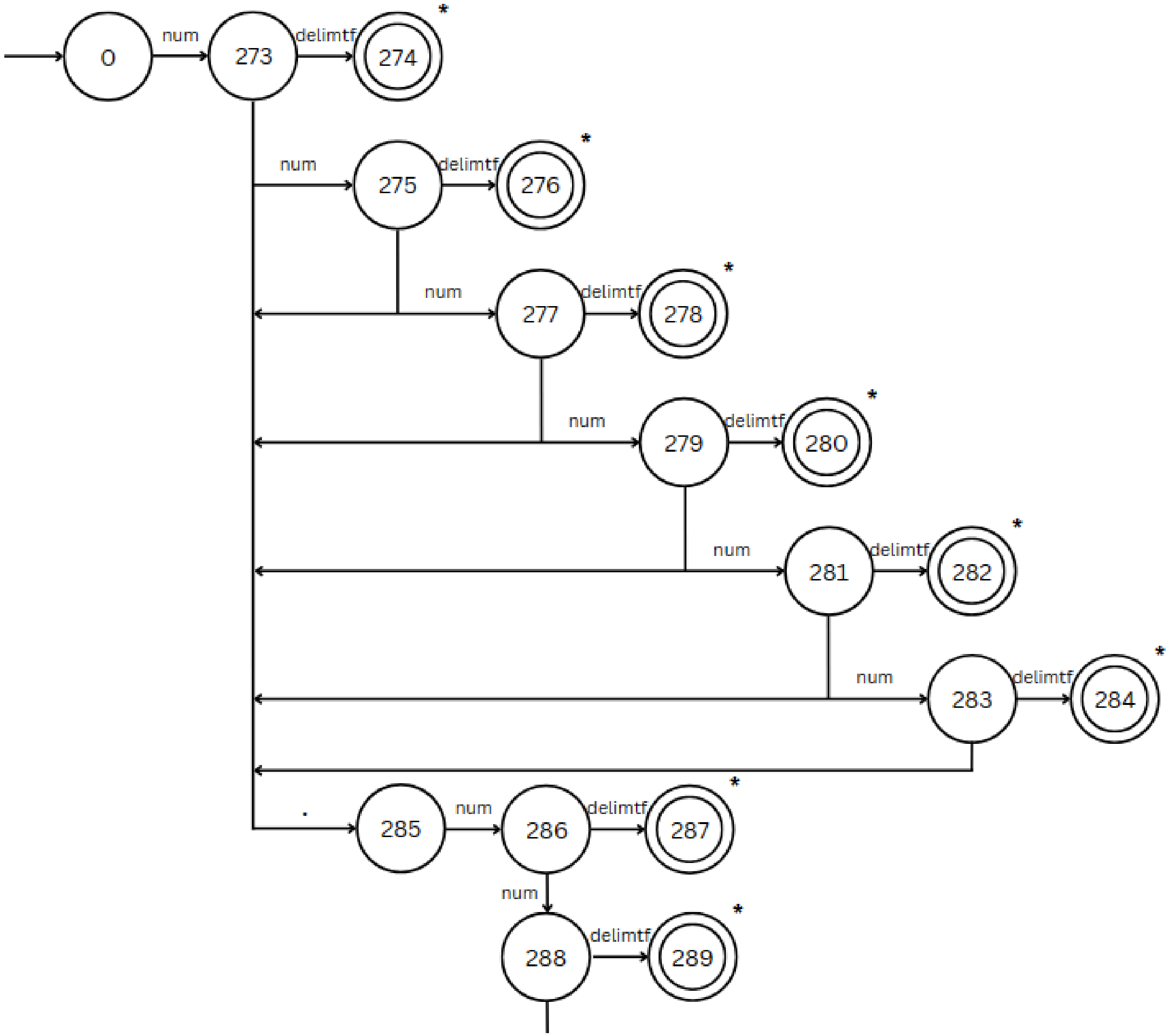
b. Reserved Symbols

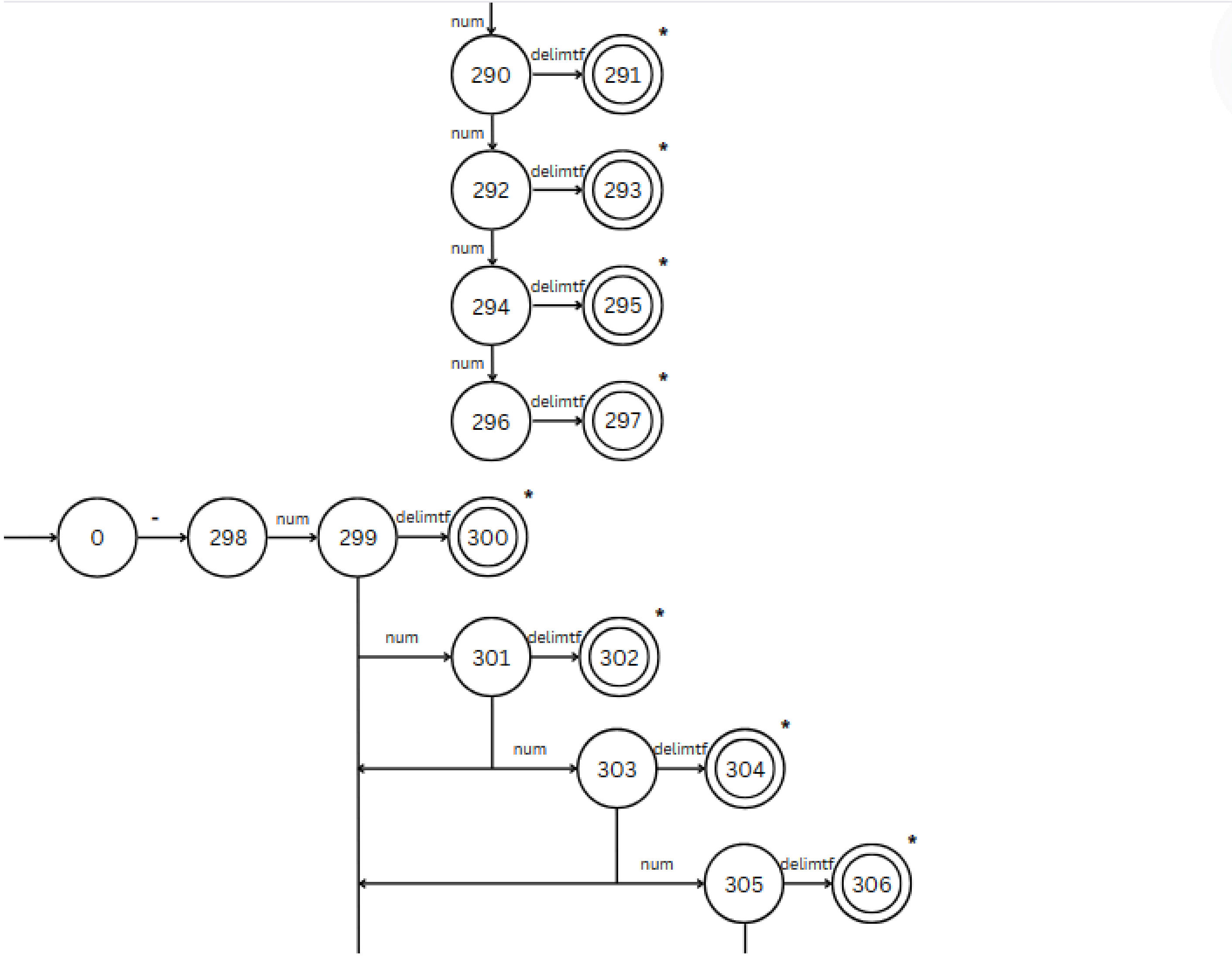


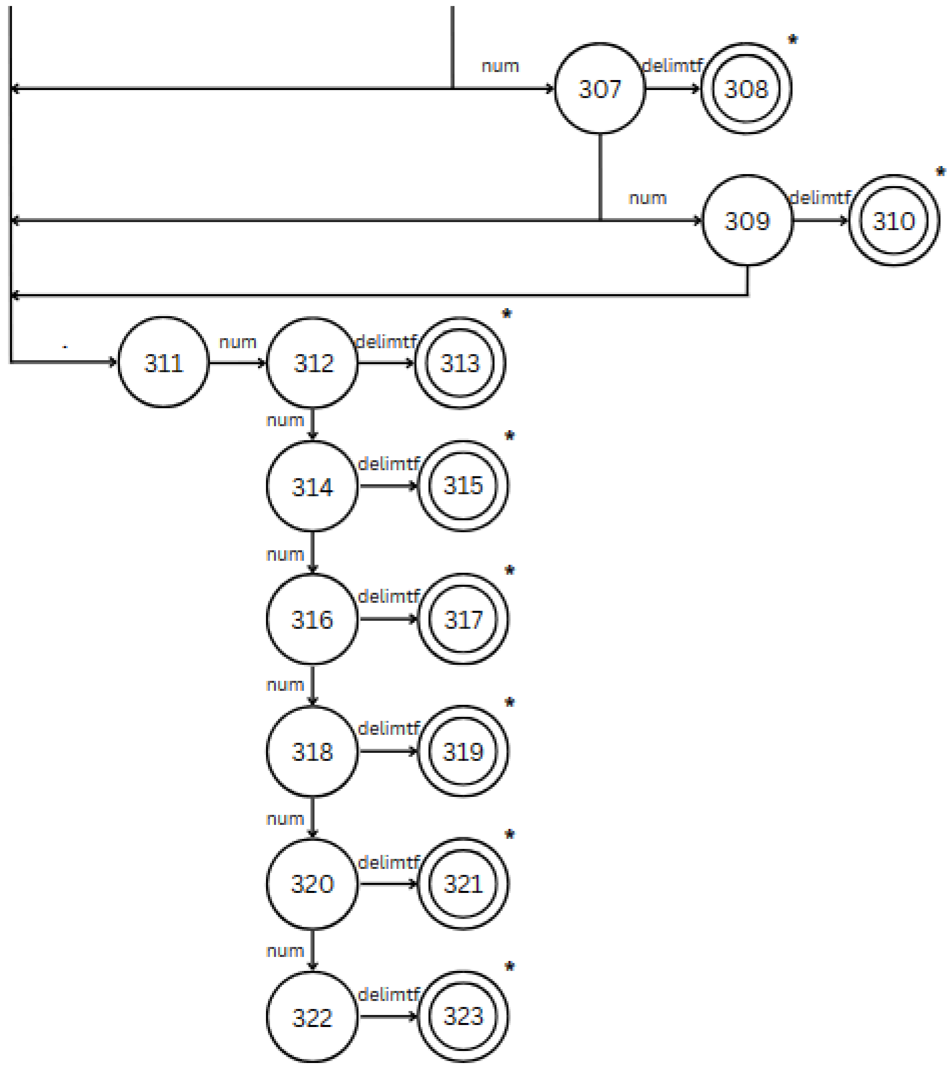

-
234
-
235
delim14
236
*
(
237
delim6
238
*
[
239
delim6
240
*
{
241
delim23
242
*
]
243
delim12
244
*
}
245
delim18
246
*
)
247
delim17
248
*
,
249
delim7
250
*
:
251
delim15
252
*
.
253
delim28
254
*
;
255
delim16
256
*
#
257
delim20
258
*
?
259
delim20
260
*
=
261
delim8
262
*
=
263
delim8
264
*
!
265
delim8
266
*
&
267
delim8
268
*
/
269
delim8
270
*
_
271
delim21
272
*

