Documentation for “**IYKYK**” Programming Language

Group 2 Members:

Alamag, Jose Luis
Calendario, Mark Kenneth
Caspe, Mark Vincent
Favorito, Vince Lennard
Lalis, Reygine
Villegas, Daniel

Principles of Programming Languages
COSC 30063
BSCS 3-1N

I. Introduction

Ayo! Check this out — A new programming language just dropped for intellectuals of this generation. “If You Know, You Know” (IYKYK) language is the new glow up for the current existing programming language. The language is named after the GenZ term “IYKYK” which is something you would say when a topic is exclusive to a certain group. This language uses famous GenZ terms as part of their syntax. This is to show the creativity of the current youth in creating a new programming language that is engaging and easy to understand among their current peers. Since the current demographic of our future programmers are GenZ, having a language that they can easily engage and relate to will be helpful in learning programming.

Don’t get twisted; even though it’s hella beginner friendly IYKYK isn’t playing around. Its development aims to solve or improve some parts in programming languages, no cap! These sheeshable improvements are the following:

1. Improved Callback Functions
2. Function Contracts
3. Undefined Value Safety
4. Stepwise Incrementation
5. Improved and Simplified Looping
6. HTML / CSS Support

“IYKYK” uses a functional paradigm of programming. This means that programming using this language relies on functions. This programming language is heavily influenced by OG languages such as C and C++ when it comes to most of its syntax and paradigm, as well as other influences from Python, C#, and PHP, it took most of their good attributes and improved some concerns in programming. It is developed for the purpose of software development just like most of the languages that influenced it. A major slay for the new for programming.

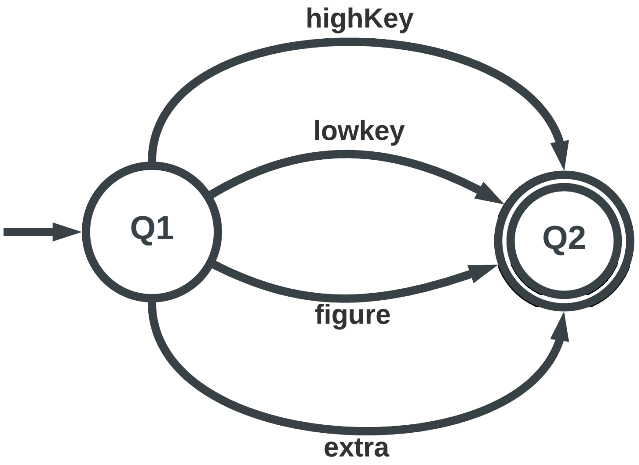
Alright, so "IYKYK" is the ultimate GenZ code, tailor-made for GenZ squad. It's not just about coding; it's a whole vibe. Learning with it is a breeze—it's easy, chill, and zero intimidation. The identifiers? Lit. They make coding fun and engaging, turning it into a dope experience. "IYKYK" isn't just a technical language; it's all about welcoming vibes for every programmer out there. It's the code that speaks our language and brings those good vibes into the coding game. 🚀✌️ #IYKYKLanguage #GenZCode

II. Syntactic Elements of Language

1. Character Set

mainCharacter = {characters, numbers, extra}
characters = {highKey, Lowkey}
numbers = {-figure* | figure*}
highKey = {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}
lowKey = {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}
figure = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
extra = {!, {, }, (,), [,], ., ,, <, >, =, +, *, -, /, _, %, \, ^}

State Machine:



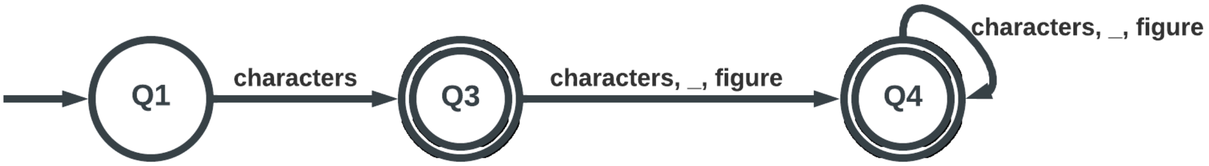
2. Identifiers

Rules

Identifiers can accept lowKey and highKey only. **identifier** = {characters, figure}
Here are the identifier rules for our proposed programming language:

1. Identifiers must not contain *extra* symbols from the character set. It must contain *characters* and *figure* only.
2. Identifiers must start with *characters* from the character set. Starting with *figure* characters is not allowed.
3. This proposed programming language encusers to use camelCase as its naming convention for readability and consistency. A valid camelCase requires the first word to start with a lowercase letter, and subsequent words are capitalized.
4. Identifiers are case sensitive.

State Machine:



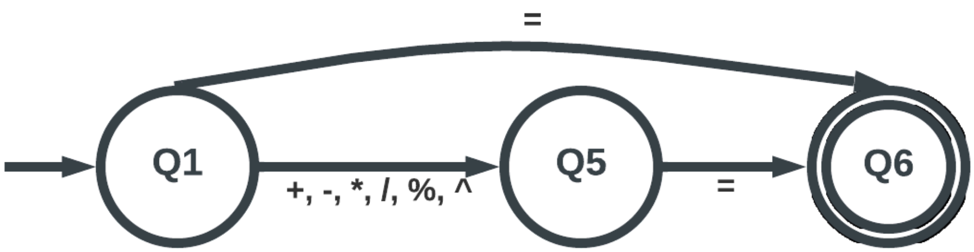
3. Operation Symbols

Assignment Operation

assignmentOperation = {=, +=, -=, /=, *=, %=, ^=}

Extra	Example	Expanded Form	Description
=	X = Y	X = Y	Assigns the value of Y to X
+=	X += Y	X = X + Y	Assigns the combined values of X and Y to X
-=	X -= Y	X = X - Y	Assigns the result of subtracting Y from the current value of X to X.
*=	X *= Y	X = X * Y	Assigns the multiplied value of X and Y to X.
/=	X /= Y	X = X / Y	Assigns the result of X divided by Y to X.
%=	X %= Y	X = X % Y	Assigns the remainder of X divided by Y to X.
^=	X ^= Y	X = X ^ Y	X powered by Y is assigned to X.

