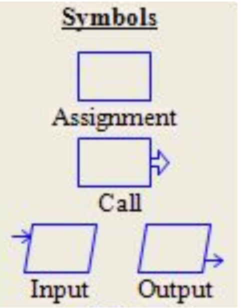
RAPTOR Syntax and Semantics By Lt Col Schorsch



Program - an ordered collection of instructions that, when executed, causes the computer to behave in a predetermined manner.

Variable - A variable names a memory location. By using that variable's name you can store data to or retrieve data from that memory location.

A **variable** has 4 properties: ❶ a name, ❷ a memory location, ❸ a data type, ❹ a value. You can assign a value to a variable using an assignment statement (see below). RAPTOR variables are declared on first use, they must be assigned a value on first use and based on that value it's *data type* will be Number, String, or an Array of Numbers.

Data Type - A Data Type is the name for a group of data values with similar properties.

A Data Type has 4 properties: ❶ a name, ❷ a set of values, ❸ a notation for *literals* of those values, ❹ operations and functions which can be performed on those values.

RAPTOR has two simple data types: Number and String (Array data types are described later)

Type name	Literal Values	Operations grouped from lowest to highest precedence
Number	-32, 0, 1, 49, etc. -2.1, 3.1415, etc.	[=,<,<=,>,>=,/=,!=], [+, -], [*, /, rem, mod], [**,^]
String	"Hello", "Bob", etc.	[=,<,<=,>,>=,/=,!=], [+]

Operator – An operator directs the computer to perform some computation on data.

Operators are placed between the data (operands) being operated on (i.e. **X / 3**, **Y + 7**, **N < M**, etc.)

basic math operators:	+, -, *, /, ^, **, rem, mod	+, -, *, / are defined as one would expect, ** and ^ are exponentiation, ex 2**4 is 16, 3^2 is 9 rem (remainder) and mod (modulus) return the remainder (what is left over) when the right operand divides the left operand, ex 10 rem 3 is 1, 10 mod 3 is 1
Concatenation operator:	+	Joins strings and numbers (i.e. "Average is " + (Total / Number))

The following operators are only used in decisions (see Selection and Iteration)

Relational operators:	=, !=, /=, <, >, >=, <=	Used to compare numbers and strings, = is equals, != and /= are both not equals. <, >, >=, <= are defined as expected. The result of a relational comparison is a Boolean value.

Logical operators: and, or, not, xor

Expression	Result
True and True	True
True and False	False
False and True	False
False and False	False

Expression	Result
True o	Tru
True or True	Tru
FalseroFalse	Tru
False or True r False	Fals e

Expression	Result
Not(True)	False
Not(False)	True

xor is true when either operand is true (but not when both operands are true).

Function – A function performs a computation on data and returns a value.

Functions use parentheses to indicate their data (i.e. **sqrt(4.7)**, **sin(2.9)**, etc.)

Basic math: sqrt, log, abs, ceiling, floor
- sqrt returns the square root, ex sqrt(4) is 2
- log returns the natural logarithm, ex log(e) is 1
- abs returns the absolute value, ex abs(-9) is 9
- ceiling rounds up to a whole number, ex ceiling(3.14159) is 4
- floor rounds down to a whole number, ex floor(10/3) is 3

Trigonometry: sin, cos, tan, cot, arcsin, arccos, arctan, arccot
- Angles are in radians, ex sin(pi) is 0.
- arctan and arccot are the two parameter versions of those functions.
- (i.e. arctan(X/Y) is written in RAPTOR as arctan(X,Y)).

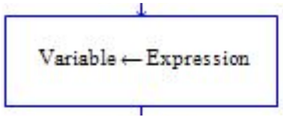
Miscellaneous: Length_Of
- Length_Of returns the number of characters in a string
- ex `Name ← "Stuff"` followed by `Length_Of(Name)` is 5
- (also returns the number of elements in an array which you will learn later)

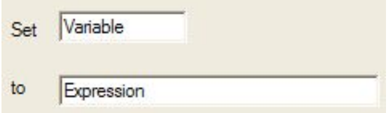
Random
- Returns a random number between [0.0,1.0)
- (`Random * X + Y` extends the range by X and shifts it by Y)

Assignment Statement - An assignment statement is used to evaluate an *expression* and store the results in a *variable*. The *expression* is on the right hand side of the assignment operator, ⟵.