c. Literals

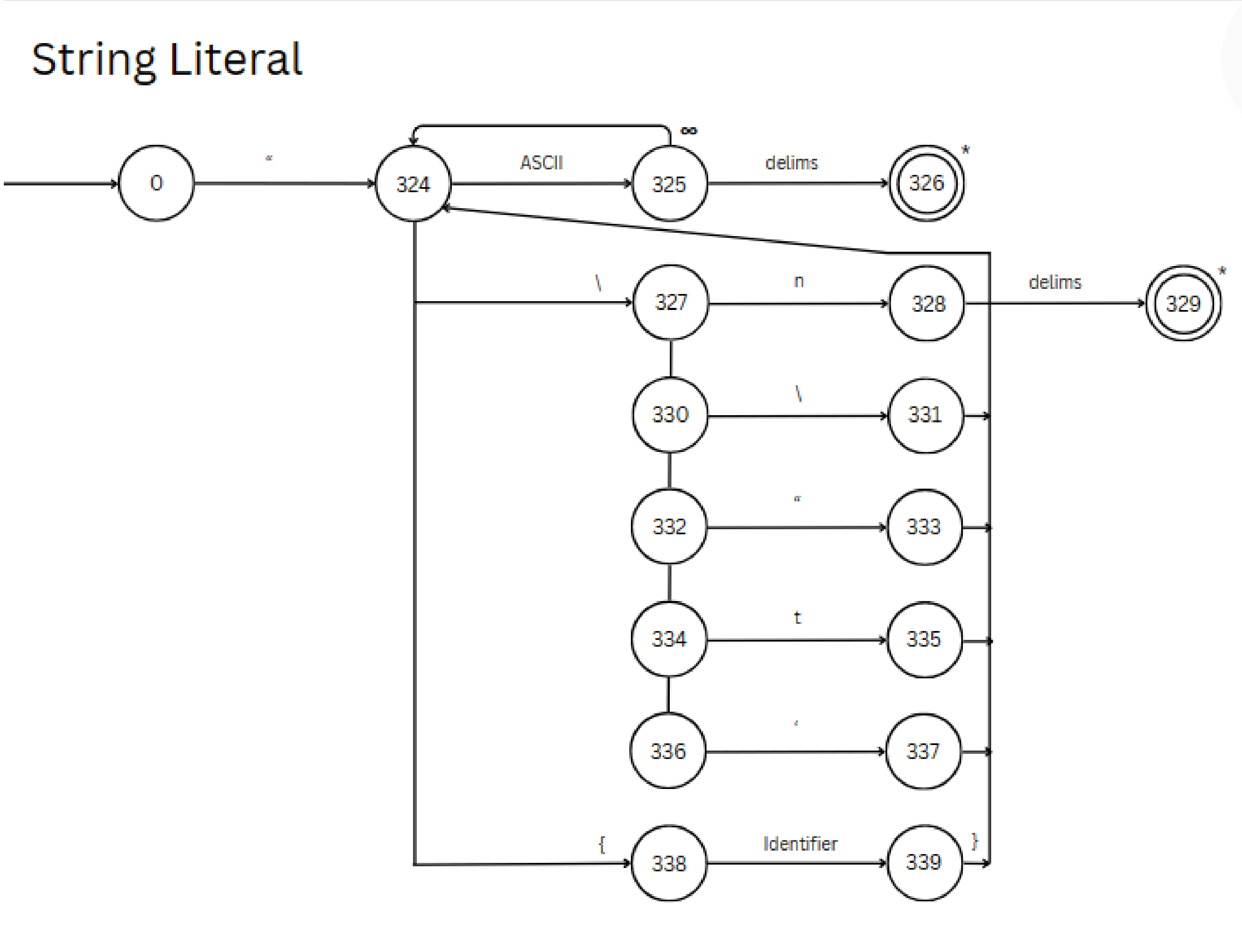
i. tint and flora literal



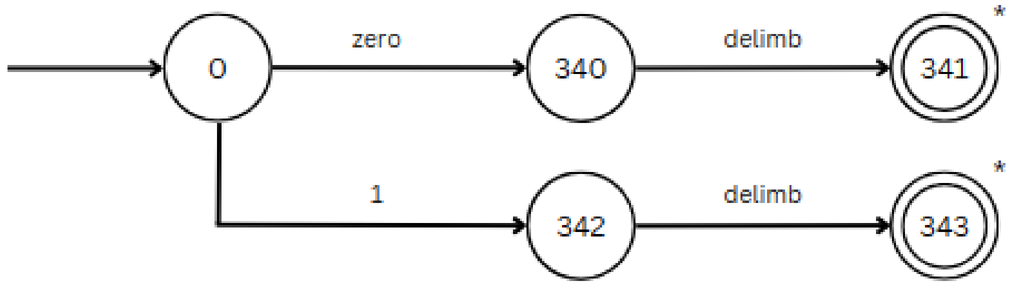

num
290
delimtf
291
*
num
292
delimtf
293
*
num
294
delimtf
295
*
num
296
delimtf
297
*
0
-
298
num
299
delimtf
300
*
num
301
delimtf
302
*
num
303
delimtf
304
*
num
305
delimtf
306
*


num
307
delimtf
308
*
num
309
delimtf
310
*
.
311
num
312
delimtf
313
*
num
314
delimtf
315
*
num
316
delimtf
317
*
num
318
delimtf
319
*
num
320
delimtf
321
*
num
322
delimtf
323
*

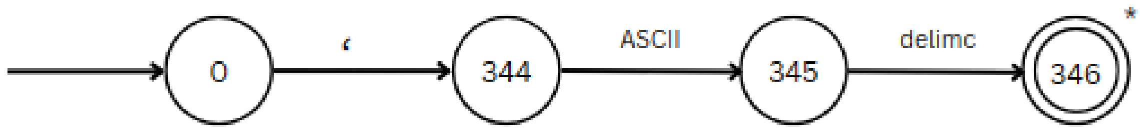
ii. string literal



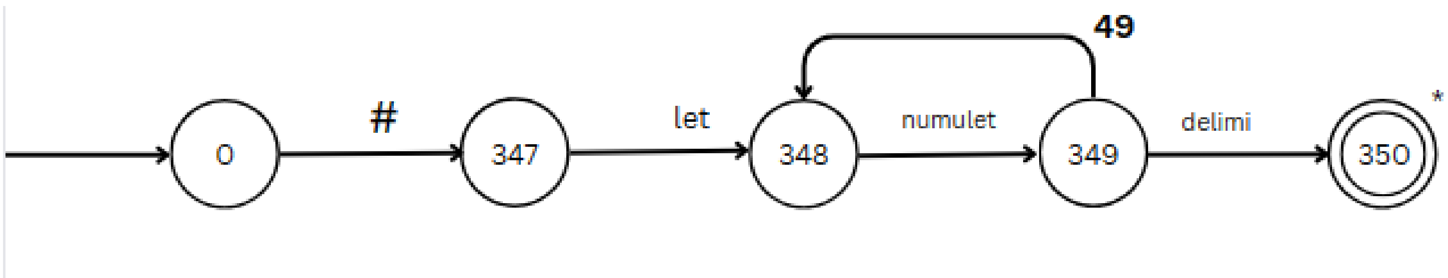
iii. bloom literal



iv. chard literal



v. identifier



X. Context Free Grammar

1	<program>	→	seed <global> garden() (<statement>); <function> plant
2	<global>	→	floral <constant> <insert-variable>; <global>
3	<global>	→	ε
4	<constant>	→	hard
5	<constant>	→	ε
6	<statement>	→	<constant> <insert-variable>; <statement>
7	<statement>	→	<i/o-statement>; <statement>
8	<statement>	→	leaf (<insert-condition>) (<filter-statement>); <eleaf> <else> <statement>
9	<statement>	→	<assignment>; <statement>
10	<statement>	→	<iterative>; <statement>
11	<statement>	→	tree (#identifier) (branch <check-branch>); <statement>
12	<statement>	→	clear; <statement>
13	<statement>	→	regrow <all-type-value> <add-at>;
14	<statement>	→	break;
15	<statement>	→	ε
16	<insert-variable>	→	<common-type> #identifier <common-data> <more-data>
17	<insert-variable>	→	<sqnc-type> #identifier <sqnc-value> <more-sqnc>
18	<common-type>	→	tint
19	<common-type>	→	flora

20	<common-type>	→	chard
21	<common-type>	→	string
22	<common-type>	→	bloom
23	<common-data>	→	= <insert-data>
24	<common-data>	→	ε
25	<insert-data>	→	<data>
26	<insert-data>	→	<open-parenthesis> <insert-operation>
27	<insert-operation>	→	<arithmetic> <close-parenthesis>
28	<insert-operation>	→	<condition> <close-parenthesis>
29	<data>	→	tint literal <operate-number>
30	<data>	→	flora literal <operate-number>
31	<data>	→	chard literal
32	<data>	→	string literal
33	<data>	→	bloom literal
34	<data>	→	#identifier <insert-func> <indexing> <start-end-step> <concatenate> <operate-number> <operate-logic>
35	<data>	→	lent (<all-type-value>) <operate-number>
36	<data>	→	<common-type> <typecast>
37	<data>	→	<supply-dirt> (<all-type-value>)
38	<data>	→	bare
39	<data>	→	ε
40	<open-parenthesis>	→	(<open-parenthesis>
41	<open-parenthesis>	→	ε
42	<close-parenthesis>	→	) <close-parenthesis>

43	<close-parenthesis>	→	ε
44	<arithmetic>	→	<tint> <operate-number>
45	<arithmetic>	→	<flora> <operate-number>
46	<operate-number>	→	<operator> <open-parenthesis> <arithmetic> <close-parenthesis>
47	<operate-number>	→	ε
48	<operator>	→	+
49	<operator>	→	-
50	<operator>	→	*
51	<operator>	→	/
52	<operator>	→	%
53	<operator>	→	**
54	<operator>	→	//
55	<tint>	→	tint literal
56	<tint>	→	lent (<all-type-value>)
57	<tint>	→	tint (<all-type-value>)
58	<tint>	→	#identifier <insert-func> <indexing>
59	<flora>	→	flora literal
60	<flora>	→	flora (<all-type-value>)
61	<flora>	→	#identifier <insert-func> <indexing>
62	<concatenate>	→	<indexing> + <all-type-value> <concatenate>
63	<concatenate>	→	ε
64	<condition>	→	<data> <operate-logic>
65	<condition>	→	<sequence> <operate-logic>
66	<operate-logic>	→	<cond-operator> <open-parenthesis>

			<condition> <close-parenthesis>
67	<operate-logic>	→	ε
68	<cond-operator>	→	==
69	<cond-operator>	→	!=
70	<cond-operator>	→	>
71	<cond-operator>	→	<
72	<cond-operator>	→	>=
73	<cond-operator>	→	<=
74	<cond-operator>	→	=&
75	<cond-operator>	→	=/
76	<cond-operator>	→	at
77	<cond-operator>	→	nut <nuts-and-ats>
78	<nuts-and-ats>	→	at
79	<nuts-and-ats>	→	nut <nuts-and-ats>
80	<nuts-and-ats>	→	ε
81	<supply-dirt>	→	getItems
82	<supply-dirt>	→	getKeys
83	<supply-dirt>	→	getValues
84	<insert-func>	→	(<argument>) <instance-grab>
85	<insert-func>	→	ε
86	<instance-grab>	→	.#identifier
87	<instance-grab>	→	ε
88	<indexing>	→	[<insert-index>] <indexing>
89	<indexing>	→	ε
90	<typecast>	→	(<all-type-value>) <concatenate> <operate-number> <operate-logic>

91	<typecast>	→	ε
92	<more-data>	→	, <common-type> #identifier <common-data> <more-data>
93	<more-data>	→	ε
94	<sqnc-type>	→	florist
95	<sqnc-type>	→	tulip
96	<sqnc-type>	→	dirt
97	<sqnc-type>	→	stem
98	<sqnc-value>	→	= <sequence>
99	<sqnc-value>	→	ε
100	<sequence>	→	<dirt> <open> <dirt> <insert-sqnc> <close>
101	<sequence>	→	<supply-dirt> (<all-type-value>)
102	<sequence>	→	<sqnc-type> <typecast>
103	<sequence>	→	#identifier <insert-func> <indexing> <start-end-step>
104	<open>	→	[
105	<open>	→	{
106	<dirt>	→	string literal :
107	<dirt>	→	ε
108	<close>	→	]
109	<close>	→	}
110	<more-sqnc>	→	, <sqnc-type> #identifier <sqnc-value> <more-sqnc>
111	<more-sqnc>	→	ε
112	<insert-sqnc>	→	<data> <next-sqnc>
113	<insert-sqnc>	→	<open> <insert-sqnc> <close>

			<next-sqnc>
114	<insert-sqnc>	→	*#identifier <add-kwargs>
115	<insert-sqnc>	→	ε
116	<next-sqnc>	→	, <insert-next-sqnc>
117	<next-sqnc>	→	ε
118	<insert-next-sqnc>	→	<dirt> <insert-sqnc>
119	<insert-next-sqnc>	→	*#identifier <add-kwargs>
120	<start-end-step>	→	[<insert-start>
121	<start-end-step>	→	ε
122	<insert-start>	→	<insert-data> : <close-start>
123	<insert-start>	→	: <skip-start>
124	<close-start>	→	<close-end>
125	<close-start>	→	<insert-data> <close-end>
126	<close-end>	→	] <start-end-step>
127	<close-end>	→	: <insert-data>] <start-end-step>
128	<skip-start>	→	<insert-data> <close-end> <start-end-step>
129	<skip-start>	→	: <insert-data>] <start-end-step>
130	<all-type-value>	→	<insert-data>
131	<all-type-value>	→	<sequence>
132	<all-type-value>	→	inpetal (string literal)
133	<i/o-statement>	→	<insert-inpetal> inpetal (string literal)
134	<i/o-statement>	→	mint (<all-type-value>)
135	<insert-inpetal>	→	<common-type> #identifier =
136	<insert-inpetal>	→	<sqnc-type> #identifier =
137	<insert-inpetal>	→	#identifier <insert-func> <indexing>

			<start-end-step> <more-id> <assignment-op>
138	<inpetal-state>	→	string literal
139	<inpetal-state>	→	ε
140	<more-id>	→	, #identifier <insert-func> <indexing> <start-end-step> <more-id>
141	<more-id>	→	ε
142	<eleaf>	→	eleaf (<condition>) (<statement>); <eleaf>
143	<eleaf>	→	ε
144	<else>	→	moss (<statement>);
145	<else>	→	ε
146	<assignment>	→	<insert-inpetal> <all-type-value>
147	<assignment>	→	<assign> <insert-assign>
148	<assign>	→	<insert-inpetal>
149	<assign>	→	ε
150	<insert-assign>	→	<common-type> (<all-type-value>)
151	<insert-assign>	→	<sqnc-type> (<all-type-value>)
152	<assignment-op>	→	=
153	<assignment-op>	→	+=
154	<assignment-op>	→	-=
155	<assignment-op>	→	*=
156	<assignment-op>	→	/=
157	<assignment-op>	→	%=
158	<assignment-op>	→	**=
159	<assignment-op>	→	//=

160	<iterative>	→	fern (<insert-fern>
161	<iterative>	→	willow (<condition>) (<statement>)
162	<insert-fern>	→	tint #identifier = tint literal; <condition>; #identifier <assignment-op> <tint>;) (<statement>)
163	<insert-fern>	→	<all-type-value> <more-value> at <sequence>;) (<statement>)
164	<more-value>	→	, <all-type-value> <more-value>
165	<more-value>	→	ε
166	<check-branch>	→	<all-type-value> <insert-branch> <more-branch>
167	<check-branch>	→	_: <statement>
168	<insert-branch>	→	: <operate-branch>
169	<insert-branch>	→	leaf (<condition>) (<statement>);
170	<operate-branch>	→	<statement>
171	<operate-branch>	→	branch <check-branch>
172	<more-branch>	→	branch <check-branch>
173	<more-branch>	→	ε
174	<argument>	→	<insert-argument>
175	<argument>	→	<common-type> #identifier <common-data> <more**kwargs>
176	<argument>	→	ε
177	<insert-argument>	→	<all-type-value> <add-argument>
178	<insert-argument>	→	#identifier (<argument>) <add-argument>
179	<insert-argument>	→	ε
180	<add-argument>	→	, <argument>
181	<add-argument>	→	ε