State Machine:

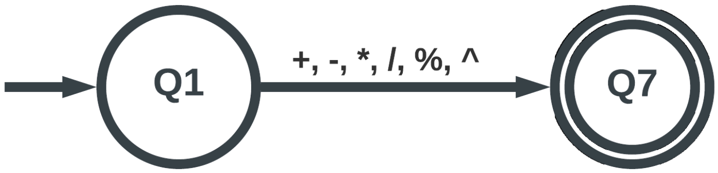


Arithmetic Operation

arithmeticOperation = {+, -, *, /, %, ^}

Extra	Description	Sample	Description
+	Addition	X + Y	Combines the value of X and Y together.
-	Subtraction	X - Y	Subtracts the value of Y from X
*	Multiplication	X * Y	Multiplies the value of X by Y, can also be expressed as multiplies the value of Y by X
/	Division	X / Y	Divides the value of X by Y
%	Modulo	X % Y	Returns the remainder after X is divided by Y
^	Exponent	X ^ Y	Calculates the value of X to the power of Y

State Machine:

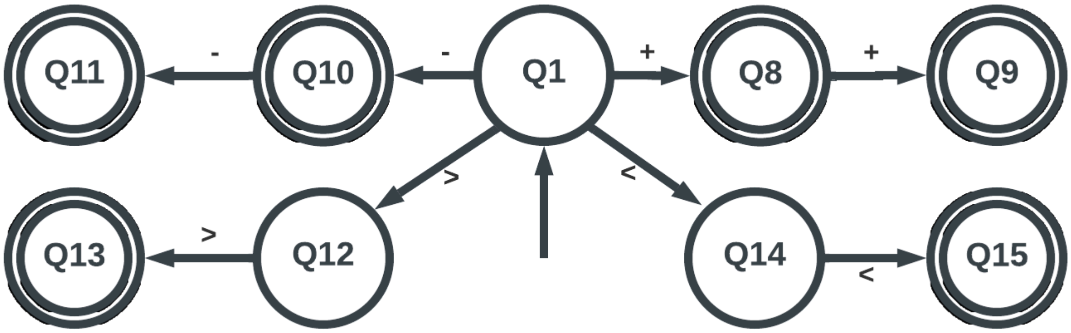


Unary Operation

unaryOperation = {+, -, ++, –, >>, <<}

Extra	Example	Description
+	+X	Turns the value of X into a number
-	-X	Turns the value of X into a number and makes it negative
++	++X (Prefix) or X++ (Postfix)	Adds 1 to the value of X
--	--X (Prefix) or X-- (Postfix)	Subtracts 1 to the value of X
>>	Stepwise Increment	Increment a variable value by N
<<	Stepwise Decrement	Decrement a variable value by N

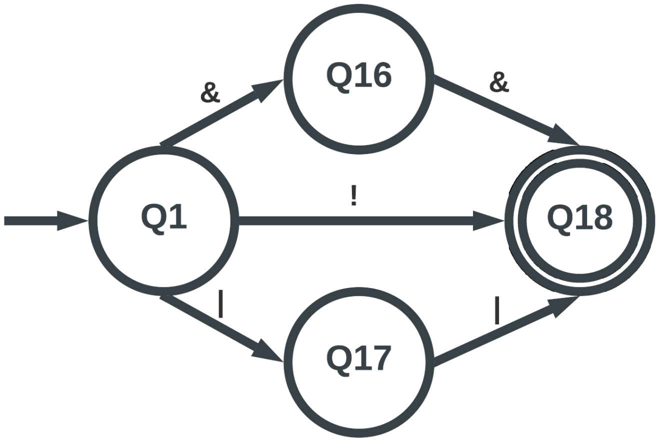
State Machine:



Logical Operation

logicalOperation = {&&, ||, !}

Extra	Example	In Word	Description
!	!X	Not	Inverses the boolean of X
&&	X && Y	And	Returns noCap only when both X and Y are true Returns cap when one of X and Y is false
||	X || Y	Or	Returns noCap when one of X and Y are true

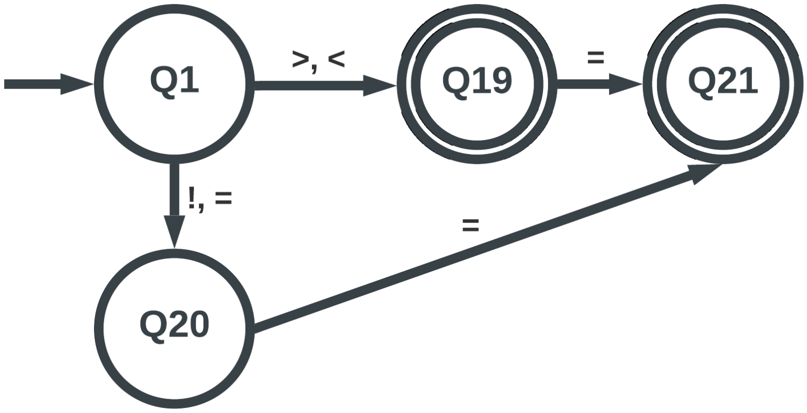


Relational Operation

relationalOperation = {<, >, ==, <=, >=, !=}

Extra	Operation	Example	Description
<	Less than	X < Y	Expression is true if the value of X is less than Y, otherwise return false.
>	Greater than	X > Y	Expression is true if the value of X is greater than Y, otherwise return false.
==	Equal to	X == Y	Expression is true if both operands are equal, otherwise return false.
!=	Unequal to	X != Y	Expression is true if both operands are not equal, otherwise return false.
<=	Less than or equal to	X <= Y	Expression is true if the value of X is less than or equal to Y, otherwise return false.
>=	Greater than or equal to	X >= Y	Expression is true if the value of X is greater than or equal to Y, otherwise return false.

State Machine



4. Keywords and Reserved Words

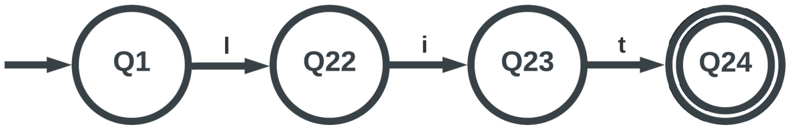
Keyword	Meaning	Description
Variable / Constant		
lit	variable	Used for variable declaration.
fire	constant	Used for constant declaration.
Conditionals		
yeet	if	Checks if the specified condition is met (true) and then

		executes the block of code it contains.
yas	else	Serves as an optional statement when the preceding conditional statement/s are not met.
yikes	else if	Defines a new condition if the preceding condition is false. The block of code it contains is executed when the condition is satisfied.
Looping		
relapse	loop	Repeat a section of code for a specific number of times
as	variable	State determiner of the iteration. Optional.
worse	initialization	It initializes the iteration.
recover	loop termination	It stops the loop.
periodt	break	Forces the loop to stop.
swerve	continue	Skips the current iteration of the loop and continues to the next one.
Exception Handling		
sus	try	Tests a block of code for errors while it is being executed.
dead	catch	Handles try block error occurrences by defining a block of code to be executed.
gotcha	finally	Executes a block of code even there is an exception
slay	raise	Raise or throw an error.
Simplified Callback Function		
delay	callback function instantiation	Enables a callback function avoiding the callback hell.
chill	pause	This keyword is permitted on callback functions. Allowing a function to run in a multi-threaded like state outside the main call stack.
Boolean		
real	true	Boolean value indicating that the condition is true.
cap	false	Boolean value indicating that the condition is false.