An *expression*'s value (after it is evaluated) is stored in the *variable* on the left hand side of the ⟵ operator. An *expression* must evaluate to a value of the same data type as the *variable* in which it is being stored.

Syntax:
Variable ⟵ *Expression*

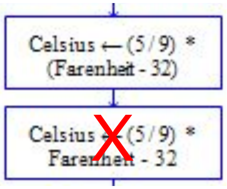



Set: Variable
to: Expression

An *expression* is either a *variable*, a *literal*, or some *computation* (such as 3.14 * Radius). A *literal* (such as 2.143, 42, "Help") evaluates to itself. A *variable* evaluates to the data stored at its memory location. Evaluating a *computation* involves evaluating the literals, variables, operators and functions in the expression.

`Age ← 21`	The value 21 is stored in variable Age's memory location
`Count ← Count + 1`	The value that is stored in Count's memory location is incremented by 1
`Force ← Mass * Acc`	Mass and Acc are multiplied together, the product is stored in variable Force
`Delta_X ← abs(X2 - X1)`	Take the absolute value difference and store it in Delta_X
`Name ← "Schorsch"`	Assigns the string "Schorsch" to the variable Name's memory location

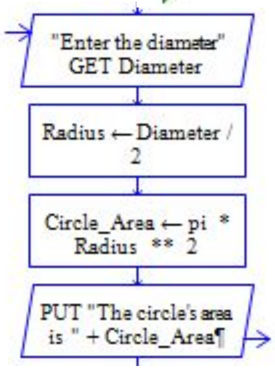
Order of operations matters!
Precedence levels from lowest to highest
[=,<,<=,>,>=,/=,!=], [+, -], [*, /, rem, mod], [**,^]


Celsius ⟵ (5 / 9) * (Farenheit - 32)
Celsius ⟵ (5 / 9) * Farenheit - 32
X

Correct Equation

Incorrect Equation

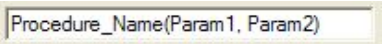
Circle Area program:
Given a diameter this program computes and displays the area of a circle with that diameter

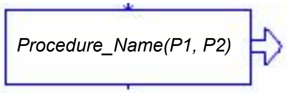

"Enter the diameter" GET Diameter
Radius ⟵ Diameter / 2
Circle_Area ⟵ pi * Radius ** 2
PUT "The circle's area is " + Circle_Area¶

Procedure Call - A procedure is a set of executable statements that have been given a name.

Calling a procedure executes the statements associated with that procedure.

Procedure_name (Parameter 1, Parameter 2, etc.)


Procedure_Name(Param1, Param2)



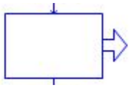
The number and order of parameters in the call must match the expected number and order.
The data types of the parameters in the call must match the expected data types of the parameters.
Procedure parameters can be used to give (supply) a procedure with data or can accept (receive) data.
Parameters must be variables if they receive a value.
Parameters can be an expression (computation), variable or literal if they supply a value.

`Delay_for(0.2)`	delays execution for 2/10ths of a second
`Clear_Console`	erases the master console contents
`Draw_Circle(X, Y, 7, Blue)`	draws a blue circle at location X,Y with a radius of 7

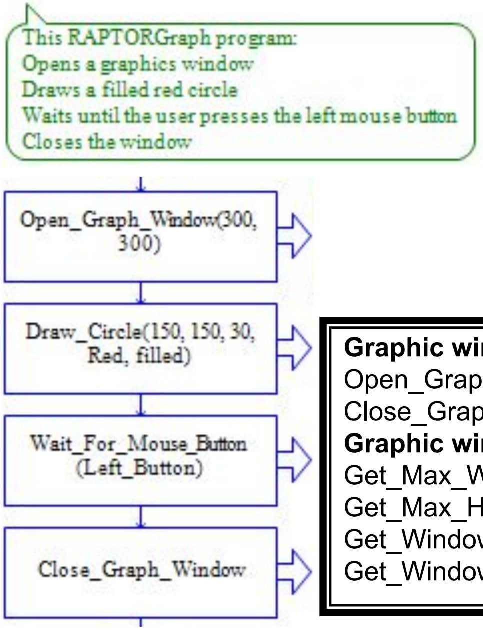
RAPTORGraph Syntax and Semantics

RAPTORGraph is a collection of procedures and functions that a RAPTOR programmer can use to create a graphics window, draw and animate graphical objects in that window, and interact with the graphics window using the keyboard and mouse.