182	<more**kwargs>	→	, <common-type> #identifier <common-data> <more**kwargs>
183	<more**kwargs>	→	ε
184	<function>	→	<common-type> #identifier (<parameter>) (<statement>); <function>
185	<function>	→	viola #identifier (<undefined-param>) (<statement>); <function>
186	<function>	→	ε
187	<add-at>	→	<more-value> at <all-type-value>
188	<add-at>	→	ε
189	<parameter>	→	<undefined-param>
190	<parameter>	→	<common-type> #identifier <common-data> <next-parameter>
191	<parameter>	→	<sqnc-type> #identifier <sqnc-value> <next-parameter>
192	<parameter>	→	#identifier (<parameter>) <next-parameter>
193	<parameter>	→	ε
194	<undefined-param>	→	<common-type> *#identifier <add-kwargs>
195	<undefined-param>	→	**#identifier
196	<undefined-param>	→	ε
197	<add-kwargs>	→	, **#identifier
198	<add-kwargs>	→	ε
199	<next-parameter>	→	, <parameter>
200	<next-parameter>	→	ε

First Set			
1	<program>	→	{"seed"}
2	<global>	→	{"floral", ε}
3	<constant>	→	{"hard", ε}
4	<statement>	→	{"hard", "tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#", "mint", "leaf", "fern", "willow", "tree", "clear", “regrow”, "break", ε}
5	<insert-variable>	→	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem"}
6	<common-type>	→	{"tint", "flora", "chard", "string", "bloom"}
7	<common-data>	→	{“=”, ε}
8	<insert-data>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “(“}
9	<insert-operation>		{tint literal, “lent”, “tint”, “#”, flora literal, “flora”, chard literal, string literal, bloom literal,“chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
10	<data>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, ε}
11	<open-parenthesis>	→	{“(“, ε}
12	<close-parenthesis>	→	{“)“, ε}
13	<arithmetic>	→	{tint literal, "lent", “tint”, "#", "flora", flora literal}
14	<operate-number>	→	{"+", "-", "*", "/", "%", "**", "//", ε}
15	<operator>	→	{"+", "-", "*", "/", "%", "**", "//"}
16	<tint>	→	{tint literal, “lent”, “tint”, “#”}
17	<flora>	→	{flora literal, “flora”, “#”}
18	<concatenate>	→	{"[", "+", ε}

19	<condition>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
20	<operate-logic>	→	{“==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, ε}
21	<cond-operator>	→	{“==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”}
22	<nuts-and-ats>	→	{“at”, “nut”, ε}
23	<supply-dirt>	→	{“getItems”, “getKeys”, “getValues”}
24	<insert-func>	→	{“(“, ε}
25	<instance-grab>	→	{".", ε}
26	<indexing>	→	{"[", ε}
27	<typecast>	→	{“(“}
28	<more-data>	→	{“,”, ε}
29	<sqnc-type>	→	{“florist”, “tulip”, “dirt”, “stem”}
30	<sqnc-value>	→	{“=”, ε}
31	<sequence>	→	{string literal, “[“, “{“, “getItems”, “getKeys”, “getValues”, “florist”, “tulip”, “dirt”, “stem”}
32	<open>	→	{"[", "{"}
33	<dirt>	→	{string literal, ε}
34	<close>	→	{"]", "}"}
35	<more-sqnc>	→	{“,”, ε}
36	<insert-sqnc>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“}
37	<next-sqnc>	→	{“,”, ε}
38	<insert-next-sqnc>	→	{string literal, ε, “*”}
39	<start-end-step>	→	{“[“, ε}
40	<insert-start>	→	{tint literal, flora literal, chard literal, string

			literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “(“, “:”}
41	<close-start>	→	{“]”, “:”, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “(“}
42	<close-end>	→	{“]”, “:”}
43	<skip-start>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “(“, “:”}
44	<all-type-value>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”, “inpetal”}
45	<i/o-statement>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, “mint”}
46	<insert-inpetal>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”}
47	<inpetal-state>	→	{string literal, ε}
48	<more-id>	→	{“,”, ε}
49	<eleaf>	→	{“eleaf”, ε}
50	<else>	→	{“moss”, ε}
51	<assignment>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”}
52	<assign>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, ε}
53	<insert-assign>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”}
54	<assignment-op>	→	{“=”, “+=”, “-=”, “*=”, “/=”, “%=”, “**=”, “//=”}
55	<iterative>	→	{“fern”, “willow”}
56	<insert-fern>	→	{“tint”, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“,

			“florist”, “tulip”, “dirt”, “stem”, “inpetal”}
57	<more-value>	→	{“,”, ε}
58	<check-branch>	→	{tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”, “inpetal”, “_:”}
59	<insert-branch>	→	{“:”, “leaf”}
60	<operate-branch>	→	{“hard”, “tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, “mint”, “leaf”, “fern”, “willow”, “tree”, “clear”, “break”, ε, “_:”}
61	<more-branch>	→	{“branch”, ε}
62	<argument>	→	{“tint”, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”, “inpetal”, ε}
63	<insert-argument>	→	{“tint”, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”, “inpetal”, ε}
64	<add-argument>	→	{“,”, ε}
65	<more**kwargs>	→	{“,”, ε}
66	<function>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “viola”, “#”}
67	<add-at>	→	{“,”, “at”, ε}
68	<parameter>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “**”, “florist”, “tulip”, “dirt”, “stem”, “#”}
69	<undefined-param>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “**”, ε}
70	<add-kwargs>	→	{“,”, ε}
71	<next-parameter>	→	{“,”, ε}

Follow Set			
1	<program>	→	{ $ }
2	<global>	→	{"garden"}
3	<constant>	→	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem"}
4	<statement>	→	{")"}
5	<insert-variable>	→	{";"}
6	<common-type>	→	{“#”, “(“, “*”}
7	<common-data>	→	{“,”, $, “;”, “)”}
8	<insert-data>	→	{“,”, $, “;”, “)”}
9	<insert-operation>	→	{“,”, $, “;”, “)”}
10	<data>	→	{“,”, $, “;”, “)”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”}
11	<open-parenthesis>	→	{tint literal, “lent”, “tint”, “#”, flora literal, “flora”, chard literal, string literal, bloom literal, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
12	<close-parenthesis>	→	{“,”, $, “;”, “)”}
13	<arithmetic>	→	{“)”, $, “,”, “;”}
14	<operate-number>	→	{“)”, $, “,”, “;”}
15	<operator>	→	{“(“, tint literal, “lent”, “tint”, “#”, flora literal, “flora”}
16	<tint>	→	{“+”, “-”, “*”, “/”, “%”, “**”, “//”, $, “)”, “,”, “;”}
17	<flora>	→	{“+”, “-”, “*”, “/”, “%”, “**”, “//”, $, “)”, “,”, “;”}
18	<concatenate>	→	{“+”, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, $, “,”, “;”, “)”}
19	<condition>	→	{“)”, $, “,”, “;”}
20	<operate-logic>	→	{“)”, $, “,”, “;”}
21	<cond-operator>	→	{“(“, $, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”,

			“chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
22	<nuts-and-ats>	→	{“(“, $, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
23	<supply-dirt>	→	{“(“}
24	<insert-func>	→	{"[“, $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ";", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
25	<instance-grab>	→	{"[“, $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ";", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
26	<indexing>	→	{"[“, $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ";", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
27	<typecast>	→	{“,”, $, “;”, “)”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”}
28	<more-data>	→	{“;”}
29	<sqnc-type>	→	{“#”, “(“}
30	<sqnc-value>	→	{“,”, $, “;”}
31	<sequence>	→	{“==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, $}
32	<open>	→	{string literal, tint literal, flora literal, chard literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“}
33	<dirt>	→	{“[“, “{“, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “]“, “}“}
34	<close>	→	{“==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, $, “,”, “]”, “}”}
35	<more-sqnc>	→	{“;”}

36	<insert-sqnc>	→	{“]”, “}”}
37	<next-sqnc>	→	{“]”, “}”}
38	<insert-next-sqnc>	→	{“]”, “}”}
39	<start-end-step>	→	{“[“, “+”, $, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”, “)", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
40	<insert-start>	→	{“[“, “+”, $, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”, “)", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
41	<close-start>	→	{“[“, “+”, $, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”, “)", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
42	<close-end>	→	{“[“, “+”, $, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”, “)", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
43	<skip-start>	→	{“[“, “+”, $, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”, “)", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
44	<all-type-value>	→	{“)”, “[“, “+”, $ “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, “,”, “;”}
45	<i/o-statement>	→	{“;”}
46	<insert-inpetal>	→	{“inpetal”}
47	<inpetal-state>	→	{“)”}
48	<more-id>	→	{“=”, “+=”, “-=”, “*=”, “/=”, “%=”, “**=”, “//=”}
49	<eleaf>	→	{“moss”, “hard”, “tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, “mint”, “leaf”, “fern”, “willow”, “tree”, “clear”, “break”, $, “)”, “regrow”}
50	<else>	→	{“hard”, “tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, “mint”, “leaf”, “fern”, “willow”, “tree”, “clear”, “break”, $, “)”, “regrow”}

51	<assignment>	→	{“;”}
52	<assign>	→	{“tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”}
53	<insert-assign>	→	{“;”}
54	<assignment-op>	→	{“inpetal”, tint literal, “lent”, “tint”, “#”}
55	<iterative>	→	{“;”}
56	<insert-fern>	→	{“;”}
57	<more-value>	→	{“at”}
58	<check-branch>	→	{“)”}
59	<insert-branch>	→	{“branch, “)”, $}
60	<operate-branch>	→	{“branch, “)”, $}
61	<more-branch>	→	{“)”}
62	<argument>	→	{“)”}
63	<insert-argument>	→	{“)”}
64	<add-argument>	→	{“)”}
65	<more**kwargs>	→	{“)”}
66	<function>	→	{“plant”}
67	<add-at>	→	{“;”}
68	<parameter>	→	{“)”}
69	<undefined-param>	→	{“)”}
70	<add-kwargs>	→	{“)”}
71	<next-parameter>	→	{“)”}

Predict Set			
1	First(seed <global> garden() (<statement>); <function> plant)	First(seed)	{"seed"}
2	First(floral <constant> <insert-variable>; <global>)	First(floral)	{"floral"}
3	First(ε) U Follow(<global>)	Follow(<global>)	{"garden"}
4	First(hard)	First(hard)	{"hard"}
5	First(ε) U Follow(<constant>)	Follow(<constant>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem"}
6	First(<constant> <insert-variable>; <statement>)	First(<constant>)	{"hard", $}
7	First(<i/o-statement>; <statement>)	First(<i/o-statement>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#", "mint"}
8	First(leaf (<insert-condition>) (<statement>); <eleaf> <else> <statement>)	First(leaf)	{"leaf"}
9	First(<assignment>; <statement>)	First(<assignment>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#"}
10	First(<iterative>; <statement>)	First(<iterative>)	{"fern", "willow"}
11	First(tree (#identifier) (branch <check-branch>); <statement>)	First(tree)	{“tree”}
12	First(clear; <statement>)	First(clear)	{"clear"}
13	First(regrow <all-type-value> <add-at>;)	First(regrow)	{“regrow”}
14	First(break;)	First(break)	{"break"}
15	First(ε) U Follow(<statement>)	Follow(<statement>)	{")", "regrow"}
16	First(<common-type> #identifier <common-data> <more-data>)	First(<common-type>)	{"tint", "flora", "chard", "string", "bloom"}
17	First(<sqnc-type> #identifier <sqnc-value> <more-sqnc>)	First(<sqnc-type>)	{"florist", "tulip", "dirt", "stem"}
18	First(tint)	First(tint)	{"tint"}

19	First(flora)	First(flora)	{"flora"}
20	First(chard)	First(chard)	{"chard"}
21	First(string)	First(string)	{"string"}
22	First(bloom)	First(bloom)	{"bloom"}
23	First(= <insert-data>)	First(=)	{"="}
24	First(ε) U Follow(<common-data>)	Follow(<common-data>)	{",", $, ";", ")"}
25	First(<data>)	First(<data>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare"}
26	First(<open-parenthesis> <insert-operation>)	First(<open-parenthesis>)	{"(", $}
27	First(<arithmetic> <close-parenthesis>)	First(<arithmetic>)	{tint literal, "lent", "tint", "#", flora literal, "flora"}
28	First(<condition> <close-parenthesis>)	First(<condition>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem"}
29	First(tint literal <operate-number>)	First(tint literal)	{tint literal}
30	First(flora literal <operate-number>)	First(flora literal)	{flora literal}
31	First(chard literal)	First(chard literal)	{chard literal}
32	First(string literal)	First(string literal)	{string literal}
33	First(bloom literal)	First(bloom literal)	{bloom literal}
34	First(#identifier <insert-func> <indexing> <start-end-step> <concatenate> <operate-number> <operate-logic>)	First(#)	{"#"}
35	First(lent (<all-type-value>) <operate-number>)	First(lent)	{"lent"}

36	First(<common-type> (<all-type-value>) <concatenate> <operate-number> <operate-logic>)	First(<common-type>)	{"tint", "flora", "chard", "string", "bloom"”}
37	First(<supply-dirt> (<all-type-value>))	First(<supply-dirt>)	{"getItems", "getKeys", "getValues"}
38	First(bare)	First(bare)	{“bare”}
39	First(ε) U Follow(<data>)	Follow(<data>)	{“,”, $, “;”, “)”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”}
40	First((<open-parenthesis>)	First(( )	{“(“}
41	First(ε) U Follow(<open-parenthesis>)	Follow(<open-parenthesis>)	{tint literal, "lent", "tint", "#", flora literal, "flora", chard literal, string literal, bloom literal, "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem"
42	First( ) <close-parenthesis>)	First( ))	{“)”}
43	First(ε) U Follow(<close-parenthesis>)	Follow(<close-parenthesis>)	{",", $, ";", ")"}
44	First(<tint> <operate-number>)	First(<tint>)	{tint literal, "lent", "tint", "#"}
45	First(<flora> <operate-number>)	First(<flora>)	{flora literal, "flora", "#"}
46	First(<operator> <open-parenthesis> <arithmetic> <close-parenthesis>)	First(<operator>)	{"+", "-", "*", "/", "%", "**", "//"}
47	First(ε) U Follow(<operate-number>)	Follow(<operate-number>)	{")", $, ",", ";"}
48	First(+)	First(+)	{"+"}
49	First(-)	First(-)	{"-"}
50	First(*)	First(*)	{"*"}
51	First(/)	First(/)	{"/"}
52	First(%)	FIrst(%)	{"%"}
53	First(**)	First(**)	{"**"}
54	First(//)	First(//)	{"//"}
55	First(tint literal)	First(tint literal)	{tint literal}