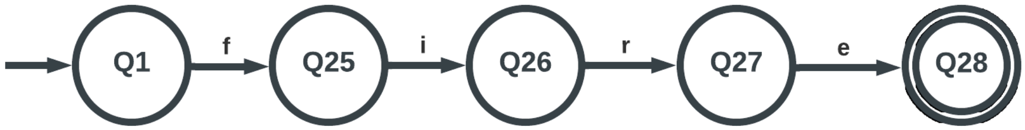
Types		
fig	integer	Refers to a positive or negative integer value.
yarn	string	Refers to the sequence of characters that makes up a text or word.
char	character	Refers to the single character value.
fuzzy	float	Refers to a number value with a decimal point.
ghosted	null	It indicates that a data element or variable has no defined value.
tea	boolean	Refers to a true or false value.
Other Reserved Words		
flex	print	It displays text, numbers, or value of a variable into the output device or console.
spill	scan	It reads and processes the data entered by the user.
Keywords that can be used with routines		
routine	function	Block of code that performs a task and can be reused multiple times.
bounce	return	Ends the current function and returns a value to the calling function.
bet	require	A contractor or condition needed to satisfy before making a function call.

State Machines:

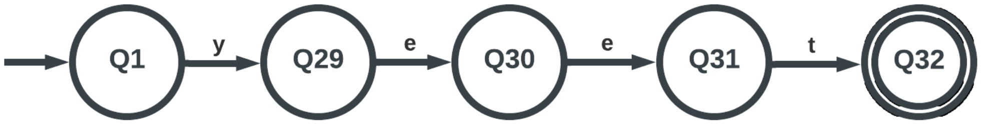
lit



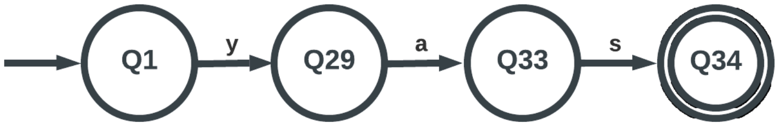
fire



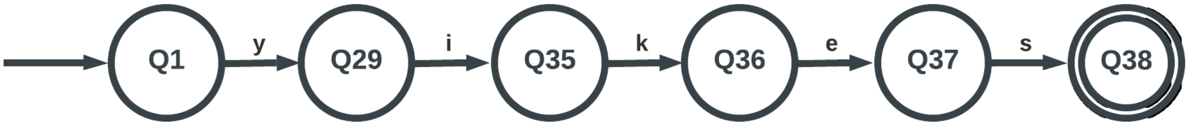
yeet



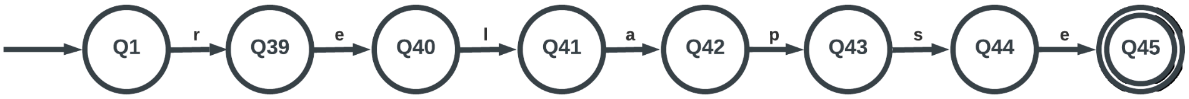
yas



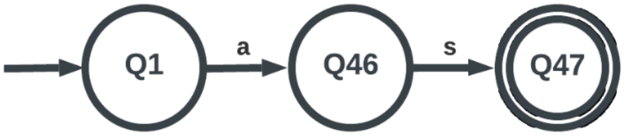
yikes



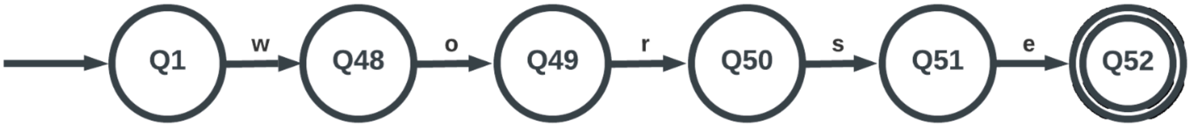
relapse



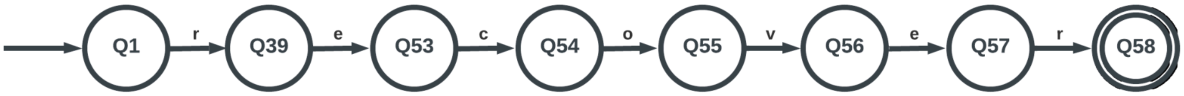
as



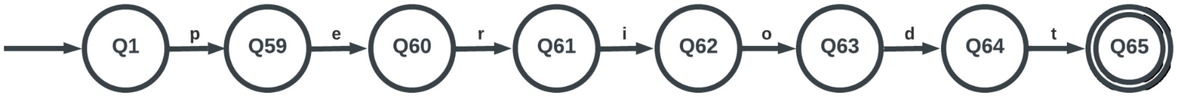
worse



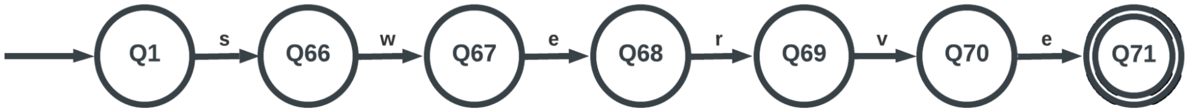
recover



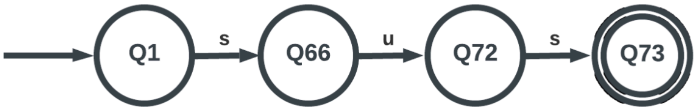
periodt



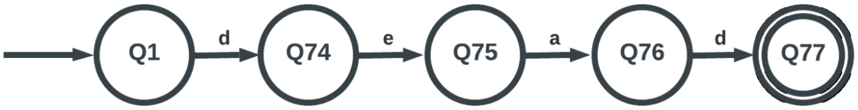
swerve



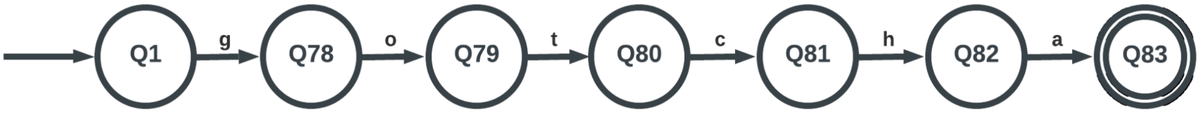
sup



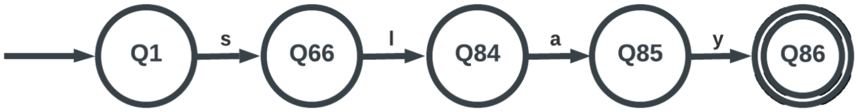
dead



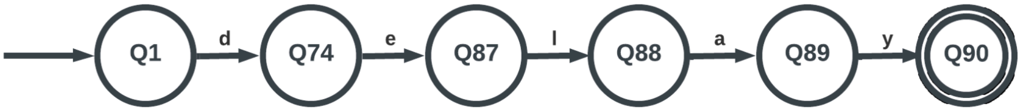
gotcha



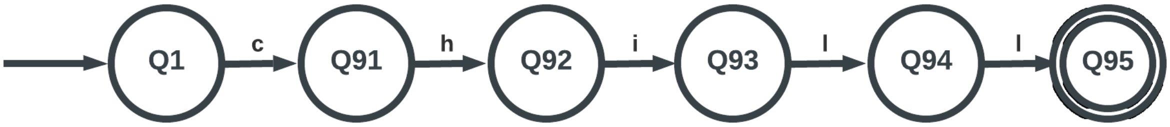
slay