Procedure calls occur only in call symbols.



Function calls return a value and therefore can occur anywhere a value can occur. (i.e. in assignment, decision, and output statements and as procedure call parameters.)



This RAPTORGraph program:
Opens a graphics window
Draws a filled red circle
Waits until the user presses the left mouse button
Closes the window

Open_Graph_Window(300, 300)

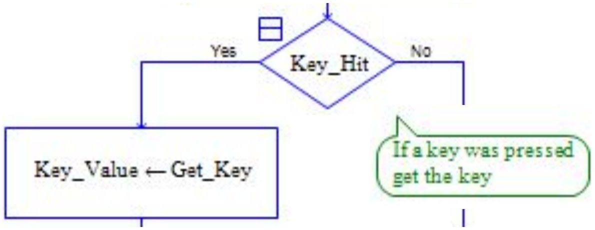
Draw_Circle(150, 150, 30, Red, filled)

Wait_For_Mouse_Button (Left_Button)

Close_Graph_Window

Graphic window opening and closing procedures
Open_Graph_Window(X_Size, Y_Size)
Close_Graph_Window
Graphic window "size" functions
Get_Max_Width -> returns available screen pixel width
Get_Max_Height -> returns available screen pixel height
Get_Window_Width -> returns current window pixel width
Get_Window_Height -> returns current window pixel height

Keyboard input procedure
Wait_For_Key
Keyboard input functions
Key_Hit -> returns True / False (whether a key was pressed)
Get_Key -> returns the numeric ASCII value of the pressed key
Get_Key_String -> returns a string value of the pressed key

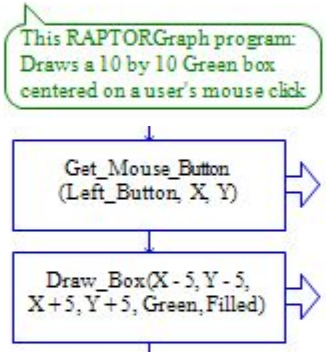


Key_Value ← Get_Key

If a key was pressed get the key

Drawing procedures
Put_Pixel(X, Y, Color)
Draw_Line(X1, Y1, X2, Y2, Color)
Draw_Box(X1, Y1, X2, Y2, Color, Filled/Unfilled)
Draw_Circle(X, Y, Radius, Color, Filled/Unfilled)
Draw_Ellipse(X1, Y1, X2, Y2, Color, Filled/Unfilled)
Draw_Arc(X1, Y1, X2, Y2, StartX, StartY, EndX, EndY, Color)
Clear_Window(Color)
Flood_Fill(X, Y, Color)
Display_Text(X, Y, String Expression, Color)
Display_Number(X, Y, Number Expression, Color)

Mouse input procedures
Wait_for_Mouse_Button(Which_Button)
Get_Mouse_Button(Which_Button, X, Y)
Mouse input functions
Mouse_Button_Pressed(Which_Button) –> returns True / False
Mouse_Button_Released(Which_Button) –> returns True / False
Get_Mouse_X –> returns X coordinate of mouse location
Get_Mouse_Y –> returns Y coordinate of mouse location



This RAPTORGraph program: Draws a 10 by 10 Green box centered on a user's mouse click

Get_Mouse_Button (Left_Button, X, Y)

Draw_Box(X - 5, Y - 5, X + 5, Y + 5, Green, Filled)

RAPTORGraph Colors
Black, Blue, Green, Cyan, Red, Magenta, Brown, Light_Gray, Dark_Gray, Light_Blue, Light_Green, Light_Cyan, Light_Red, Light_Magenta, Yellow, White
(Get_Pixel returns 0 for Black, 1 for Blue, …,16 for White)

Graphics window query function
Get_Pixel(X, Y) -> returns the number code for the color of the pixel at (X, Y)

How to animate an object in RAPTORGraph
Place the following inside of a loop

Draw some an object relative to an X,Y point with the drawing procedures
Delay_For some small time period
Draw the object again in white (i.e. erase it)
Update the X,Y point where you are drawing by some small offset

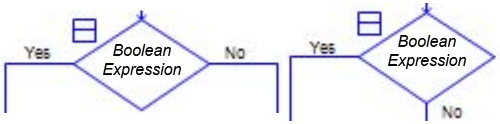
RAPTOR Syntax and Semantics – Selection and Iteration Control Structures

Decision -

A decision is part of a Selection or Iteration (loop) statement. A decision symbol (its value during execution) determines which way execution will continue. Use relational operators (and logical operators) to get a Boolean value for the decision.