56	First(lent (<all-type-value>))	First(lent)	{“lent”}
57	First(tint (<all-type-value>))	First(tint)	{“tint”}
58	First(#identifier <insert-func> <indexing>)	First(#)	{“#”}
59	First(flora literal)	First(flora literal)	{flora literal}
60	First(flora (<all-type-value>))	First(flora)	{“flora”}
61	First(#identifier <insert-func> <indexing>)	First(#)	{“#”}
62	First(<indexing> + <all-type-value> <concatenate>)	First(<indexing>)	{“[“, $}
63	First(ε) U Follow(<concatenate>)	Follow(<concatenate>)	{“+”, “-”, “*”, “/”, “%”, “**”, “//”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”, $, “,”, “;”, “)"}
64	First(<data> <operate-logic>)	First(<data>)	{tint literal, "flora literal", "chard literal", "string literal", "bloom literal", "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare"}
65	First(<sequence> <operate-logic>)	First(<sequence>)	{string literal, "[", "{", "getItems", "getKeys", "getValues", "florist", "tulip", "dirt", "stem", "#"}
66	First(<cond-operator> <open-parenthesis> <condition> <close-parenthesis>)	First(<cond-operator>)	{"==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut"}
67	First(ε) U Follow(<operate-logic>)	Follow(<operate-logic>)	{"(", $, tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem"}
68	First(==)	First(==)	{“==”}
69	First(!=)	First(!=)	{“!=”}
70	First(>)	First(>)	{“>”}

71	First(<)	First(<)	{“<”}
72	First(>=)	First(>=)	{“>=”}
73	First(<=)	First(<=)	{“<=”}
74	First(=&)	First(=&)	{“=&”}
75	First(=/)	First(=/)	{“=/”}
76	First(at)	First(at)	{“at”}
77	First(nut <nuts-and-ats>)	First(nut)	{“nut”}
78	First(at)	First(at)	{“at”}
79	First(nut <nuts-and-ats>)	First(nut)	{“nut”}
80	First(ε) U Follow(<nuts-and-ats>)	Follow(<nuts-and-ats>)	{“(“, ε, tint literal, flora literal, chard literal, string literal, bloom literal, “#”, “lent”, “tint”, “flora”, “chard”, “string”, “bloom”, “getItems”, “getKeys”, “getValues”, “bare”, “[“, “{“, “florist”, “tulip”, “dirt”, “stem”}
81	First(getItems)	First(getItems)	{“getItems”}
82	First(getKeys)	First(getKeys)	{“getKeys”}
83	First(getValues)	First(getValues)	{“getValues”}
84	First((<argument>) <instance-grab>)	First(( )	{“(“}
85	First(ε) U Follow(<insert-func>)	Follow(<insert-func>)	{"[", $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ":", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
86	First(.#identifier)	First(.)	{“.”}
87	First(ε) U Follow(<instance-grab>)	Follow(<instance-grab>)	{"[", $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ":", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
88	First([<insert-index>] <indexing>)	First([ )	{“[“}

89	First(ε) U Follow(<indexing>)	Follow(<indexing>)	{"[", $, ",", ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "+", "-", "*", "/", "%", "**", "//", ",", ";", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
90	First((<all-type-value>) <concatenate> <operate-number> <operate-logic>)	First(( )	{“(“}
91	First(ε) U Follow(<typecast>)	Follow(<typecast>)	{“,”, $, “;”, “)”, “==”, “!=”, “>”, “<”, “>=”, “<=”, “=&”, “=/”, “at”, “nut”}
92	First(, <common-type> #identifier <common-data> <more-data>)	First(,)	{“,”}
93	First(ε) U Follow (<more-data>)	Follow(<more-data>)	{“;”}
94	First(florist)	First(florist)	{“florist”}
95	First(tulip)	First(tulip)	{“tulip”}
96	First(dirt)	First(dirt)	{“dirt”}
97	First(stem)	First(stem)	{“stem”}
98	First(= <sequence> <concatenate>)	First(=)	{“=”}
99	First(ε) U Follow(<sqnc-value>)	Follow(<sqnc-value>)	{“,”, $, “;”}
100	First(<dirt> <open> <dirt> <insert-sqnc> <close>)	First(<dirt>)	{string literal, $}
101	First(<supply-dirt> (<all-type-value>))	First(<supply-dirt>)	{"getItems", "getKeys", "getValues"}
102	First(<sqnc-type> <typecast>)	First(<sqnc-type>)	{"florist", "tulip", "dirt", "stem"}
103	First(#identifier <insert-func> <indexing> <start-end-step>)	First(#)	{“#”}
104	First([ )	First([ )	{“[“}
105	First({)	First({)	{“{“}
106	First(string literal)	First([ )	{string literal}
107	First(ε) U Follow(<dirt>)	Follow(<dirt>)	{"[", "{", tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard",

			"string", "bloom", "getItems", "getKeys", "getValues", "bare", "]", "}"}
108	First( ])	First( ])	{"]"}
109	First(})	First(})	{"}"}
110	First(, <sqnc-type> #identifier <sqnc-value> <more-sqnc>)	First(,)	{","}
111	First(ε) U Follow(<more-sqnc>)	Follow(<more-sqnc>)	{";"}
112	First(<data> <next-sqnc>)	First(<data>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare"}
113	First(<open> <insert-sqnc> <close> <next-sqnc>)	First(<open>)	{"[", "{"}
114	First(*#identifier <add-kwargs>)	First(*)	{"*"}
115	First(ε) U Follow(<insert-sqnc>)	Follow(<insert-sqnc>)	{"]", "}"}
116	First(, <insert-next-sqnc>)	First(,)	{","}
117	First(ε) U Follow(<next-sqnc>)	Follow(<next-sqnc>)	{"]", "}"}
118	First(<dirt> <insert-sqnc>)	First(<dirt>)	{string literal, ε}
119	First(*#identifier <add-kwargs>)	First(*)	{"*"}
120	First([<insert-start>)	First([ )	{"["}
121	First(ε) U Follow(<start-end-step>)	Follow(<start-end-step>)	{",", $, ";", ")", "==", "!=", ">", "<", ">=", "<=", "=&", "=/", "at", "nut", "=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
122	First(tint literal : <close-start>)	First(tint literal)	{tint literal}
123	Frst(: <skip-start>)	First(:)	{":"}
124	First(<close-end>)	First(<close-end>)	{"]", ":"}
125	First(tint literal <close-end>)	First(tint literal)	{tint literal}
126	First( ] <start-end-step>)	First( ])	{"]"}
127	First(: tint literal] <start-end-step>)	First(:)	{":"}

128	First(tint literal <close-end> <start-end-step>)	First(tint literal)	{tint literal}
129	First(: tint literal] <start-end-step>)	First(:)	{“:”}
130	First(<insert-data>)	First(<insert-data>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "("}
131	First(<sequence> <concatenate>)	First(<sequence>)	{string literal, "[", "{", "getItems", "getKeys", "getValues", "florist", "tulip", "dirt", "stem", "#"}
132	First(inpetal (string literal) <concatenate> <operate-number> <operate-logic>)	First(inpetal)	{"inpetal"}
133	First(<insert-inpetal> inpetal (string literal))	First(<insert-inpetal>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#"}
134	First(mint (<all-type-value>))	First(mint)	{“mint”}
135	First(<common-type> #identifier =)	First(<common-type>)	{"tint", "flora", "chard", "string", "bloom"}
136	First(<sqnc-type> #identifier =)	First(<sqnc-type>)	{"florist", "tulip", "dirt", "stem"}
137	First(#identifier <insert-func> <indexing> <start-end-step> <more-id> <assignment-op>)	First(#)	{“#”}
138	First(string literal)	First(string literal)	{string literal}
139	First(ε) U Follow(<inpetal-state>)	Follow(<inpetal-state>)	{“)”}
140	First(, #identifier <insert-func> <indexing> <start-end-step> <more-id>)	First(,)	{“,”}
141	First(ε) U Follow(<more-id>)	Follow(<more-id>)	{"=", "+=", "-=", "*=", "/=", "%=", "**=", "//="}
142	First(eleaf (<condition>) (<statement>); <eleaf>)	First(eleaf)	{“eleaf”}

143	First(ε) U Follow(<eleaf>)	Follow(<eleaf>)	{"moss", "hard", "tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#", "mint", "leaf", "fern", "willow", "tree", "clear", "break", $, ")", "regrow"}
144	First(moss (<statement>);)	First(moss)	{“moss”}
145	First(ε) U Follow(<else>)	Follow(<else>)	{"hard", "tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#", "mint", "leaf", "fern", "willow", "tree", "clear", "break", $, ")", "regrow"}
146	First(<insert-inpetal> <all-type-value>)	First(<insert-inpetal>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#"}
147	First(<assign> <insert-assign>)	First(<assign>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", "#", $}
148	First(<insert-inpetal>)	First(<insert-inpetal>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem", “#”}
149	First(ε) U Follow(<assign>)	Follow(<assign>)	{"tint", "flora", "chard", "string", "bloom", "florist", "tulip", "dirt", "stem"}
150	First(<common-type> (<all-type-value>))	First(<common-type>)	{"tint", "flora", "chard", "string", "bloom"}
151	First(<sqnc-type> (<all-type-value>))	First(<sqnc-type>)	{"florist", "tulip", "dirt", "stem"}
152	First(=)	First(=)	{“=”}
153	First(+=)	First(+=)	{“+=”}
154	First(-=)	First(-=)	{“-=”}
155	First(*=)	First(*=)	{“*=”}
156	First(/=)	First(/=)	{“/=”}
157	First(%=)	First(%=)	{“%=”}
158	First(**=)	First(**=)	{“**=”}
159	First(//=)	First(//=)	{"//="}

160	First(fern (<insert-fern>)	First(fern)	{“fern”}
161	First(willow (<condition>) (<statement>))	First(willow)	{“willow”}
162	First(tint #identifier = tint literal; <condition>; #identifier <assignment-op> <tint>;) (<statement>))	First(tint)	{“tint”}
163	First(<all-type-value> <more-value> at <sequence>;) (<statement>))	First(<all-type-value>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem", "inpetal"}
164	FIrst(, <all-type-value> <more-value>)	First(,)	{“,”}
165	First(ε) U Follow(<more-value>)	Follow(<more-value>)	{“at”}
166	First(<all-type-value> <insert-branch> <more-branch>)	First(<all-type-value>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem", "inpetal"}
167	First(_: <statement>)	First(_:)	{“_:”}
168	First(: <statement>)	First(:)	{“:”}
169	First(leaf (<condition>) (<statement>);)	First(leaf)	{“leaf”}
170	First(<statement>)	First (<statement>)	{“hard”, “tint”, “flora”, “chard”, “string”, “bloom”, “florist”, “tulip”, “dirt”, “stem”, “#”, “mint”, “leaf”, “fern”, “willow”, “tree”, “clear”, “break”, ε}
171	First(branch <check-branch>)	First (branch)	{“branch”}
172	First(branch <check-branch>)	First(branch)	{“branch”}
173	First(ε) U Follow(<more-branch>)	Follow(<more-branch>)	{“)”}

174	First(<insert-argument>)	First(<insert-argument>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem", "inpetal", $}
175	First(<common-type> #identifier <common-data> <more**kwargs>)	First(<common-type>)	{"tint", "flora", "chard", "string", "bloom"}
176	First(ε) U Follow(<argument>)	Follow(<argument>)	{“)”}
177	First(<all-type-value> <add-argument>)	First(<all-type-value>)	{tint literal, flora literal, chard literal, string literal, bloom literal, "#", "lent", "tint", "flora", "chard", "string", "bloom", "getItems", "getKeys", "getValues", "bare", "[", "{", "florist", "tulip", "dirt", "stem", "inpetal"}
178	First(#identifier (<argument>) <add-argument>)	First(#)	{“#”}
179	First(ε) U Follow(<insert-argument>)	Follow(<insert-argument>)	{“)”}
180	First(, <argument>)	First(,)	{“,”}
181	First(ε) U Follow(<add-argument>)	Follow(<add-argument>)	{“)”}
182	First(, <common-type> #identifier <common-data> <more**kwargs>)	First(,)	{“,”}
183	First(ε) U Follow(<more**kwargs>)	Follow(<more**kwargs>)	{“)”}
184	First(<common-type> #identifier (<parameter>) (<statement> regrow <all-type-value> <add-at>;); <function>)	Fist(<common-type>)	{“tint", "flora", "chard", "string", "bloom"}
185	First(viola #identifier (<undefined-param>) (<statement>); <function>)	First(viola)	{“viola”}
186	First(ε) U Follow(<function>)	Follow(<function>)	{“plant”}
187	First(<more-value> at <all-type-value>)	First(<more-value>)	{“[“, $}

188	First(ε) U Follow(<add-at>)	Follow(<add-at>)	{“;”}
189	First(<undefined-param>)	First(<undefined-param>)	{“tint", "flora", "chard", "string", "bloom", “**”, $}
190	First(<common-type> #identifier <common-data> <next-parameter>)	First(<common-type>)	{“tint", "flora", "chard", "string", "bloom"}
191	First(<sqnc-type> #identifier <sqnc-value> <next-parameter>)	First(<sqnc-type>)	{"florist", "tulip", "dirt", "stem"}
192	First(#identifier (<parameter>) <next-parameter>)	First(#)	{“#”}
193	First(ε) U Follow(<parameter>)	Follow(<parameter>)	{“)”}
194	First(<common-type> *#identifier <add-kwargs>)	First(<common-type>)	{“tint", "flora", "chard", "string", "bloom"}
195	First(**#identifier)	First(**)	{“**”}
196	First(ε) U Follow(<undefined-param>)	Follow(<undefined-param>)	{“)”}
197	First(, **#identifier)	First(,)	{“,”}
198	First(ε) U Follow(<add-kwargs>)	Follow(<add-kwargs>)	{“)”}
199	First(, <parameter>)	First(,)	{“,”}
200	First(ε) U Follow(<next-parameter>)	Follow(<next-parameter>)	{“)”}