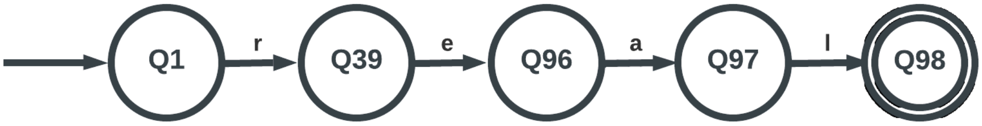
delay



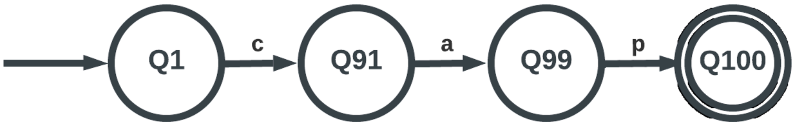
chill



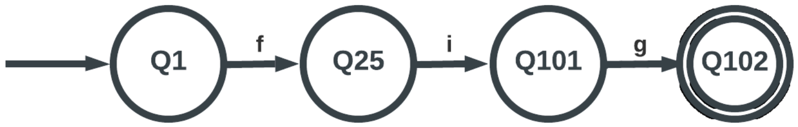
real



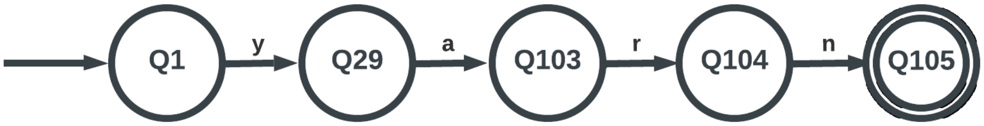
cap



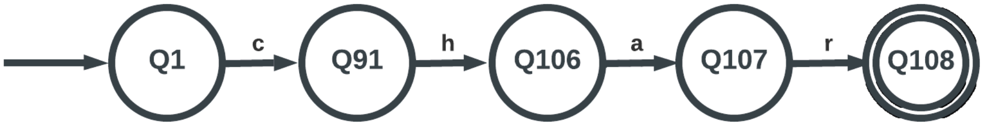
fig



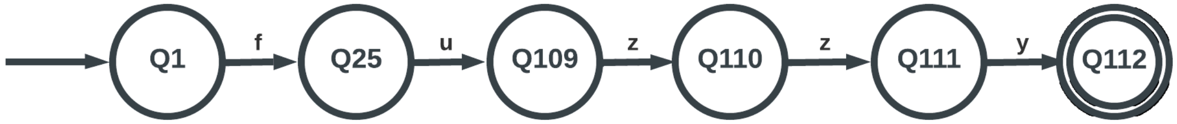
yarn



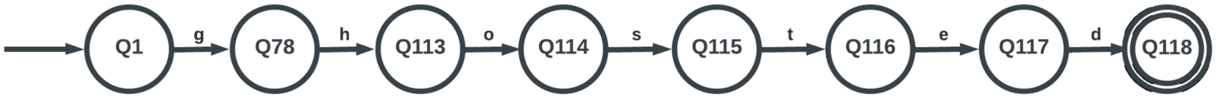
char



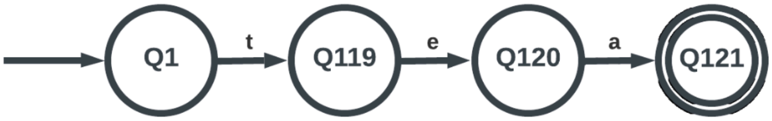
fuzzy



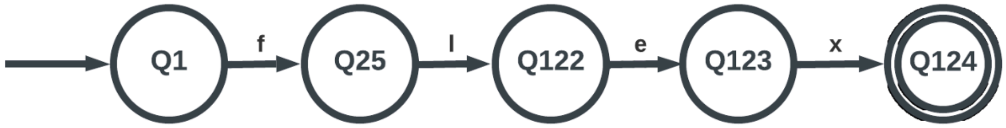
ghosted



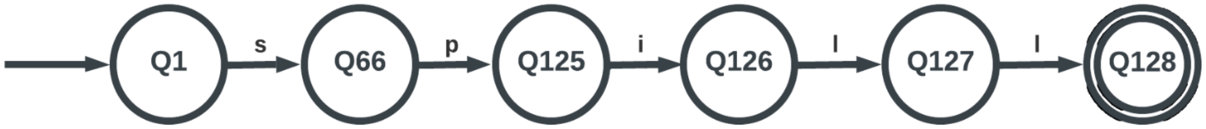
tea



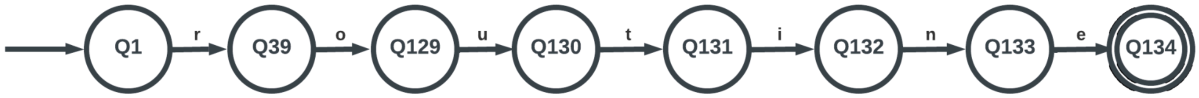
flex



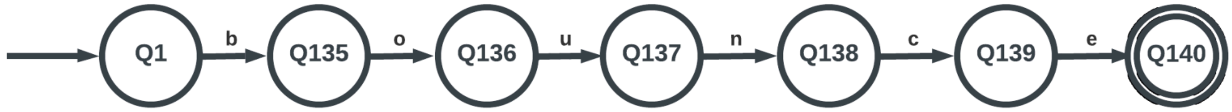
spill



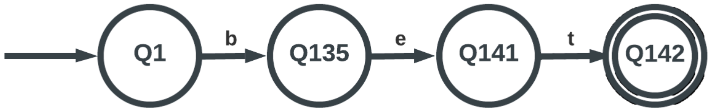
routine



bounce



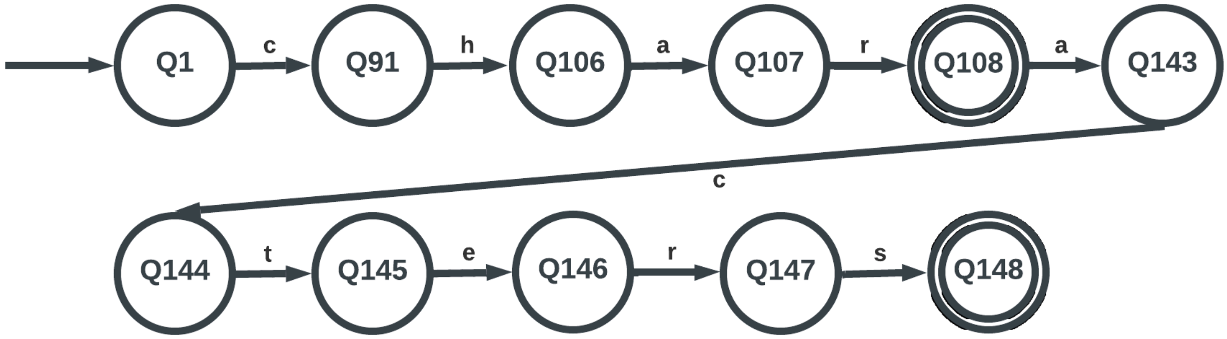
bet



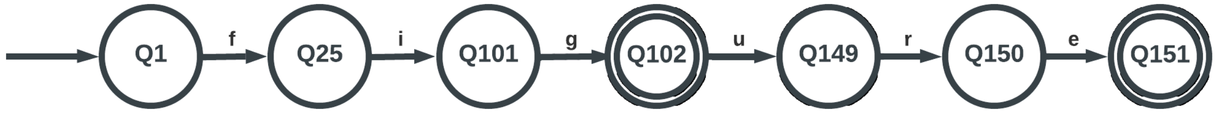
5. Noise words

Noise Word	Shorthand	Original Notation	Definition
acters	char	characters	Full notation for data type characters, often shorthanded as char, making "acters" act as a noise word. It can be typed for clearer reading and definition.
ure	fig	figure	Full notation for data type figure, often shorthanded as fig, making "ure" act as a noise word. It can be typed for clearer reading and definition.

characters



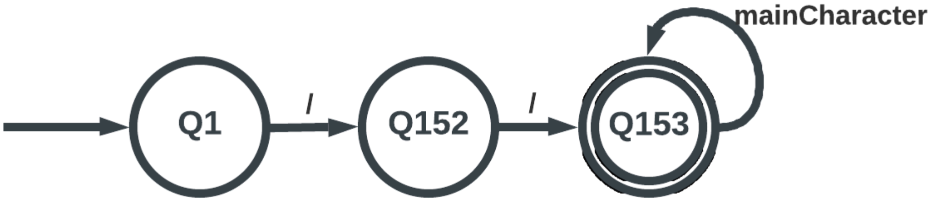
figure



6. Comments

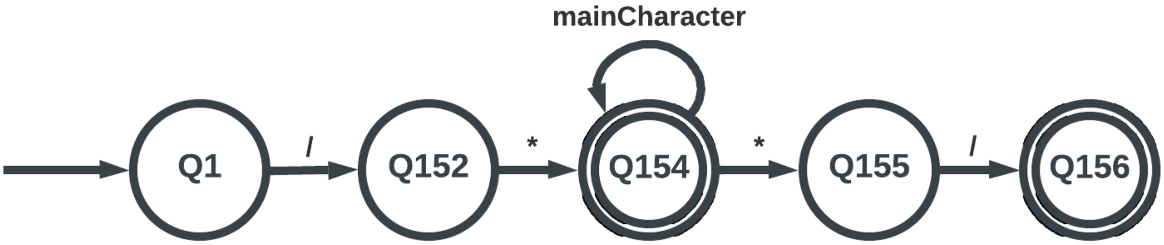
Single-line comments (//)

Single-line comments can be expressed using two consecutive forward slashes (//). Any characters (from the mainCharacter) after these slashes will be ignored by the compiler.



Multi-line comments (/* */)

Multi-line comments are used to document or describe a chunk of code. Multi-line comments can be open with a forward slash followed by an asterisk (/*). It can be closed using an asterisk followed by a forward slash (*/). The compiler will ignore anything inside the multi-line comment.



