Pseudo-code Standard for Code-Translation Program ver. 3.0.3

Latest Update: 2020/4/5

This document is an explanatory document about the syntax of the pseudocode used in program Pseudocode Translator. When writing the pseudocode in such style, please note the following details:

- Pseudocode (syntax, structure) does not form a complete system-such as Java is an object-oriented programming language; pseudocode can only be regarded as a logical expression similar to programming.
- Because of the previous reason, the syntax design of pseudocode does not involve concepts such as classes, packages, processes, and threads; in other words, pseudocode can only express some relatively simple (and fragmented) operations on algorithms or data structures.
- The standard is designed on a Java-like grammar.
- The program does not ensure that the translated code would be 100% executable, especially the translated C code.
- The syntax environment of pseudo code is relatively loose, and multiple expressions may point to one meaning. You can feel this in the grammar rules below.
- The pseudo-code is NOT case sensitive. For example, "PRINT" is same as "print", "pRinT" and so on.
- One or more of the spaces between the words in the pseudocode represent the same result; however, in some statements, has one space or has no spaces between characters is very important.
- Indentation before each line is not necessary; however, it will greatly facilitate your reading; the translated code's line indentation is based on your indentation input.

This document has 16 sections and covers almost all uses of this pseudocode, thank you for reading this manual before using Pseudocode Translator.

1. Print and Hello World

Use the keyword "**PRINT**" to print out a word or a sentence.

All words after "PRINT" in a line will be printed out by the program. If you want to print the value of some variables, please use the keyword "**OUTPUT**" (see part 3).

Example
PRINT Hello World
Output Result
Hello World

2. Define variables and constants, data types

A variable or a constant should be defined before use them. It is a common but important concept in the majority of programming language. The name, value and data type of a variable or a constant could be defined at the same time.

Firstly, here are three basic data types of the pseudo-code: **Number**, **String** and **Boolean**:

- **Number** represents decimal numbers: e.g., 0, 1, 2.5, -4, 35;
- **String** represents character strings;
- **Boolean** has only two values: **TRUE** or **FALSE**.

A special case of the data type is **Array**, but it is not a basic data type of the pseudo-code. You can see more detailed on Array in part 10.

When defining a variable or a constant, you can also define its data type at the same time. Use keyword "**DEFINE**" or "**CREATE**" to define them; in detailed, the complete syntax for define a **variable** is the following:

```
DEFINE variable [= value] AS data type
CREATE data type variable [= value]
```

Note: "[]" means parts in this symbol could be omitted.

Example

```
DEFINE x AS NUMBER
DEFINE x = 5 AS NUMBER
DEFINE y = "Liverpool" AS STRING
CREATE NUMBER x,y                                                          (1)
CREATE NUMBER x = 5
```

Note 1: you must define a data type for defining a variable, but value is alternative.

Note 2: if you do not define a value but define a data type for the variable, the program will give them a default value according to its data type. The following is the default value of different data types:

Data Types	Default Values
Number	0
String	"" (a null character)
Boolean	false

Note 3: you must add a double quote("") when defining or representing a value of string.

Note 4: pay attention on the statement (1), you can define two or more variables at same time; but note that define each name of variables that split by a comma (,), and **DO NOT contain any space** here.

Also, the complete syntax for define a **constant** is:

DEFINE A CONSTANT const = value **AS data type**

CREATE A CONSTANT data type const = value

Example

```
DEFINE A CONSTANT y = "Liverpool" AS STRING
CREATE A CONSTANT NUMBER x = 5
```

Note 1: you MUST define a value to a constant, and also the data type.

Note 2: once a constant is defined, its value is no longer allowed to be changed; that is, if you try to change the value of the constant, the program will report error.

The pseudo-code has defined several common constants as following, you can use them directly:

Constants Names	Values
PI	3.14159265359…… (pi)
EULE	2.71828182846…… (Euler's number)
MAX_8BIT, MIN_8BIT	±2^(8 – 1) – 1
MAX_16BIT, MIN_16BIT	±2^(16-1) – 1
MAX_32BIT, MIN_32BIT	±2^(32-1) – 1

Note: as shown above, you cannot define a number that bigger than **MAX_32BIT** or smaller than **MIN_32BIT**, it may make the translated code run failed.

In addition, for Boolean variables, note that the value of a Boolean only has **TRUE** or **FALSE**.

Finally, the name of the variables and constants also has some rules on them:

- It should start with an English alphabet (a-z, A-Z), and should only contains numbers (0-9), English alphabets and underline (_);
- It should not be a **keyword** of this pseudo-code;
- A name of variables or constants can be defined at most once.