C - GRASS PLUS COMPILER SYSTEM TESTING

XI. Test Script

a. Lexical Test Script

a. Reserved Words

NUMBER	TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT	REMARKS	TESTER
DATA TYPES					
1	tint(#b);	No Lexical Error	No Lexical Error	**PASSED**	GUMAWIDRAA
2	bloom;	Lexical Error **bloom** - Invalid Delim	Lexical Error **bloom** - invalid Delim	**PASSED**	GUMAWIDRAA
3	flora[#a];	Lexical Error **flora** - Invalid Delim	Lexical error **flora** - Invalid Delim	**PASSED**	GUMAWIDRAA
4	char+	Lexical Error **char** - Expecting hashtag (#) symbol	Lexical Error **char** - Expecting hashtag (#) symbol	**PASSED**	GUMAWIDRAA
5	florist a;	Lexical Error **a** - Expecting hashtag (#) symbol	Lexical Error **a** - Expecting hashtag (#) symbol	**PASSED**	GUMAWIDRAA
6	stem=	Lexical Error **stem** - Invalid Delim	Lexical Error **stem** - Invalid Delim	**PASSED**	GUMAWIDRAA
7	stem[	Lexical Error **stem** - Invalid Delim	Lexical Error **stem** - Invalid Delim	**PASSED**	GUMAWIDRAA
8	tulip{	Lexical Error **tulip** - Invalid Delim	Lexical Error **tulip** - Invalid Delim	**PASSED**	GUMAWIDRAA
9	string ({#	Lexical Error { - Invalid Delim # - Expecting numulet	Lexical Error { - Invalid Delim # - Expecting numulet	**PASSED**	GUMAWIDRAA
10	string *a;	Lexical Error **a** - Expecting hashtag (#) symbol * - Invalid Delim	Lexical Error **a** - Expecting hashtag (#) symbol * - Invalid Delim	**PASSED**	GUMAWIDRAA
11	florist(a);	Lexical Error **a** - Expecting hashtag (#)	Lexical Error **a** - Expecting hashtag (#)	**PASSED**	GUMAWIDRAA

		symbol	symbol		
12	tint{b};	Lexical Error **tint** - Invalid Delim **b** - Expecting Hashtag (#) Symbol	Lexical Error **tint** - Invalid Delim **b** - Expecting Hashtag (#) Symbol	**PASSED**	GUMAWIDRAA
13	chard #c;	No Lexical Error	No Lexical Error	**PASSED**	GUMAWIDRAA
14	string(#e);	No Lexical Error	No Lexical Error	**PASSED**	GUMAWIDRAA
15	dirt string	No Lexical Error	No Lexical Error	**PASSED**	GUMAWIDRAA
INPUT AND OUTPUT STATEMENTS					
16	inpetal #a;	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
17	inpetal[]	Lexical Error **inpetal** - Invalid Delim **[** - Invalid Token	Lexical Error **inpetal** - Invalid Delim **[** - Invalid Token	**PASSED**	LIMLDP
18	mint)	Lexical Error **mint,)**- Invalid Delim	Lexical Error **mint,)**- Invalid Delim	**PASSED**	LIMLDP
19	mint();	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
20	inpetal("Enter Num: ");	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
21	inpetal(10);	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
22	inpetal([ ]);	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
23	mint(['a', 'b']);	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
CONDITIONAL STATEMENTS					
24	leaf ( );	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
25	moss()	Lexical Error **moss, (**- Invalid Delim	Lexical Error **moss, (**- Invalid Delim	**PASSED**	LIMLDP
26	eleaf{};	Lexical Error **eleaf, {**- Invalid Delim	Lexical Error **eleaf, {**- Invalid Delim	**PASSED**	LIMLDP
27	leaf[]	Lexical Error **leaf, [**- Invalid Delim	Lexical Error **leaf, [**- Invalid Delim	**PASSED**	LIMLDP
28	at()	Lexical Error **at, (**- Invalid Delim	Lexical Error **at, (**- Invalid Delim	**PASSED**	LIMLDP

29	nut:	Lexical Error **nut, :**- Invalid Delim	Lexical Error **nut, :**- Invalid Delim	**PASSED**	LIMLDP
30	nut@	Lexical Error **nut, @**- Invalid Delim	Lexical Error **nut, @**- Invalid Delim	**PASSED**	LIMLDP
31	true.	Lexical Error **true, .**- Invalid Delim	Lexical Error **true, .**- Invalid Delim	**PASSED**	GUMAWIDRAA
32	false=	Lexical Error = - Invalid Delim	Lexical Error = - Invalid Delim	**PASSED**	GUMAWIDRAA
33	break\	Lexical Error **break,** \- Invalid Delim	Lexical Error **break,** \- Invalid Delim	**PASSED**	GUMAWIDRAA
34	/break	Lexical Error /, **break**- Invalid Delim	Lexical Error /, **break**- Invalid Delim	**PASSED**	GUMAWIDRAA
35	branch|	Lexical Error **branch,** |- Invalid Delim	Lexical Error **branch,** |- Invalid Delim	**PASSED**	GUMAWIDRAA
36	tree_(	Lexical Error **tree,** _- Invalid Delim	Lexical Error **tree,** _- Invalid Delim	**PASSED**	GUMAWIDRAA
OTHERS					
37	viola(	Lexical Error **viola, (**- Invalid Delim	Lexical Error **viola, (**- Invalid Delim	**PASSED**	GUMAWIDRAA
38	viola;	Lexical Error **viola, ;**- Invalid Delim	Lexical Error **viola, ;**- Invalid Delim	**PASSED**	GUMAWIDRAA
39	=bare	Lexical Error **=, bare**- Invalid Delim	Lexical Error **=, bare**- Invalid Delim	**PASSED**	GUMAWIDRAA
40	[bare]	Lexical Error **]** - Invalid Delim	Lexical Error **]** - Invalid Delim	**PASSED**	GUMAWIDRAA
41	getkeys();	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
42	hard-	Lexical Error **hard,-** - Invalid Delim	Lexical Error **hard,-** - Invalid Delim	**PASSED**	LIMLDP
43	getItems);	Lexical Error **getItems,)**- Invalid Delim	Lexical Error **getItems,)**- Invalid Delim	**PASSED**	LIMLDP

44	big	Lexical Error **big** - Expecting hashtag (#) symbol	Lexical Error **big** - Expecting hashtag (#) symbol	**PASSED**	LIMLDP
45	tree();	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
46	break;	No Lexical Error	No Lexical Error	**PASSED**	LIMLDP
47	regrow;	Lexical Error **regrow** - Invalid Delim	Lexical Error **regrow** - Invalid Delim	**PASSED**	LIMLDP
48	regrow c;	Lexical Error **c** - Expecting hashtag (#) symbol	Lexical Error **c** - Expecting hashtag (#) symbol	**PASSED**	LIMLDP
49	hard #c	Lexical Error **#c** - Invalid Delim	Lexical Error **#c** - Invalid Delim	**PASSED**	LIMLDP
50	#a.lent	Lexical Error **#a, lent** - Invalid Delim	Lexical Error **#a, lent** - Invalid Delim	**PASSED**	LIMLDP

b. Syntax Test Script

ITEM	TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT	REMARKS	TESTER
BASIC STRUCTURE					
1	seed	SYNTAX ERROR -Expecting plant and garden	SYNTAX ERROR -Expecting plant and garden	PASSED	GUMAWIDRAA
2	(seed) (plant)	SYNTAX ERROR -Expecting seed, plant and garden	SYNTAX ERROR -Expecting seed, plant and garden	PASSED	GUMAWIDRAA
3	seed()(); plant()();	SYNTAX ERROR -Expecting plant and garden	SYNTAX ERROR -Expecting plant and garden	PASSED	GUMAWIDRAA
4	seed garden()(); plant	NO SYNTAX ERROR	NO SYNTAX ERROR		GUMAWIDRAA
5	garden(seed)(seed) garden(plant)(plant)	SYNTAX ERROR - Expecting seed and plant	SYNTAX ERROR - Expecting seed and plant	PASSED	GUMAWIDRAA
VARIABLE DECLARATION					

6	seed
	floral tint #testNo_39;
	garden() (flora #testNo_39;);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

7	seed
	hard floral florist #testNo_17;
	garden() (stem #testNo_1313;);
	plant
		SYNTAX ERROR
		- Expecting floral for global declaration
			SYNTAX ERROR
			- Expecting floral for global declaration
				PASSED	LIMLDP

8	seed floral florist #testNo_18; hard garden() (stem #testNo_1414;); plant	SYNTAX ERROR - Expecting garden	SYNTAX ERROR - Expecting garden	PASSED	LIMLDP
9	seed	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

	garden()(hard tint #r = 34;); plant				
10	seed garden()(tint #a, #b, #c;); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP
TYPECASTING					

11	seed
	garden() (
	string #testNo_36;
	tint(#testNo_36);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

12	seed garden() (string #testNo_37; tint#testNo_37;); plant	SYNTAX ERROR	SYNTAX ERROR	PASSED	LIMLDP
13	seed garden() (string #testNo_38; #testNo_38 tint;); plant	SYNTAX ERROR	SYNTAX ERROR	PASSED	LIMLDP

14	seed
	
	floral tint #r = 10;
	
	garden()(
	string #s = string(#r);
	);
	
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

15	seed
	
	floral tint #r = 10;
	
	garden()(
	flora #h = flora(#r);
	);
	
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

INPUT AND OUTPUT STATEMENTS					

16	seed garden() (tulip #testNo_125 = {125, 125.125, "b", "stringB", "false"}; mint#testNo_125;); plant	SYNTAX ERROR - Expecting (for mint	SYNTAX ERROR - Expecting (for mint	PASSED	GUMAWIDRAA
17	seed garden(mint (#testNo_125)) (mint (tulip #testNo_125 = {125, 125.125, "b", "stringB", "false"};) mint (#testNo_125);); plant	SYNTAX ERROR	SYNTAX ERROR	PASSED	GUMAWIDRAA

18	seed
	floral hard string #globalmint_id = "mint of a string";
	garden() (mint (#globalmint_id));
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

19	seed
	floral hard string #globalmint_id = "mint of a string";
	garden() (#functionmint_id("string on a function"));
	string #functionmint_id(string #funcParam_id) (
	mint(#globalmint_id);
	regrow #funcParam_id;
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

20	seed
	floral hard string #globalmint_id = mint("mint of a string");
	garden() mint((mint (#globalmint_id)));
	plant
		SYNTAX ERROR
		mint unknown literal, expecting string literal
			SYNTAX ERROR
			mint unknown literal, expecting string literal
				PASSED	GUMAWIDRAA

ARITHMETIC OPERATIONS					

21	seed
	garden() (
	tint #testNo_125 = 71 + 71;
	mint(#testNo_125);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

22	seed	NO SYNTAX	NO SYNTAX	PASSED	LIMLDP

	garden() (
	flora #testNo_125 = 71 + 71 - (71) * (2 / 4);
	mint(#testNo_125);
	);
	plant
		ERROR	ERROR		

23	seed
	garden() (
	tint #testNo_7 = (7+7);
	flora #testNo_3 = 3737 - (9+12) * (40-6);
	
	flora #testNo_73 = (#testNo_3) / #testNo_7;
	mint(#testNo_73);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

24	seed
	garden() (
	tint #testNo_125 = 71 +-*/% 71;
	mint(#testNo_125);
	);
	plant
		SYNTAX ERROR
		line: tint #testNo_125 = 71 +-*/% Undefined operator
			SYNTAX ERROR
			line: tint #testNo_125 = 71 +-*/% Undefined operator
				PASSED	LIMLDP

25	seed
	garden() (
	flora #testNo_125 = ((((71 + 71 - (71) * (2 / 4));
	mint(#testNo_125);
	);
	plant
		SYNTAX ERROR
		line: flora #testNo_125 = ((
		Expecting flora literal
			SYNTAX ERROR
			line: flora #testNo_125 = ((
			Expecting flora literal
				PASSED	LIMLDP

COMPARISON OPERATIONS					
26	seed garden() (chard #testNo_1 = “a”; string #testNo_8 = “abracadabra”; leaf (#testNo_1 at #testNo_8) (mint(“poof”);); moss (mint(“...”););); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

27	seed
	floral tint #globalTint_id = 11;
	garden()(
	mint (#functionTint_id(1) nut #globalTint_id);
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

	); tint #functionTint_id (tint #funcParam_id)(tint #localTint_id = inpetal("Suppose the user inputs 10"); regrow #funcParam_id + #localTint_id;); plant				

28	seed
	floral flora #globalFlora_id = 3.14;
	garden()(
	flora #localFlora_id = 3.14;
	
	mint(#globalFlora_id == localFlora_id);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

29	seed
	garden()(
	tint #localTint_id = (534 + 58) - (5 * 5), #anotherTint_id = 234;
	
	leaf(#localTint_id <= #anotherTint_id) (
	mint("anotherTint steps higher");
	);
	eleaf(#localTint_id >= 10000) (
	mint("the localTint has a great potential");
	);
	moss(
	mint("anotherTint steps in to greater heights");
	);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