7. Blanks (spaces)

Blanks or white spaces in IYKYK programming language can be used on the following instances:

1. **Variable Declaration/Assignment Statement**
   To separate a data type, identifier, and value when declaring a variable.
   Examples:
   lit<space>myAge;
   myAge<optional space>=<optional space>21;

2. **Function Declaration**
   To separate ‘routine’ keyword, identifier, and code block.
   Example: routine<space>getAge()<optional space>{}

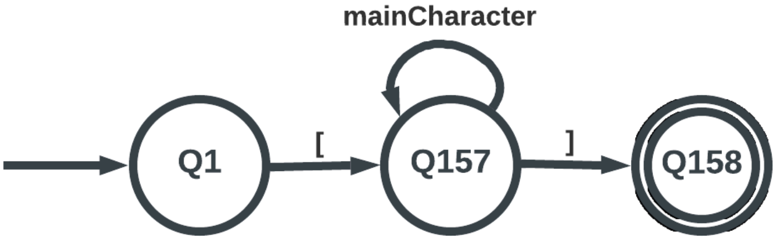
3. **Code Organization (not strict) -** involves the use of white spaces, including indentation, tabs, and empty lines. These white spaces are used to create visual separation between code blocks, enhancing readability and ease of maintenance.

In short, using a white spaces are only required after using reserved keywords. Other than that, white spaces are optional.

8. Delimiters and brackets

Square Brackets ([])

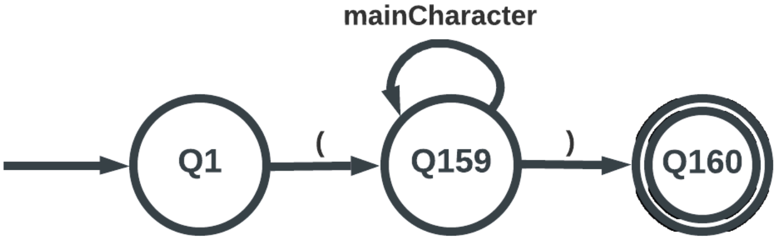
Primarily used to define lists, which are ordered collections of items. They enclose the list elements, separating them with commas. An open bracket should have a close bracket partner.



Parentheses (())

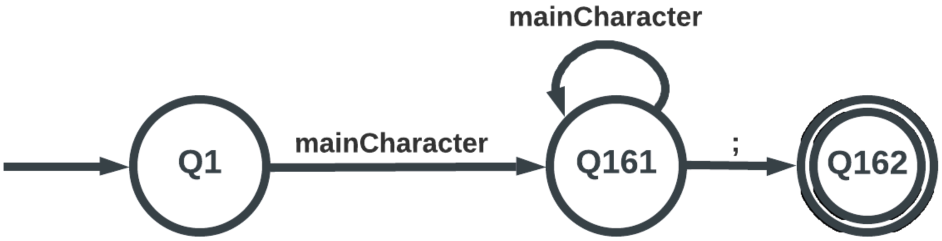
Parentheses serve multiple purposes and these are the following:

1. **Enclosing Expressions** - used to group expressions and control the order of operations in mathematical calculations.
2. **Defining Function** - enclose the arguments passed to functions during function calls.
3. **Invoking Method** - used to invoke methods on objects, specifying the method name and any arguments.