Relational: =, <, <=, >, >=, /=, !=

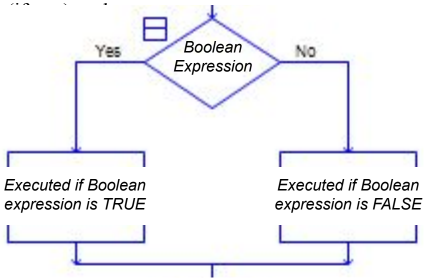
Logical: and, or, not, xor



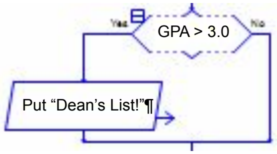
Selection Statement -

A selection statement is used to decide whether or not to do something, or to decide which of several things

If the *Boolean Expression* is TRUE,
execute the left hand path
otherwise
execute the right hand path

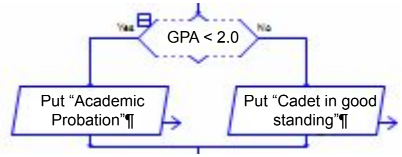


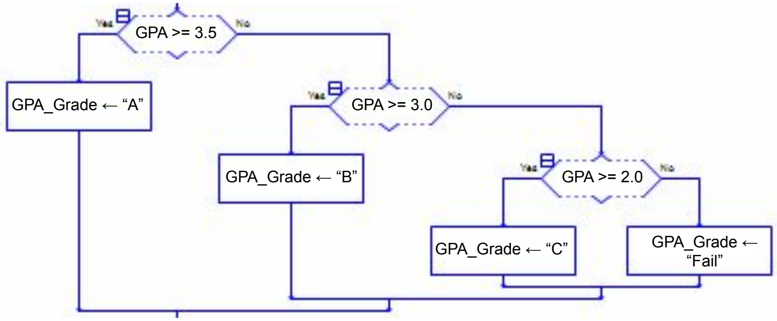
Yes
Boolean Expression
No
Executed if Boolean expression is TRUE
Executed if Boolean expression is FALSE



If the value of the variable GPA is greater than 3.0
then execute the statement
`Put("Dean's List")`
otherwise do nothing

If a student's GPA is less than 2.0
then execute the statement
`Put("Academic probabtion")`
otherwise execute the statement
`Put("Cadet in good standing")`

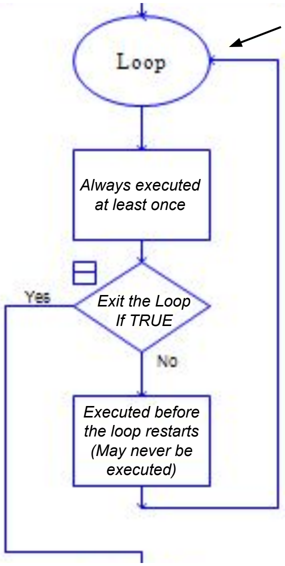




This last example requires several decision statements as there are several decisions (more than two possible paths). The code assigns a nominal "grade" based on a student's GPA. The "pattern" of these selection statements is called cascading selections.

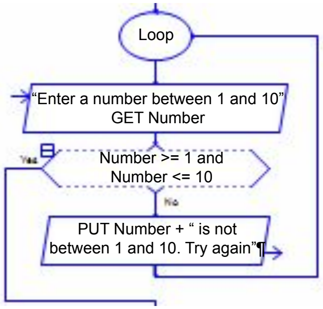
Iteration Statement (loop statement) –

An Iteration statement enables a group of statements to be executed more than once. Use **I.T.E.M** (Initialize, Test, Execute, and Modify) to ensure your loop (and **loop control variable**) are correct.



Loop
Always executed at least once
Yes
Exit the Loop If TRUE
No
Executed before the loop restarts (May never be executed)

A Condition Controlled Loop (basic loop) repeats its statements until a condition (the decision statement) becomes true.