30	seed garden()(mint(234 > 12 < 00023);); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA
LOGICAL OPERATIONS					
31	seed garden()(florist #localFlorist_id = [324, 342.234, ‘e’, "englishLanguage", "true"];	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

	fern(#element at #localFlorist_id) (mint('e' at #element););); plant				
32	seed garden()(mint("b" not at "pinball");); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP
33	seed garden()(mint(("h" not "e") not ("l" at "encyclopedia");); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP
34	seed garden()(floral tint #floralTint_id not break; floral bloom #floralBloom_id at clear;); plant	SYNTAX ERROR line: garden()(floral floral undefined	SYNTAX ERROR line: garden()(floral floral undefined	PASSED	LIMLDP
35	seed garden()(not string #stringID not at chard #chardID not not;); plant	SYNTAX ERROR line: garden()(not not undefined	SYNTAX ERROR line: garden()(not not undefined	PASSED	LIMLDP
CONDITIONAL STATEMENTS					

36	seed
	garden()(
	bloom #isTrue = true;
	
	leaf(#isTrue != true)(
	mint("this is good");
	);
	eleaf(599 > 499)(
	mint("go for 499");
	);
	moss(
	mint("goes home");
	);
	);
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

37	seed garden()(tint #treeID = inpetal("Enter	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

	value");
	
	tree (#treeID) (
	branch 1 :
	mint("Entering case 1");
	break;
	branch 2 :
	mint("Entering case 2");
	break;
	branch 3 :
	mint("Entering case 3");
	break;
	branch 4 :
	mint("Entering case 4");
	break;
	);
	);
	plant				

38	seed tree (#seed_grows_to_a_tree)(branch : break;); garden(tree)(mint("look, this is just a bark");); plant tree (#plant_grows_to_a_tree)(branch : break;);	SYNTAX ERROR line: seed tree Expecting garden or floral	SYNTAX ERROR line: seed tree Expecting garden or floral	PASSED	LIMLDP
39	tree (seed #hashtagSeed) (branch : break;); tree (garden #hashtagGarden) (branch : break;); tree (plant #hashtagPlant) (branch : break;);	SYNTAX ERROR Expecting seed at the start of the code	SYNTAX ERROR Expecting seed at the start of the code	PASSED	LIMLDP
40	tree bloom leaf(seed not at plant) (#treeVial_sample) (branch : break;); tree at garden bloom moss at florist #identification = ["there is a tree", "there is a branch", "there is a leaf", "what am i missing"]; tree bloom eleaf(plant bloom at garden) (#treantMystery) (branch : break; branch : leaf(#treeVial_sample >= #treantMystery)(mint("its last leaf falls");););	SYNTAX ERROR Expecting seed at the start of the code	SYNTAX ERROR Expecting seed at the start of the code	PASSED	LIMLDP
ITERATIVE STATEMENTS					

41	seed
	
	garden()(
	fern(tint #i = 1; #i < 10; #i+=1;) (
	mint(#i);
	);
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA

	); plant				
42	seed floral tint #c = 10; garden()(willow(#c<100)(#c *= 2; ;); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA
43	seed floral tint #c = 10; garden()(willow(#c>0)(#c -= 2; ;); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA
44	seed garden()(tint #c = 1000; willow(#c<25)(#c //= 2; ;); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA
45	seed garden()(tint #c = 1000; willow(#c<25)(#c //= 2; ;); plant	NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	GUMAWIDRAA
FUNCTIONS					
46	seed tint #functiontint_id (tint #tintNo1_id, flora #floraNo1_id) (regrow #floraNo1_id // #tintNo1_id); plant	SYNTAX ERROR	SYNTAX ERROR	PASSED	GUMAWIDRAA

47	seed
	bloom #functionbloom_id (chard #chardNo1_id, string #stringNo1_id) (regrow #chardNo1_id at #stringNo1_id;);
	garden()();
	plant
		SYNTAX ERROR	SYNTAX ERROR	PASSED	GUMAWIDRAA

48	seed	NO SYNTAX	NO SYNTAX	PASSED	LIMLDP

	garden()(
	tint #a = #double(5);
	mint(#a);
	);
	
	tint #double(tint #x)(regrow #x * 2;);
	
	plant
		ERROR	ERROR		

49	seed
	garden() (
	
	tint #multiply_by_5 = #multiply_by(5);
	
	tint #result = #multiply_by_5(10);
	
	);
	
	tint #multiply_by(tint #n)(
	tint #multiplier(tint #x)(
	regrow #x * #n;
	);
	regrow multiplier;
	);
	
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

50	seed
	
	floral tint #kk = 9999;
	
	garden()(
	#minty(#kk);
	);
	
	viola #minty(tint #a) (
	mint(#a);
	);
	
	plant
		NO SYNTAX ERROR	NO SYNTAX ERROR	PASSED	LIMLDP

c. Semantic Test Script

Item	Test Case	Expected Output	Actual Output	Remarks	Tester
TYPE CHECKING					
1	seed garden()(tint #a = 5;); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
2	seed garden()(string #a = 5;); plant	SEMANTIC ERROR	SEMANTIC ERROR: TYPE MISMATCH	PASSED	CALADIAOJZ
3	seed garden()(florist #a = [“list”: {‘a’, ‘b’, ‘c’}];); plant	SEMANTIC ERROR	SEMANTIC ERROR: TYPE MISMATCH	PASSED	CALADIAOJZ
4	seed garden()(tulip #a = {5, 0.34, ‘1’, “people”};); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
5	seed garden()(dirt #a = {“list”: {‘a’, ‘b’, ‘c’}};); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
6	seed garden() (tint #a = 10 + “5”;); plant	SEMANTIC ERROR	ERROR MESSAGE: SEMANTIC ERROR: TYPE MISMATCH	PASSED	TENIOJR

<table>
<tr><td>7</td><td>seed

garden() (
  string #a = “5”;
  tint #b = 10 + tint(#a);
);

plant</td><td>NO SEMANTIC ERROR</td><td>NO SEMANTIC ERROR</td><td>PASSED</td><td>TENIOJR</td></tr>
<tr><td>8</td><td>seed

garden() (
  flora #c = 47.85;
  string #d = “adding numeric to alphabeic”;

  mint(#d + “is equals to” + string(#c));
);

plant</td><td>NO SEMANTIC ERROR</td><td>NO SEMANTIC ERROR</td><td>PASSED</td><td>TENIOJR</td></tr>
<tr><td>9</td><td>seed

garden() (
  flora #e = 34863.525438;
  flora #f = 96.256;

  tint #g = #e / #f;
);

plant</td><td>NO SEMANTIC ERROR</td><td>NO SEMANTIC ERROR</td><td>PASSED</td><td>TENIOJR</td></tr>
<tr><td>10</td><td>seed

garden() (
  chard #t = ‘t’;
  chard #r = ‘r’;
  chard #u = ‘u’;
  chard #e = ‘e’;

  bloom #true = bloom(#t + #r + #u + #e);
);

plant</td><td>NO SEMANTIC ERROR</td><td>NO SEMANTIC ERROR</td><td>PASSED</td><td>TENIOJR</td></tr>
<tr><td>11</td><td>seed

garden() (
  hard tint #candy;

  #candy = 18;
  #candy = 24;
  #candy = 12;
  #candy = 34;
);

plant</td><td>SEMANTIC ERROR</td><td>SEMANTIC ERROR: IMMUTABLE VARIABLE'S VALUE IS BEING EDITED</td><td>PASSED</td><td>TENIOJR</td></tr>
</table>

12	seed
	
	garden() (
	  hard florist #metalShed;
	
	  tulip(#metalShed);
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: IMMUTABLE VARIABLE'S VALUE IS BEING EDITED	PASSED	TENIOJR

13	seed floral hard string #bundle; garden() (#bundle += "workshop";); plant	SEMANTIC ERROR	SEMANTIC ERROR: IMMUTABLE VARIABLE'S VALUE IS BEING EDITED	PASSED	TENIOJR
VARIABLE USAGE					

14	seed
	
	garden() (
	  tint #a = 10;
	  flora #b = #zen;
	  #zen = (#a * 2) / 3;
	  flora #zen;
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: UNINITIALIZED VARIABLE USAGE	PASSED	TENIOJR

15	seed
	
	garden() (
	  tint #a = 10;
	  flora #zen;
	  #zen = (#a * 2) / 3;
	  flora #b = #zen;
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

16	seed
	
	garden() (
	  string #repeater = "tell this";
	  mint(#repeater);
	
	  #repeater += " again";
	  mint(#repeater);
	
	  #repeater += " and again";
	  mint(#repeater);
	
	  #repeater += " one more time";
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

	mint(#repeater);
	
	#repeater += " keep it going";
	mint(#repeater);
	);
	
	plant				

17	seed garden() (florist #variableUsage_101 = [#a, #b, #c, #d, #e, #f, #g, #h, #i, #j, #k];); plant	SEMANTIC ERROR	SEMANTIC ERROR: UNDECLARED VARIABLE USAGE	PASSED	TENIOJR
18	seed floral florist #vowel = ['a', 'e', 'i', 'o', 'u']; garden() (#chard1 = #vowel[0]; #chard2 = #vowel[1]; #chard3 = #vowel[2]; #chard4 = #vowel[3]; #chard5 = #vowel[4];); plant	SEMANTIC ERROR	SEMANTIC ERROR: UNDECLARED VARIABLE USAGE	PASSED	TENIOJR

19	seed
	
	garden() (
	florist #array = [10, 20.34, 'c', "string", true];
	
	flora #get_value = #array[1];
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

20	seed
	
	garden() (
	florist #array = [10, 20.34, 'c', "string", true];
	
	bloom #get_value = #array[-1];
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

21	seed	NO SEMANTIC	NO SEMANTIC	PASSED	TENIOJR

	garden() (
	  tulip #matrix = {1,2,3,
	[1,2,3, {“book”:[1,2,3],
	“string”, “line”:[1,2,3}]};
	
	  florist #getMatrix =
	getItems(#matrix[“book”]);
	);
	
	plant
		ERROR	ERROR		

22	seed
	
	floral stem #fixgroup =
	{1,2,3,4,5,6,7,8,9,10};
	
	garden() (
	  fern (#i at #fixgroup) (
	    mint(#megaIndex());
	  );
	);
	
	  viola #megaIndex() (
	    florist #indexer =
	[#fixgroup[#fixgroup[[#fixgr
	oup[0]]],
	#fixgroup[#fixgroup[[#fixgro
	up[1]]],
	#fixgroup[#fixgroup[[#fixgro
	up[2]]],
	#fixgroup[#fixgroup[[#fixgro
	up[3]]],
	#fixgroup[#fixgroup[[#fixgro
	up[4]]],
	#fixgroup[#fixgroup[[#fixgro
	up[5]]],
	#fixgroup[#fixgroup[[#fixgro
	up[6]]],
	#fixgroup[#fixgroup[[#fixgro
	up[7]]],
	#fixgroup[#fixgroup[[#fixgro
	up[8]]],
	#fixgroup[#fixgroup[[#fixgro
	up[9]]],];
	
	  regrow #indexer;
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

23	seed floral tulip #masterIndex = {1,2,3,4,5,6,7,8,9,10}; garden() (florist #indexes = {1,2,3};	SEMANTIC ERROR	SEMANTIC ERROR: UNINITIALIZED VARIABLE USAGE	PASSED	TENIOJR

	fern(#i at #masterIndex) (
	mint(#indexes[#i]);
	);
	);
	
	plant				

24	seed garden() (mint (67 % 41 + 3 * 12 - 5 / 2 ** 5 // 23);); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

25	seed
	
	garden() (
	flora #a = 15.3;
	flora #b = 73845.1231;
	tint #c = 90000;
	tint #arithmetic_isEqualsTo = 5 + #b * #c // #a = 43;
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

SCOPE CHECKING					

26	seed
	floral tint #x = 10;
	
	garden() (
	#func123();
	mint(#x);
	);
	
	viola #func123() (
	#x = 20;
	mint(#x);
	);
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

27	seed
	
	garden() (
	chard #chara = ‘c’;
	string #word = “these are words”;
	
	mint(#chard);
	);
	
	viola #voidFunction() (
	mint(#word);
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: VARIABLE OUT OF SCOPE	PASSED	TENIOJR

28	seed
	
	floral flora #rose = 250.45
	
	garden() (
	  mint(#rose);
	  #rose = 299.99
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

29	seed
	
	garden() (
	  mint(#func);
	  mint(#integralFunction);
	);
	
	tint #integralFunction(tint #func) (
	  return #func ** 5
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: VARIABLE OUT OF SCOPE	PASSED	TENIOJR

30	seed
	
	garden() (
	  string #crunch = “koko”;
	  mint(#crunch);
	  leaf(1) (
	    #crunch += “krantsh”;
	    mint(#crunch);
	    leaf(2) (
	      #crunch += “por onli payb tawsan milyon dalars”;
	      mint(#crunch);
	    );
	  );
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

CONDITIONAL & ITERATIVE USAGE					

31	seed
	
	garden() (
	  fern(tint #i=1; #i<10; #i+=1) (
	    mint(#i);
	  );
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ

32	seed garden() (	SEMANTIC ERROR	SEMANTIC ERROR: EXCEEDED THE	PASSED	CALADIAOJZ

	fern(tint #i=1; #i<10;
	#i+=1) (
	mint(#i);
	
	fern(tint #j=1; #j<10;
	#j+=1) (
	mint(#j);
	
	fern(tint #k=1; #k<10;
	#k+=1) (
	mint(#k);
	
	fern(tint #l=1; #l<10;
	#l+=1) (
	mint(#l);
	
	);
	);
	);
	);
	);
	
	plant
			MAXIMUM NUMBER OF NESTING		

33	seed
	
	garden() (
	leaf(1) (
	mint(1);
	);
	eleaf(2) (
	mint(2);
	);
	eleaf(3) (
	mint(3);
	);
	eleaf(4) (
	mint(4);
	);
	eleaf(5) (
	mint(5);
	);
	moss (
	mint(6);
	);
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ