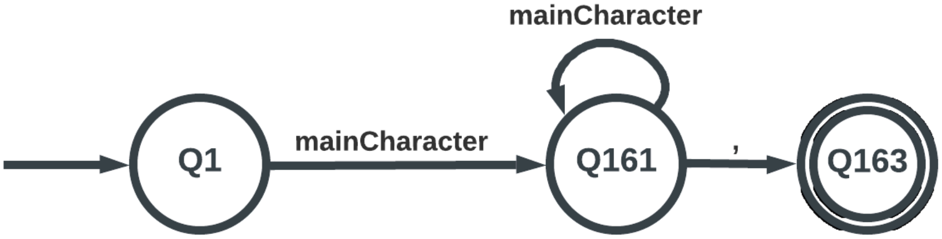
Semicolon (;)

Semicolon will be used to terminate a line of code and to separate multiple statements on a single line.



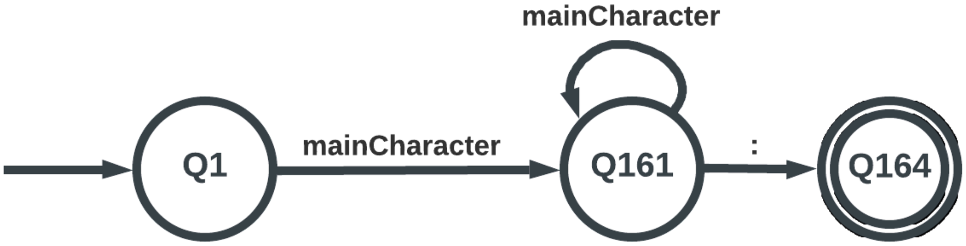
Comma (,)

Commas are used to separate the variable names in variable declaration, elements within an array, and function parameters.



Colon (:)

Colons are used to separate keys and values in objects.



9. Free-and-fixed-field formats

The proposed IYKYK programming language is a free-form programming language because its syntax is heavily influenced by the C programming language. IYKYK allows a flexible and free coding style where the placement of code statements such as declarations, control statements, and other elements stated in this documentation is not fixed or limited to any form. Hence, IYKYK codes can be formed and organized with line breaks, spacing, and indentation.

10. Expression

Arithmetic Expression Evaluation Rules

Priority Number	Operator Symbols	Description	Associativity
#1	*	Multiplication	Left-to-right
	/	Division	
	%	Modulo	

	^	Exponent	
#2	+	Addition	Left-to-right
	-	Subtraction	

Relational Operation Evaluation Rules

Priority Number	Operator Symbols	Description	Associativity
#1	>	Greater Than	Left-to-right
	<	Less Than	
	>=	Greater Than or Equal To	
	<=	Less Than or Equal To	
	==	Is Equal To	
	!=	Is Not Equal To	

Logical Operation Evaluation Rules

Priority Number	Operator Symbols	Description	Associativity
#1	&&	And	Left-to-right
#2	||	Or	Left-to-right
#3	!	Not	Left-to-right

Logical Expression (Boolean Expression) Evaluation Rules

Priority Number	Operator Symbols	Description	Associativity
#1	Relational Operation		Left-to-right
#2	Logical Operation		Left-to-right

Assignment Expression

Priority Number	Operator Symbols	Description	Associativity
#1	=	Assign	Right-to-left
	*=	Multiply then assign	

	/=	Divide then assign	
	+=	Add then assign	
	-=	Subtract then assign	
	%=	Assignment by remainder	
	^=	Assignment by exponent	

Unary Expression

Priority Number	Operator Symbols	Description	Sequencing
#1	++	Pre-Increment	Right-to-left
	--	Pre-Decrement	
	++	Post-Increment	Left-to-right
	--	Post-Decrement	

11. Statements

*Note: *this is NOT a Backus-Naur Form (BNF)**

Variable / Constant Declaration

Syntactical Format	Definition
<lit | fire> <identifier>;	
<lit | fire> <identifier> …, <identifier>;	
<lit | fire> <identifier> <assignmentOperation> <value>;	
<lit | fire> <identifier>, …, <identifier> <assignmentOperation><value>;	
<lit | fire> <identifier> <assignmentOperation> <value>, …, <identifier> <assignmentOperation> <value>;	

Input Statement

Syntactical Format	Sample Code
<lit | fire> varName = spill();	age = spill();
<varName> = spill();	age = spill();

Output Statement

Syntactical Format	Sample Code	Output
flex(<identifier>);	flex(age);	<value of age>
flex(“<yarn>”);	flex(“Sup, babe?”);	Sup, babe?
flex(“<yarn>” + “<yarn>”);	flex(“Sup, ” + “babe?”);	Sup, babe?
flex(“<yarn>” + “<identifier>”);	flex(“Sup ” + babeName);	Sup, <babeName>
flex(“<yarn>”...);	flex(“Sup babe #” + babeCount + “?”);	Sup babe #<babeCount>?

If

Syntactical Format	Sample Code	Output

yeet (<condition>) {
  // statement
}
	yeet (1 < 5) {
	  flex(“true”);
	}
		true

If-Else

Syntactical Format	Sample Code	Output

yeet (<condition>) {
  // statement
}
 yas {
  // default
}
	yeet (1 > 5) {
	  flex(“true”);
	}
	yas {
	  flex(“false”);
	}
		false

If-Else-If

Syntactical Format	Sample Code	Output

yeet (<condition>) {
  // statement
	yeet (1 == 2) {
	  yikes(“1 is equal to 2”);
		2 is greater than 1

}
yikes (<condition>) {
   // statement
}
	}
	yikes (2 > 1) {
	   yikes(“2 is greater than 1”);
	}	

Conditional Statement (If, else if, else)

Syntactical Format	Sample Code	Output

yeet (<condition>) {
   // statement
}
yikes (<condition>) {
   // statement
}
yas {
   // default
}
	yeet (1 == 2) {
	   yikes(“1 is equal to 2”);
	}
	yikes (2 < 1) {
	   yikes(“2 is greater than 1”);
	}
	yas {
	   yikes(“All of the conditions are false”);
	}
		All of the conditions are false

Nested If

Syntactical Format	Sample Code	Output

yeet (<condition>) {
   yeet (<condition>) {
      // yeet | statement
   }
}
	yeet (1 == 1) {
	  flex(“First level”);
	
	  yeet (2 == 2) {
	     flex(“Second level”);
	
	     yeet (3 == 3) {
	        flex(“Third level”);
	
	        yeet (4 == 5) {
	            flex(“Fourth level”);
	        }
	     }
	  }
	}
		First level
		Second level
		Third level

Nested If-Else

Syntactical Format	Sample Code	Output

yeet (<condition>) {
   yeet (<condition>) {
      // yeet-yas | statement
   }
   yas {
      // yeet-yas | statement
   }
}
yas {
   yeet (<condition>) {
      // yeet-yas | statement
   }
   yas {
      // yeet-yas | statement
   }
}
	yeet (1 > 2) {
	   yeet (true) {
	      flex(“Will not be printed”);
	   }
	   yas {
	      flex(“Will not be printed”);
	   }
	}
	yas {
	   yeet (3 == 3) {
	      flex(“Hello Mars!”);
	   }
	   yas {
	      flex(“Hello Jupiter!”);
	   }
	}
		Hello Mars

Nested If-Else-If

Syntactical Format	Sample Code	Output

yeet (<condition>) {
   yeet (<condition>) {
      // yeet-yikes | statement
   }
   yikes (<condition>) {
      // yeet-yikes | statement
   }
}
yikes (<condition>) {
   yeet (<condition>) {
      // yeet-yikes | statement
   }
   yikes (<condition>) {
      // yeet-yas | statement
   }
}
	yeet (1 == 1) {
	   flex(“What’s that flying?”);
	
	   yeet (2 > 2) {
	      flex(“No cap, it’s a dove!”);
	   }
	   yikes (3 == 3) {
	      flex(“No cap, it’s a bird!”);
	   }
	}
	yikes (false) {
	   flex(“What’s that crawling?”);
	
	   yeet (4 == 4) {
	      flex(“No cap, it’s a snake!”);
	   }
	   yikes (5 == 5) {
	      flex(“No cap, it’s a worm!”);
	   }
	}
		What’s that flying?
		No cap, it’s a bird!