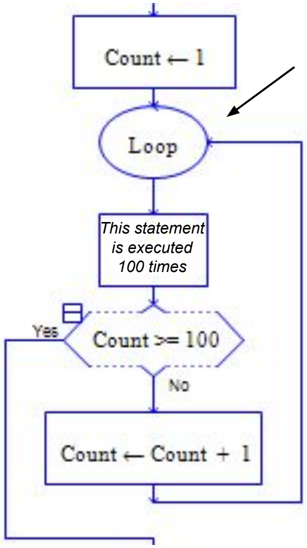
Loop
"Enter a number between 1 and 10"
GET Number
Yes
Number >= 1 and Number <= 10
No
PUT Number + " is not between 1 and 10. Try again"¶

Initialize (and modify) the loop control variable

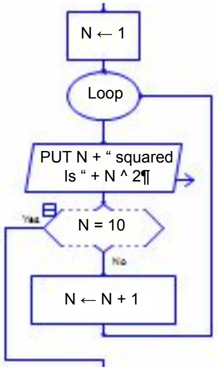
Test the loop control variable

Execution step

The validation loop above will continue to execute until the user enters a number between 1 and 10. `Number` is the loop control variable.



A Count Controlled Loop repeats its statements a fixed number of times. (This executes the loop 100 times because of the decision: `Count >= 100`).



Initialize the loop control variable (above the loop)

Execution step

Test the loop control variable

Modify the loop control variable

The count controlled loop above executes exactly 10 times (it displays the numbers 1 through 10 and the squares of those numbers). `Count` is the loop control variable.

RAPTOR Syntax and Semantics - Arrays

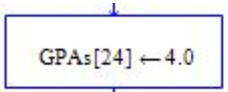
Array variable -

Array variables are used to store many values (of the same type) without having to have many variable names. Instead of many variables names a count-controlled loop is used to gain access (index) the individual elements (values) of an array variable.

RAPTOR has one and two dimensional arrays of numbers.
A one dimensional array can be thought of as a sequence (or a list).
A two dimensional array can be thought of as a table (grid or matrix).

To create an array variable in RAPTOR, use it like an array variable. i.e. have an index, ex. Score[1], Values[x], Matrix[3,4], etc.

All array variables are indexed starting with 1 and go up to the largest index used so far. RAPTOR array variables grow in size as needed.



The assignment statement
`GPAs[24] ← 4.0`
assigns the value `4.0` to the 24th element of the array GPAs.
If the array variable GPAs had not been used before then the other 23 elements of the GPAs array are initialized to 0 at the same time.
i.e. The array variable GPAs would have the following values:

1	2	3	4 …																			… 23	24
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4.0

The initialization of previous elements to `0` happens only when the array variable is created. Successive assignment statements to the GPAs variable affect only the individual element listed.
For example, the following successive assignment statements

```
GPAs[20]  ←  1.7
GPAs[11]  ←  3.2
```

would place the value `1.7` into the 20th position of the array, and would place the value `3.2` into the 11th position of the array.
i.e. `GPAs[20] ← 1.7`
`GPAs[11] ← 3.2`

1	2	3	4 …																			… 23	24
0	0	0	0	0	0	0	0	0	0	3.2	0	0	0	0	0	0	0	0	1.7	0	0	0	4.0

An array variable name, like GPAs, refers to ALL elements of the array. Adding an *index* (position) to the array variable enables you to refer to any specific element of the array variable.

Two dimensional arrays work similarly.
i.e. Table[7,2] refers to the element in the 7th row and 2nd column.

Individual elements of an array can be used exactly like any other variable. E.g. the array element GPAs[5] can be used anywhere the number variable X can be used.

The `Length_Of` function can be used to determine (and return) the number of elements that are associated with a particular array variable.

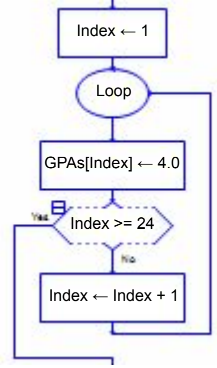
For example, after all the above, `Length_Of(GPAs)` is `24`.

Array variables in action-

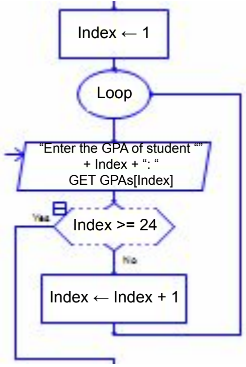
Arrays and count-controlled loop statements were made for each other.