34	seed
	
	garden() (
	leaf(1) (
	mint(1);
	
	leaf(2) (
	mint(2);
	
	leaf(3) (
		SEMANTIC ERROR	SEMANTIC ERROR: EXCEEDED THE MAXIMUM NUMBER OF NESTING	PASSED	CALADIAOJZ

	mint(3);
	
	leaf(4) (
	mint(4);
	);
	);
	);
	);
	);
	
	plant				

35	seed
	
	garden() (
	tint #apple = 3;
	tint #orange = 2;
	tint #mango = 1;
	tint #banana = 3;
	
	tree(#apple) (
	branch 1:
	tree (#orange) (
	branch 1:
	tree (#mango) (
	branch 1: break;
	branch 2: break;
	branch 3:
	tree (#banana) (
	branch 1: break;
	branch 2: break;
	branch 3: break;
	);
	);
	branch 2: break;
	branch 3: break;
	);
	branch 2:
	tree (#orange) (
	branch 1:
	tree (#mango) (
	branch 1: break;
	branch 2: break;
	branch 3:
	tree (#banana) (
	branch 1: break;
	branch 2: break;
	branch 3: break;
	);
	);
	branch 2: break;
	branch 3:
	tree (#mango) (
	branch 1: break;
	branch 2: break;
	branch 3:
	tree (#banana) (
	branch 1: break;
	branch 2: break;
		SEMANTIC ERROR	SEMANTIC ERROR: EXCEEDED THE MAXIMUM NUMBER OF NESTING	PASSED	CALADIAOJZ

	branch 3: break;););); branch 3: break;);); plant				
FUNCTION USAGE					
36	seed garden() (#greet (Herbicode);); string #greet (string #name) (regrow “Hello, ” + #name;); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
37	seed garden() (#trimodule(1,2,3,4,5,6,7,8,9, 10);); flora #trimodule(tint #first, tint #second, tint #third) (regrow (#first + #first) * (#first + #second + #third) * (#second + #third ** 3);); plant	SEMANTIC ERROR	SEMANTIC ERROR: PARAMETERS AND ARGUMENTS MISMATCH	PASSED	CALADIAOJZ

38	seed
	
	garden() (
	#infiniteRecurse();
	);
	
	viola #infiniteRecurse() (
	mint(“redacted”);
	#infiniteRecurse();
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: PARAMETERS AND ARGUMENTS MISMATCH	PASSED	CALADIAOJZ

39	seed garden() (#a(); #b(); #c();	SEMANTIC ERROR	SEMANTIC ERROR: PARAMETERS AND ARGUMENTS MISMATCH	PASSED	CALADIAOJZ

	#d(); #e();); viola #a_b_c_d_e() (mint("ABCDEFGHIJKLMNOPQ RSTUVWXYZ");); plant				
40	seed garden() (string #read_func = #isString(1,2,3,4,5,6,7,8,9,10);); tint #isString(tint *#b) (tint #x = 1; fern(#i at #b) (#x *= (#i ** 2);); regrow #x;); plant	SEMANTIC ERROR	SEMANTIC ERROR: PARAMETERS AND ARGUMENTS MISMATCH	PASSED	CALADIAOJZ
41	seed garden() (#factorial (5);); tint #factorial(tint #n) (leaf (#n == 0) (regrow 1;); moss (regrow #n * #factorial(#n-1););); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
42	seed garden() (#recurse(1,2,3,4,5,6,7,8,9,10) ;); viola #recurse(tint *#args) (	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ

	fern(#i at #args) (
	  leaf(#i+1 == bare) (
	    break;
	  );
	  moss (
	    #recurse(#i) * 2;
	  );
	);
	);
	
	plant				

43	seed
	
	garden() (
	  #limiter(“this message is too long that we needed to shorten down to the number of acceptable characters“, 24);
	);
	
	bloom #limiter(string #message, tint #chara) (
	  string #temp;
	
	  leaf(#message.length(#message) <= #chara) (
	    regrow #message;
	  );
	  moss(
	    #temp = #message[0:-2];
	    #limiter();
	  );
	);
	
	tint #length(string #getMessage) (
	  tint #count = 0;
	  fern(#i at #getMessage) (
	    #count += 1;
	  );
	  regrow #count;
	);
	
	plant
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ

44	seed
	
	garden() (
	  fern(tint #i=1; #i<10; #i+=1) (
	    #fibonacci(#i);
	  );
	);
	
	tint #fibonacci(tint #a) (
	  leaf (#a <= 1) (
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ

	regrow n); moss (regrow #fibonacci(n-1) + #fibonacci(n-2));) plant				
45	seed garden() (tulip #numeric = {1,2,3,4,5,6,7,8,9,10}; #addTulips(#numeric);); tint #addTulips(tulip #a) (leaf (#a == bare) (regrow 0;); moss (regrow #a[0] + #addTulips(#a[1:]););); plant	NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	CALADIAOJZ
DIVISION BY ZERO					
46	seed garden() (mint (12/0);); plant	SEMANTIC ERROR	SEMANTIC ERROR: DIVISION BY ZERO FOUND	PASSED	TENIOJR
47	seed garden() (chard #zero = ‘0’; mint(tint(#zero) / tint(#zero));); plant	SEMANTIC ERROR	SEMANTIC ERROR: DIVISION BY ZERO FOUND	PASSED	TENIOJR

48	seed
	
	garden() (
	flora #piNut = 0.14;
	
	mint(0 / #piNut);
	);
		NO SEMANTIC ERROR	NO SEMANTIC ERROR	PASSED	TENIOJR

	plant				

49	seed
	
	garden() (
	  florist #eggs = [0, 0.0, ‘0’, “00000”];
	
	  mint(#eggs[0] / #eggs[1] + flora(#eggs[3]) + tint(#eggs[2]));
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: DIVISION BY ZERO FOUND	PASSED	TENIOJR

50	seed
	
	garden() (
	  mint(#getAnEgg(‘0’, “000”, [“00”, ‘0’, “000”, “00”]) / (49-49));
	);
	
	string #getAnEgg(chard #egg, string #eggtray, florist #eggbasket) (
	  regrow tint(#egg) + tint(#eggtray) + tint(#eggbasket[0]);
	);
	
	plant
		SEMANTIC ERROR	SEMANTIC ERROR: DIVISION BY ZERO FOUND	PASSED	TENIOJR

d. 50 Sample Programs' Source Code

INPUT/OUTPUT MACHINE PROBLEMS		
ITEM	**DESCRIPTION**	**SOURCE CODE**

1	Demonstration of output with different literals.	seed
		garden()(
		tint #num = 1;
		flora #flt = 2.5;
		string #str = "Hello";
		florist #lst = [#num, #flt, #str];
		
		mint(#num);
		mint(#flt);
		mint(#str);
		mint(#lst);
		);
		plant

2	Demonstration of output with basic string interpolation.	seed
		garden()(
		tint #num = 10;
		mint("The number is equal to {#num}.");
		);
		plant

3	Demonstration of output with string interpolation with arithmetic operation.	seed
		garden()(
		tint #num1=5, #num2=2, #num3 = #num1 * #num2;
		mint("{#num1} multiplied by {#num2} equals {#num3}");
		);
		plant

4	Demonstration of output with string interpolation with conditional operators.	seed
		garden()(
		tint #num1 = 5, #num2 = 10;
		bloom #bool = #num1 > #num2;
		mint("{#num1} is greater than {#num2}, which is {#bool}.");
		);
		plant

5	Demonstration of output with string interpolation with logical operators.	seed
		garden()(
		tint #num1=10, #num2=5;
		string #str = "Hello", #ele = "e";
		bloom #bol1 = #num1 < #num2, #bol2 = #ele at #str;
		bloom #logic = #bol1 =/ #bol2;
		mint("{#bol1} or {#bol2}, which is {#logic}.")
		);
		plant

6	Displaying output with string concatenation and interpolation.	seed garden()(string #str1 = "Hello ", #str2 = "World!", #strc =

		#str1+#str2; mint("The concatenation of {#str1} and {#str2} is {#strc}.");); plant

7	A program for taking tint input and displaying the typecasted tint value.	seed
		garden()(
		tint #num = inpetal("Enter Number: ");
		flora #flt = flora(#num);
		string #str = string(#num);
		
		mint(#flt);
		mint(#str);
		);
		plant

8	A program for taking a string user input that is a math operation with a single operator (no spaces and decimal, tint only) and display's the answer.	seed
		garden()(
		string #equation = inpetal("Enter Simple Math Operation: ");
		tint #eq1=0, #eq2=0, #ans=0, #ind=0;
		florist #lst = ["+", "-", "*", "/"];
		string #op = " ";
		
		fern(tint #i=0; #i<lent(#equation); #i+=1;)(
		leaf(#equation[#i] at #lst)(
		#op = #equation[#i];
		#ind = #i;
		break;
		);
		);
		
		tree(#op)(
		branch "+":
		#eq1 = tint(#equation[:#ind]);
		#eq2 = tint(#equation[#ind+1:]);
		#ans = #eq1 + #eq2;
		mint(#ans);
		break;
		
		branch "-":
		#eq1 = tint(#equation[:#ind]);
		#eq2 = tint(#equation[#ind+1:]);
		#ans = #eq1 - #eq2;
		mint(#ans);
		break;
		
		branch "*":
		#eq1 = tint(#equation[:#ind]);
		#eq2 = tint(#equation[#ind+1:]);
		#ans = #eq1 * #eq2;
		mint(#ans);
		break;
		
		branch "/":
		#eq1 = tint(#equation[:#ind]);
		#eq2 = tint(#equation[#ind+1:]);

		#ans = #eq1 / #eq2;
		mint(#ans);
		break;
		);
		);
		plant

9	A program for taking a temperature input in celsius and outputting a list of tuples of its conversions to kelvin and fahrenheit.	seed
		garden()(
		flora #cel = inpetal("Enter Temperature in Celsius: ");
		flora #kel = #cel + 273.15;
		flora #fah = #cel * 9/5 + 32;
		
		florist #lst = [{"Fahrenheit", #fah}, {"Kelvin", #kel}];
		mint(#lst[0]);
		mint(#lst[1]);
		);
		plant

10	Program for taking multiple tint user input and displaying the sum.	seed garden()(tint #sum = 0; willow(true)(string #input = inpetal("Enter an integer (or 'q' to quit): "); leaf((#input == "q") =/ (#input == "Q"))(break;); moss(#sum += tint(#input););); mint("The sum of all entered integers is: {#sum}");); plant
CONDITIONAL MACHINE PROBLEMS		
ITEM	**DESCRIPTION**	**SOURCE CODE**

1	Swap notation with a single leaf statement.	seed
		garden()(
		tint #a = 5, #b = 6;
		leaf(#a > #b)(
		#a, #b = #b, #a;
		mint("Values has been swapped.");
		);
		);
		plant

2	Swap notation with three variables with a leaf-moss statement.	seed garden()(tint #a = 5, #b = 6, #c = 7; leaf(#c > #b)(

		#a, #b, #c = #b, #c, #a;
		mint("Values has been swapped.");
		);
		moss(
		mint("Values was not swapped");
		);
		);
		plant

3	Swap notation with four variables through a nested leaf-moss statement.	seed
		garden()(
		tint #a = 5, #b = 6, #c = 7, #d = 9;
		leaf(#c > #b)(
		leaf(#c > #a)(
		#a, #b, #c, #d = #d, #b, #c, #a;
		mint("Values has been swapped.");
		);
		);
		moss(
		mint("Values was not swapped");
		);
		);
		plant

4	Swap notation with four variables through leaf-moss ladder.	seed
		garden()(
		tint #a = 5, #b = 6, #c = 7, #d = 9;
		leaf(#c > #b)(
		#a, #b, #c, #d = #d, #b, #c, #a;
		mint("Values has been swapped.");
		);
		eleaf(#c > #a)(
		#a, #b, #c, #d = #b, #a, #d, #c;
		mint("Values has been swapped.");
		);
		moss(
		mint("Values was not swapped");
		);
		);
		plant

5	Program for checking what day it is based on the tint input.	seed
		garden()(
		tint #input = inpetal("Enter Number: ");
		tree(#input)(
		branch 1:
		mint("Monday");
		break;
		branch 2:
		mint("Tuesday");
		break;
		branch 3:

		mint(“Wednesday”); break; branch 4: mint(“Thursday”); break; branch 5: mint(“Friday”); break; branch 6: mint(“Saturday”); break; branch 7: mint(“Sunday”); break;);); plant
6	Program for demonstrating tree-branch combined branches.	seed garden()(char #a = ‘b’; tree(#a)(branch ‘a’ =/ ‘e’ =/ ‘i’ =/ ‘o’ =/ ‘u’: mint(“Vowel”); break; branch _: mint(“Consonant”); break;);); plant

7	Program for demonstrating tree-branch list branches.	seed
		garden()(
		florist #details = [“Morning”, “Josh”];
		
		tree(#details)(
		branch [#time, #name]:
		mint(“{#time} {#name}.”);
		break;
		);
		);
		plant

8	Program for demonstrating tree-branch list branches with *args.	seed garden()(florist #details = [“Vowels”, "a", "e", "i", "o", "u"]; tree(#details)(branch [#word, *#letters]: mint(“{#word}: {#letters}”);

		break;);); plant

9	Tree-branch program, where branches have iteration.	seed
		garden()(
		florist #details = [“Vowels”, ["a", "e", "i", "o", "u"]];
		
		tree(#details)(
		branch [#word, #letters]:
		fern(tint #i=0; #i<lent(#letters); #i+=1;)(
		mint(“{#word}: {#letters[#i]}”);
		);
		break;
		);
		);
		plant

10	Tree-branch program with one one output for multiple branches.	seed garden()(chard #a = ‘b’; tree(#a)(branch ‘a’: branch ‘e’: branch ‘i’: branch ‘o’: branch ‘u’: mint(“Vowel”); break; branch _: mint(“Consonant”); break;);); plant
ITERATIVE MACHINE PROBLEMS		
ITEM	**DESCRIPTION**	**SOURCE CODE**

1	Demonstration of list concatenation through continuous positive integer user input using a willow loop.	seed
		garden()(
		florist #temp = [], #lst = [];
		mint("Enter elements for the list (enter a negative integer to stop): ”);

		willow(true)(tint #num = inpetal(); leaf(#num < 0)(break;); #temp = [#num]; #lst = #lst + #temp;); mint(#lst)); plant