Nested Conditionals

Syntactical Format	Sample Code	Output

yeet (<condition>) {
   yeet (<condition>) {
	fig score = 86;
	yeet (score >= 90) {
		Student got a B+ - Offer guidance for improvement

// yeet-yikes-yas
    // statement
  }
  yikes (<condition>) {
    // yeet-yikes-yas
    // statement
  }
  yas {
    // yeet-yikes-yas
    // statement
  }
}
yikes (<condition>) {
  yeet (<condition>) {
    // yeet-yikes-yas
    // statement
  }
  yikes (<condition>) {
    // yeet-yikes-yas
    // statement
  }
  yas {
    // yeet-yikes-yas
    // statement
  }
}
yas {
  yeet (<condition>) {
    // yeet-yikes-yas
    // statement
  }
  yikes (<condition>) {
    // yeet-yikes-yas
    // statement
  }
  yas {
    // yeet-yikes-yas
    // statement
  }
}
	yeet (score >= 95) {
	    flex("Student got an A+ - Congratulate the student");
	  }
	  yikes (score >= 90) {
	    flex("Student got an A - Encourage the student");
	  }
	  yas {
	    flex("Student got an A- - Provide some feedback");
	  }
	}
	yikes (score >= 80) {
	  yeet (score >= 85) {
	    flex("Student got a B+ - Offer guidance for improvement");
	  }
	  yikes (score >= 80) {
	    flex("Student got a B - Discuss areas for improvement");
	  }
	  yas {
	    flex("Student got a B- - Suggest ways to do better next time");
	  }
	}
	yas {
	  yeet (score >= 70) {
	    flex("Student got a C+ - Encourage seeking help if needed");
	  }
	  yikes (score >= 60) {
	    flex("Student got a C - Discuss the importance of studying");
	  }
	  yas {
	    flex("Student got below a C - Offer support and strategies for improvement");
	  }
	}	

Looping Statement

Syntactical Format	Sample Code	Output

relapse (
   as: <var>,
   worse: <integer>,
   recover: <integer>
) { // statement }
	relapse (
	   as: x,
	   worse: 1,
	   recover: 5
	) {
	   flex(“Loading ” + x + “/5”)
	}
		Loading 1/5
		Loading 2/5
		Loading 3/5
		Loading 4/5
		Loading 5/5

Nested Loop

Syntactical Format	Sample Code	Output

relapse (
   as: <var>,
   worse: <integer>,
   recover: <integer>
) {
   relapse (
      as: <var>,
      worse: <integer>,
      recover: <integer>
) {
      // statement
   }
}
	relapse (
	   as: i,
	   worse: 1,
	   recover: 5
	) {
	   relapse (
	      worse: 1
	      recover: i
	   ) {
	      flex("* ");
	   }
	   flex(" ");
	}
		*
		* *
		* * *
		* * * *
		* * * * *

Assignment Statement

Syntactical Format	Sample Code	Variable Value After Assignment

<var> = 3;	lit var = 5;
	var = 3;
		3

<var> += 3;	lit var = 5;
	var += 3;
		5

<var> = <var1>	lit cartoon1 = “Hello Kitty”; lit cartoon2 = “Mr. Bean”; cartoon2 = cartoon1;	Hello Kitty // cartoon1 Hello Kitty // cartoon2

<var> += <var1>	lit myMoney = 500.50;
	lit salary = 25000.25;
		25500.75 // myMoney
		25000.25 // salary

	myMoney += salary;	

<var> = <expression>	lit exponent = 5, number = 5, answer;
	answer = number ^ exponent;
		3125 // answer

Function Declarations

Syntactical Format	Sample Code	Function Call

routine <identifier>() {
  // statement
}
	routine greet() {
	  flex(“Hello World!”);
	}
		greet();

routine <identifier>(x, …) {
  // statement
}
	routine add(n1, n2) {
	  lit answer = n1 + n2;
	  flex(“The answer is ” + answer)
	}
		add(5, 6) // 11

Multi-Threaded Function Declaration

Syntactical Format	Sample Code	Function Call

delay routine <identifier>() {
  // statement
}
	delay routine getUserInfo() {
	  // assume slow and CPU intensive definition
	}
		chill getUserInfo();
		// will chill on that function until finished

delay routine <identifier>(args, …) {
  // statement
}
	delay routine getTweet(param1, param2) {
	  // assume slow and CPU intensive definition
	}
		chill getTweet(param1, param2);
		// will chill on that function until finished

HTML/CSS Instantiation

Syntactical Format	Sample Code	Result

htmlize {
  <HTMLTag>: {
    <attributes>:<value>
  }
}
	htmlize {
	  h1 {
	    className: "title",
	    style: "background-color: red"
	    id: "my-title",
	    content: “Grape!”
	  }
	  p {
		Grape!
		IYKYK is in the bag!

	className: "desc", style: "background-color: blue" id: "description", content: “IYKYK is in the bag!” } }	

11. IYKYK New Principles Backus Naur Form (🚀)

VARIABLE AND CHARACTERS

```
<VAR_NAME> ::= "_"* <ALPHABET>+ "_"* <DIGIT>* "_"* | "_"+

<VAR_LIST> ::= <VAR> ("," <VAR_LIST>)*

<ALPHABET> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
      "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
      "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" |
      "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
      "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

<DIGIT> ::= "0" | <NON_ZERO>

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

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

<ASCII_VALUES> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
      | "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16" | "17" |
      "18" | "19" | "20" | "21" | "22" | "23" | "24" | "25" | "26" |
```

"27" | "28" | "29" | "30" | "31" | "32" | "33" | "34" | "35" |
"36" | "37" | "38" | "39" | "40" | "41" | "42" | "43" | "44" |
"45" | "46" | "47" | "48" | "49" | "50" | "51" | "52" | "53" |
"54" | "55" | "56" | "57" | "58" | "59" | "60" | "61" | "62" |
"63" | "64" | "65" | "66" | "67" | "68" | "69" | "70" | "71" |
"72" | "73" | "74" | "75" | "76" | "77" | "78" | "79" | "80" |
"81" | "82" | "83" | "84" | "85" | "86" | "87" | "88" | "89" |
"90" | "91" | "92" | "93" | "94" | "95" | "96" | "97" | "98" |
"99" | "100" | "101" | "102" | "103" | "104" | "105" | "106" |
"107" | "108" | "109" | "110" | "111" | "112" | "113" | "114" |
"115" | "116" | "117" | "118" | "119" | "120" | "121" | "122" |
"123" | "124" | "125" | "126" | "127"