Notice in each example below the connection between the Loop Control Variable and the array index!
Notice how the `Length_Of` function can be used in the count-controlled loop test!
Notice that each example below is a count-controlled loop and has an Initialize, Test, Execute, and Modify part (I.T.E.M)!

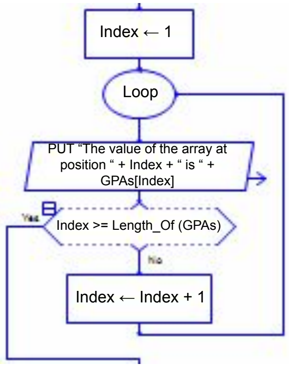
Assigning values to an array variable



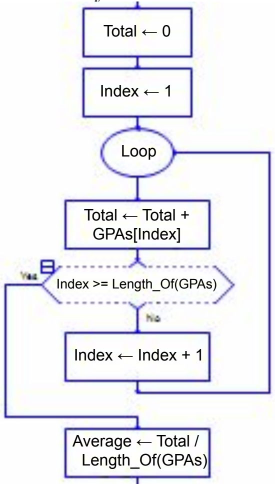
Reading values into an array variable



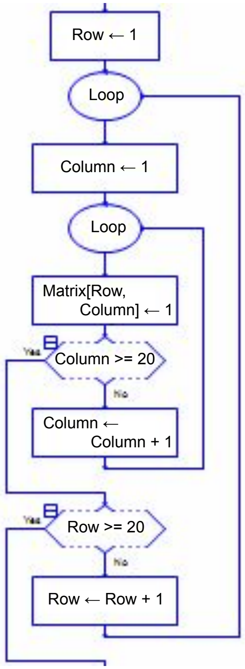
Writing out an array variable's values


Index ← 1
Loop
PUT "The value of the array at position " + Index + " is " + GPAs[Index]
Yes
Index >= Length_Of (GPAs)
No
Index ← Index + 1

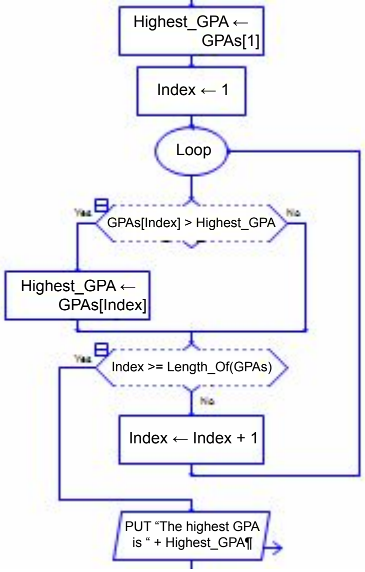
Computing the total and average of an array variable's values



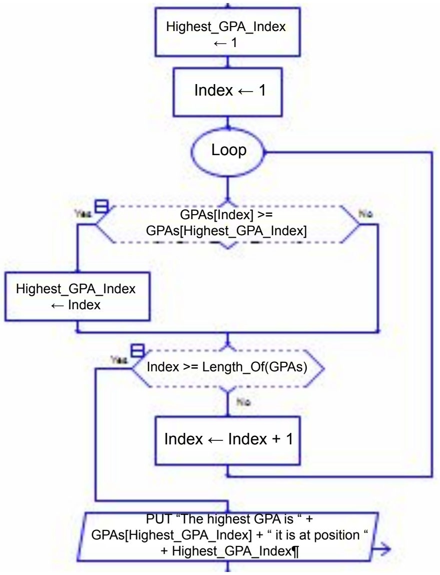
Initialize the elements of a two dimensional array (A two dimensional array requires two loops)



Find the largest value of all the values in an array variable


Highest_GPA ← GPAs[1]
Index ← 1
Loop
Yes
GPAs[Index] > Highest_GPA
No
Highest_GPA ← GPAs[Index]
Yes
Index >= Length_Of(GPAs)
No
Index ← Index + 1
PUT "The highest GPA is " + Highest_GPA¶

Find the INDEX of the largest value of all the values in an array variable


Highest_GPA_Index ← 1
Index ← 1
Loop
Yes
GPAs[Index] >= GPAs[Highest_GPA_Index]
No
Highest_GPA_Index ← Index
Yes
Index >= Length_Of(GPAs)
No
Index ← Index + 1
PUT "The highest GPA is " + GPAs[Highest_GPA_Index] + " it is at position " + Highest_GPA_Index¶