2	Fern list unpacking demonstration.	seed
		garden()(
		florist #lst = [1, "hello", 'c', 3.14];
		
		fern(#ele at #lst)(
		mint(#ele);
		);
		);
		plant

3	Dirt unpacking of elements through a fern loop.	seed
		garden()(
		dirt #dict = {"program1":"Python", "program2":"Java"};
		
		fern(#key, #value at #dict)(
		mint("{#key} : {#value}");
		);
		);
		plant

4	Demonstration of continuous string concatenation with a willow loop.	seed
		garden()(
		string #str = "", #input;
		mint("Enter words for the list (enter the word stop to stop): ");
		
		willow(true)(
		#input = inpetal();
		leaf(#input == "stop" =/ #input == "STOP")(
		break;
		);
		#str = #str + " " + #input;
		);
		mint(#str)
		);
		plant

5	Fern sequence unpacking with string interpolation of output.	seed garden()(florist #lst = [1, "hello", 'c', 3.14];

		tint #cnt = 1;
		
		fern(#ele at #lst)(
		mint(“Element {#cnt}: {#ele}”);
		#cnt += 1;
		);
		);
		plant

6	Fern unpacking of the dictionary and displaying items through string interpolation.	seed
		garden()(
		dirt #dict = {“Screening API”:”FastAPI”, “ARISE Website”:”Spring”};
		
		fern(#key, #value at #dict)(
		mint(“{#key} is coded with the use of the {#value} framework.”);
		);
		);
		plant

7	Unpacking a dictionary using a willow loop.	seed
		garden()(
		dirt #my_dict = {“a”: 1, “b”: 2, “c”: 3};
		
		florist #keys = getKeys(#my_dict);
		tint #i = 0;
		
		willow(#i < lent(#keys))(
		string #key = #keys[#i];
		tint #value = #my_dict[#key];
		mint(“Key is {#key} while value is {#value}.”);
		#i += 1;
		);
		);
		plant

8	FizzBuzz problem solution through willow loop.	seed
		garden()(
		tint #cnt = 25;
		
		willow(#cnt <= 25)(
		leaf(#cnt % 3 == 0 =& #cnt % 5 == 0)(
		mint("FizzBuzz");
		);eleaf(#cnt % 3 == 0)(
		mint("Fizz");
		); eleaf(#cnt % 5 == 0)(
		mint("Buzz");
		); moss(
		mint(#cnt);
		);
		#cnt += 1;

		);); plant

9	Program for removing all odd numbers in a list.	seed
		garden()(
		florist #lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11], #newlst = [];
		
		fern(tint #i=0; #i<lent(#lst); #i += 1;)(
		leaf(#lst[#i] % 2 == 0)(
		#newlst = #newlst + #lst[#i];
		);
		);
		mint(#newlst);
		);
		plant

10	Program for removing all prime numbers from a list.	seed
		garden()(
		florist #nums = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11], #np = [];
		tint #i = 0, #divisor = 2;
		bloom #isPrime = true;
		
		willow(#i < lent(#nums))(
		leaf(#nums[#i] <= 1)(
		#np = #np + #nums[#i];
		);
		moss(
		willow(#divisor < #nums[#i])(
		leaf(#nums[#i] % #divisor ==
		0)(
		#isPrime = false;
		break;
		);
		#divisor += 1;
		);
		leaf(#isPrime)(
		#np = #np + #nums[#i];
		);
		);
		#i += 1;
		);
		mint(#np);
		);
		plant

DATA STRUCTURES

ITEM	DESCRIPTION	SOURCE CODE
1	Displaying a florist of dirts.	seed garden()(

		florist #lst = [{"Name": "John Doe"}, {"Age": 13}];
		mint(#lst);
		);
		plant

2	Displaying a florist of tulips.	seed
		garden()(
		tulip #a = {"hello", "world"}, #b= {1, 2, 2, 4, 5};
		florist #lst = [#a, #b];
		mint(#lst);
		);
		plant

3	Displaying a florist of stems.	seed
		garden()(
		stem #a = {'a', 'e', 'i', 'o', 'u'}, #b= {1, 2, 4, 5};
		florist #lst = [#a, #b];
		mint(#lst);
		);
		plant

4	Displaying a stem of florists.	seed
		garden()(
		florist #a = ["hello", "world"], #b= [1, 2, 2, 4, 5];
		stem #lst = [#a, #b];
		mint(#lst);
		);
		plant

5	Displaying a stem of tulips.	seed
		garden()(
		tulip #a = {"hello", "world"}, #b= {1, 2, 2, 4, 5};
		florist #lst = [#a, #b];
		mint(#lst);
		);
		plant

6	Displaying a stem of dirts	seed
		garden()(
		stem #lst = {{"Name": "John Doe"}, {"Age": 13}};
		mint(#lst);
		);
		plant

7	Displaying a tulip of florists	seed
		garden()(
		florist #a = ["hello", "world"], #b= [1, 2, 2, 4, 5];
		tulip #lst = {#a, #b};
		mint(#lst);
		);
		plant

8	Displaying a tulip of dirts	seed
		garden()(
		tulip #lst = {{“Name”: “John Doe”}, {“Age”: 13}};
		mint(#lst);
		);
		plant

9	Displaying a tulip of stems	seed
		garden()(
		stem #a = {“hello”, “world”}, #b= {1, 2, 4, 5};
		tulip #lst = {#a, #b};
		mint(#lst);
		);
		plant

10	Displaying a multi-dimensional florist	seed
		garden()(
		florist #lst = [[‘a’, ‘b’, ‘c’, ‘d’], [2, 6, 4, 8, 0, 1]];
		mint(#lst);
		);
		plant

FUNCTION MACHINE PROBLEMS		
ITEM	**DESCRIPTION**	**SOURCE CODE**

1	Memoization for getting the Fibonacci sequence's nth term.	seed
		garden()(
		tint #nth = #fib(5);
		mint(#nth);
		);
		
		tint #fib(tint #n, dirt #memo = {})(
		tint #result = 0;
		leaf(string(#n) at #memo)(
		regrow #memo[string(#n)];
		); eleaf(#n <= 1)(
		#result = #n;
		); moss(
		#result = #fib(#n - 1, #memo) + #fib(#n - 2, #memo);
		);
		#memo[string(#n)] =#result;
		regrow #result;
		);
		plant

2	Memoization for solving the Coin Change problem.	seed
		garden()(
		tint #amount = 15;
		florist #coins = [1, 5, 10];
		mint(#coin_change(#amount, #coins));

<table>
<tr><td></td><td></td><td><pre>
        );

        tint #coin_change(tint #amount, florist #coins, dirt #memo={})(
                leaf(#amount < 0)(
                        regrow -1;
                );
                leaf(#amount == 0)(
                        regrow 0;
                );
                leaf(string(#amount) at #memo)(
                         regrow #memo[string(#amount)];
                );

                tint #min_coins = -1;
                fern(#coin at #coins)(
                        tint #remaining_amount = amount - coin;
                        tint #count = 1 +
                #coin_change(#remaining_amount, #coins, #memo);
                        leaf(min_coins<count)(
                                #min_coins = #count
                        );
                );
                #memo[string(#amount)] = #min_coins;
                regrow #min_coins;;
        );
plant
</pre></td></tr>
<tr><td>3</td><td>Memoization for solving exponentiation.</td><td><pre>
seed
        garden()(
                mint(#exp(2, 5));
                mint(#exp(3, 4));
        );

        tint #exp(tint #base, tint #exponent, dirt #memo={})(
                leaf(#exponent == 0)(
                        regrow 1;
                );
                leaf(exponent == 1)(
                        regrow #base;
                );

                string #key = “({#base}, {#exponent})”;
                leaf(#key at memo)(
                    regrow #memo[#key];
                );

                tint #result = #base * #expl(#base, #exponent - 1,
        #memo);
                #memo[#key] = #result;
                regrow #result;
</pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td><pre>
        );
plant
</pre></td></tr>
<tr><td>4</td><td>Function for counting the total number of characters in a string.</td><td><pre>
seed
        garden()(
                mint(#len(“Hello World!”));
        );

        tint #len(string #txt)(
                regrow lent(#txt);
        );
plant
</pre></td></tr>
<tr><td>5</td><td>Function for converting an input tint into a flora.</td><td><pre>
seed
        garden()(
                tint #n = inpetal(“Enter num:”);
                mint(#to_float(#n));
        );

        flora #to_float(tint #num)(
                regrow flora(#num);
        );
plant
</pre></td></tr>
<tr><td>6</td><td>Function for converting a positive input tint into a negative flora.</td><td><pre>
seed
        garden()(
                tint #c = inpetal(“Enter Negative Number: ”);
                mint(#to_negafloat(#c));
        );

        tint #to_negafloat(tint #num)(
                string #str = string(#num);
                tint #temp = tint(#str[1:]);
                regrow flora(#temp);
        );
plant
</pre></td></tr>
<tr><td>7</td><td>Function for converting an input flora into a tint.</td><td><pre>
seed
        garden()(
                flora #n = inpetal(“Enter decimal number:”);
                mint(#to_int(#n));
        );

        flora #to_intt(flora #dec)(
                regrow tint(#dec);
        );
plant
</pre></td></tr>
<tr><td>8</td><td>Function for converting a negative input flora into a positive tint.</td><td><pre>
seed
        garden()(
                flora #c = inpetal(“Enter Decimal Number: ”);
                mint(#to_positint(#c));
</pre></td></tr>
</table>

		);
		
		    tint #to_positint(flora #num)(
		        string #str = string(#num);
		        flora #temp = flora(#str[1:]);
		        regrow #temp;
		    );
		plant

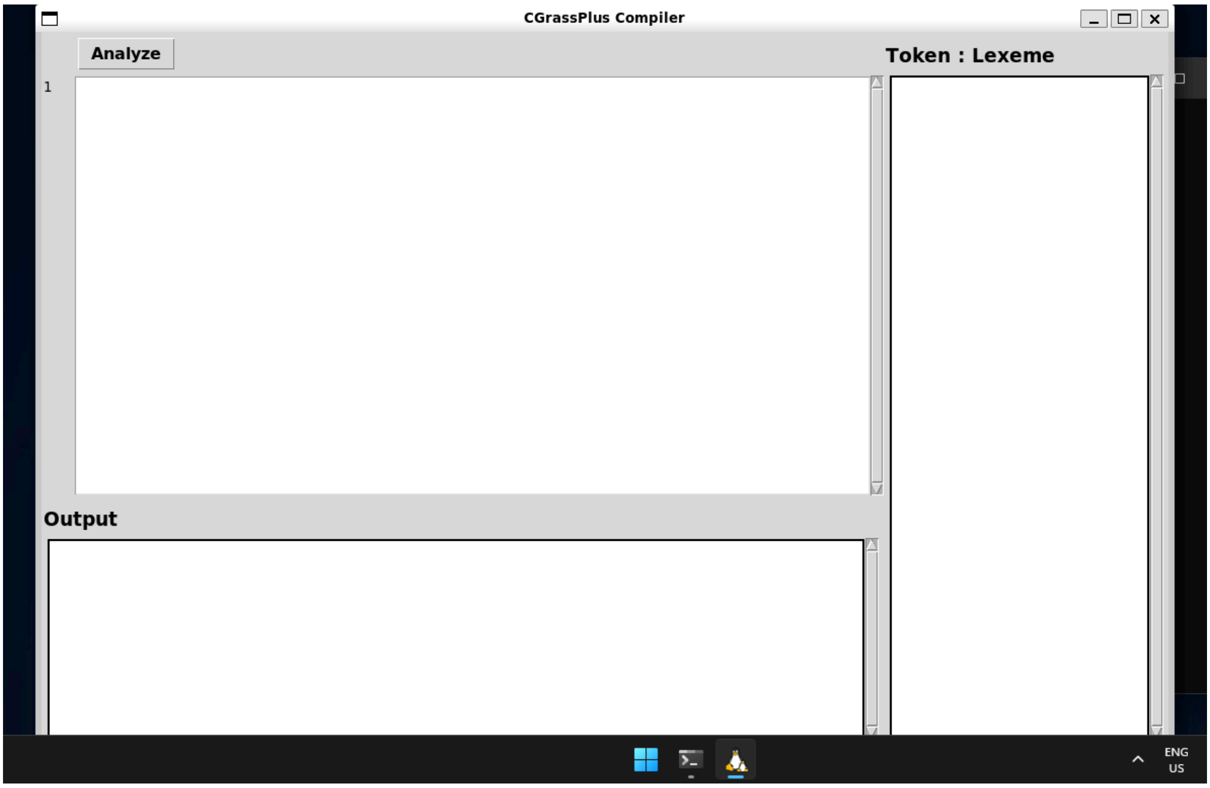
9	Function for removing the vowels in a string.	seed
		    garden()(
		        mint(#remove_vowels(“Hello World!”));
		    );
		
		    bloom #is_vowel(string #chr)(
		        string #vowels = "aeiouAEIOU";
		        fern(#vowel at #vowels)(
		            leaf(#char == #vowel)(
		                regrow true;
		            );
		        regrow false;
		    );
		
		    string #remove_vowels(string #input_string)(
		        string #result = "";
		        fern(#chr at #input_string)(
		            leaf(nut #is_vowel(#chr))(
		                #result += #chr;
		            );
		        );
		        regrow #result;
		    );
		plant

10	Program that showcases function overloading through the square figure.	seed
		    garden()(
		        mint(#square(2,2));
		    );
		
		    tint #square(tint #n)(
		        regrow 4 * #n;
		    );
		
		    tint #square(tint #n1, tint #n2)(
		        regrow #n1 * #n2;
		    );
		plant

DOCUMENTATION

XIII. C - Grass PLUS UI

Main



Output (Input)

Output

XIV. Group Members' Pictures

CANDO, Jhaime Jose O.
Leader

CALADIAO, Jerome Z.
Member

LIM, Lance Daniel P.
Member

GUMAWID, Reuel Augustus A.
Member

TENIO, Jonald R.
Member

Group Picture