VARIABLE AND CONSTANT DECLARATION

<LIT_DEC> ::= “lit ” <VAR_NAME> “=” (<FIG_VALUES> | <FUZZY_VALUES> | <CHAR_VALUES> | <YARN_VALUES> | <TEA_VALUES> | ghosted)”;”

<FIRE_DEC> ::= “fire ” <VAR_NAME> “=” (<FIG_VALUES> | <FUZZY_VALUES> | <CHAR_VALUES> | <YARN_VALUES> | <TEA_VALUES>)”;”

<**FIG_VALUES**> ::= <DIGIT>+ | "-" <NON_ZERO>+

<FUZZY_VALUES> ::= <DIGIT>+ ”.” <DIGIT>*

<**CHAR_VALUES**> ::= "'" <ALPHABET> "'" | "'" <DIGIT> "'" | "'" <SPECIAL_CHARS> "'" | <ASCII_VALUES>

<YARN_VALUES> ::= <CHAR_VALUES>+ “ ”* <CHAR_VALUES>*

<**TEA_VALUES**> ::= "real" | "cap"

CONDITIONALS

```
<INEQUALITY_OP> ::= "<" | ">" | "==" | "<=" | ">="

<LOGIC_OP> ::= "||" | "&&"

<NEGATE> ::= "!"

<LOGIC_EXPRESSION> ::= ((<INEQUALITY> | <TEA_VALUES>) | <NEGATE>
      <LOGIC_EXPRESSION>) (<LOGIC_OP> ((<LOGIC_EXPRESSION> |
      <TEA_VALUES>) | <NEGATE> <LOGIC_EXPRESSION>))*


<CONDITIONALS> ::= <YEET> <YIKES>* <YAS>*

<YEET> ::= “yeet(“ <LOGIC_EXPRESSION> “){“ <STMT> “}”

<YIKES> ::= “yikes (” <LOGIC_EXPRESSION> “){“ <STMT> “}”

<YAS> ::= “yas {” <STMT> “}”
```

🚀 LOOPING

```
<RELAPSE> ::= “relapse (” <VAR_NAME> “,worse” <FIG_VALUE> “,recover”
      <FIG_VALUE> “){“ <STMT> “)” | “relapse ( as” <VAR_NAME> “,worse”
      <FIG_VALUE> “,recover” <FIG_VALUE> “){“ <STMT> “)”
```

🚀 STEPWISE INCREMENTATION AND DECREMENTATION

```
<STEPWISE_STATEMENT> ::= <INCREMENT_STATEMENT> | <DECREMENT_STATEMENT>
<INCREMENT_STATEMENT> ::= <VAR> ">>" <FIG_VALUES> | <EXPRESSION> ";"
<DECREMENT_STATEMENT> ::= <VAR> "<<" <FIG_VALUES> | <EXPRESSION> ";"
```

🚀 FUNCTION CONTRACTORS

```
<FUNCTION CONTRACTORS> ::= “BET” “(“ <LOGIC_EXPRESSION> “,” “<YARN>”
      “)” ;
```

🚀 DYNAMIC CALLBACK

```
<CALLBACK_FUNCTION_DECLARATION> ::= "DELAY" <FUNCTION_NAME> "("
     <PARAMETER_LIST> ")" <CODE_BLOCK>
<CALLBACK_INVOCATION> ::= "CHILL" <FUNCTION_NAME> "(" <ARGUMENT_LIST>
     ")"

<FUNCTION_NAME> ::= <IDENTIFIER>
<PARAMETER_LIST> ::= <PARAMETER> ("," <PARAMETER_LIST>)*
<ARGUMENT_LIST> ::= <EXPRESSION> ("," <ARGUMENT_LIST>)*
<CODE_BLOCK> ::= "{" <STATEMENT_LIST> "}"
<STATEMENT_LIST> ::= <STATEMENT>+
```

🚀 IMPROVED SIMPLIFIED LOOPING

```
<RELAPSE> ::= “relapse (” <VAR_NAME> “,worse” <FIG_VALUE> “,recover”
     <FIG_VALUE> “){“ <STMT> “)” | “relapse ( as” <VAR_NAME> “,worse”
     <FIG_VALUE> “,recover” <FIG_VALUE> “){“ <STMT> “)”
```

🚀 HTML SUPPORT

```
<HTML_SUPPORT> ::= “htmlize” { <HTML_COMPONENT> }

<HTML_COMPONENT> ::=  <HTML_TAGS> { “<YARN>” : “<YARN>”} |
     (<HTML_COMPONENT> “,”)*

<HTML_TAGS> ::= "<body>", "<h1>", "<h2>", "<h3>", <h4>, <h5>, <h6>,
"<p>", "<a>", "<img>", "<div>", "<span>", "<ul>", "<ol>", "<li>",
"<br>", "<hr>", "<em>", "<strong>", "<blockquote>", "<cite>",
"<code>", "<pre>", "<i>", "<b>", "<u>", "<small>", "<sub>", "<sup>",
"<abbr>", "<address>", "<var>", "<samp>", "<header>", "<nav>",
"<main>", "<section>", "<article>", "<aside>", "<footer>",
"<address>", "<a>", "<em>", "<strong>", "<small>", "<s>", "<cite>",
"<q>", "<dfn>", "<abbr>", "<data>", "<time>", "<code>", "<var>",
"<samp>", "<kbd>", "<sub>", "<sup>", "<i>", "<b>", "<u>", "<mark>",
"<ruby>", "<span>", "<br>", "<img>", "<iframe>", "<embed>", "<param>",
```

"<video>", "<audio>", "<source>", "<track>", "<map>", "<area>", "<a>", "<table>", "<caption>", "<colgroup>", "<col>", "<thead>", "<tbody>", "<tfoot>", "<tr>", "<th>", "<td>", "<form>", "<label>", "<input>", "<button>", "<select>", "<datalist>", "<optgroup>", "<option>", "<textarea>", "<output>", "<progress>", "<meter>", "<fieldset>", "<legend>", "<details>", "<summary>", "<dialog>", "<script>", "<noscript>", "<template>", "<slot>", "<canvas>", "<svg>", "<math>"

🚀 UNDEFINED VALUE SAFETY

<UNDEFINED_VALUE_SAFETY> ::= <VAR_NAME> "#" "." <VAR_NAME> ("#" "." <VAR_NAME>)*

Documentation for “**IYKYK**” Programming Language

Group 2 Members:

Alamag, Jose Luis
Calendario, Mark Kenneth
Caspe, Mark Vincent
Favorito, Vince Lennard
Lalis, Reygine
Villegas, Daniel

Principles of Programming Languages
COSC 30063
BSCS 3-1N