The following table shows all keywords of the pseudo-code standard

PRINT	**CREATE**	**MAX_32BIT**	**OR**	**NULL**	**WHENEVER**	**SWAP**	**ARRAYINDEX**
OUTPUT	**AS**	**MIN_32BIT**	**NOT**	**CASE**	**WHEN**	**TEMPORARY**	**ARRAYROWINDEX**
STRING	**PI**	**PLUS**	**ASSIGN**	**OTHER**	**FUNCTION**	**TEMP_SIZE**	**ARRAYCOLUMNINDEX**
NUMBER	**ELUE**	**MINUS**	**IF**	**OF**	**RETURN**	**APPEND**	**ARRAYSORT**
BOOLEAN	**MAX_8BIT**	**TIMES**	**THEN**	**REPEAT**	**EXECUTE**	**UPPERCASE**	**SUBARRAYOF**
TRUE	**MIN_8BIT**	**DIVS**	**END**	**UNTIL**	**ADD**	**LOWERCASE**	
FALSE	**MAX_16BIT**	**MOD**	**ELSE**	**FOR**	**LIST**	**COMMENT**	
DEFINE	**MIN_16BIT**	**AND**	**DO**	**EACH**	**ELEMENT**	**STRINGINDEX**	

Note that the pseudo-code is case insensitive. Thus, both "PRINT" or "print" cannot be a variable name; and e.g., if you have defined a variable called "city", then you cannot define a variable called "City" or "cITY", etc.

Last but not least, you cannot define a variable or constant in a sub-structure, that is, conditional statements and loop structure (you can see more details about both structures in part 7 and 8).

3. Output values

Use keyword "**OUTPUT**" could print out current value of a defined variable or a constant. Please note that you must define the variables or constants first before output them.

Example

```
DEFINE x = 5 AS NUMBER
OUTPUT x
DEFINE y = "Liverpool" AS STRING
OUTPUT x,y                                                                    (1)
```

Output Result

```
x = 5
x = 5, y = "Liverpool"
```

Note: the words after "OUTPUT" **only can contains the defined variables or constants**. If you want to output more than one variable or constant at the same time, you should write the statement as the above statement (1), split each variable or constant name by a comma (","), and **DO NOT contain any space** here.

4. Operators

In the pseudo-code, it needs several operators to operate the values of variables. We can divide the operators into the following according to their uses:

- Arithmetic operators;
- Relational operators;
- Logical Operators.

(Note: in fact, "assignment" is also a type of operator. However, in this document, we describe assignment as an individual part in part 5. Also, **Array** and **String** has several their own operators, please see part 10 and 11.)

And the following 3 tables are the details of their uses.

Arithmetic Operators	Symbol and keywords	Examples
Addition	**PLUS, +**	**DEFINE** x = 1 **PLUS** 3
Subtraction	**MINUS, -**	num2 = num1 - 5
Multiplication	**TIMES, ***	num2 = 17 **TIMES** 4
Division	**DIVS, /**	num2 = num1 **DIVS PI**
Remainder	**MOD, %**	a = b **MOD** 5
Divisible	**IS DIVISIBLE BY**	**CREATE BOOLEAN** div = x **IS DIVISIBLE BY** 5
Self-increment	**INCREMENT BY**	x **INCREMENT BY** 1
Self-decrement	**DECREMENT BY**	y **DECREMENT BY** 1
Self-multiplication	**MULTI_INCREMENT BY**	z **MULTI_INCREMENT BY** i
Self-division	**DIV_DECREMENT BY**	w **DIV_DECREMENT BY** i

Note: Strictly speaking, "Divisible" is not an arithmetic operator, but a judgment statement (condition, see part 6).

Relational Operators	Symbol and keywords	Examples
Equation	`[IS]` **`EQUAL TO, ==`**	**`x == 5`**
Difference	`[IS]` **`NOT EQUAL TO, !=`**	`y` **`NOT EQUAL TO`** `100`
Bigger than	`[IS] [NOT]` **`BIGGER THAN, [<=] >`**	`x` **`IS BIGGER THAN`** `5`
Smaller than	`[IS] [NOT]` **`SMALLER THAN, [>=] <`**	`x` **`<`** `20`
Bigger than or equal to	`[IS] [NOT]` **`BIGGER THAN OR EQUAL TO, [<] >=`**	`num1` **`>=`** `num2`
Smaller than or equal to	`[IS] [NOT]` **`SMALLER THAN OR EQUAL TO, [>] <=`**	`num1` **`<=`** `30`

Note: for these operators, the comparison is based on the "value" of variables or constants; generally speaking, the comparison operation can only have its real effect if it is built on the **Number** data type.

Logical Operators	Symbol and keywords	Expressions in propositional logic
And	**AND**	$P \land Q$
Or	**OR**	$P \lor Q$
Not	**NOT**	$\neg P$

5. Assignments

Assignment is a way to change the value of variables. Use symbol "=" or keyword "**ASSIGN TO**" to assign a value to a defined variable.

When defining a variable, you may have assigned its value, but at that time you can only assign their values by the symbol "=".

Example

```
num = 30
city = "Liverpool"
ASSIGN 7 TO days
ASSIGN TRUE TO hasSuccessor
```

Note that if you try to assign a value to a variable but both has different data types, the program may fail to run.

A value could be assigned to more than one variable by using the symbol "**,**" to split the names of variables, as following:

Example

```
num1,num2 = 30
city1,city2,city3 = "Liverpool"
ASSIGN 7 TO days,July
```

In addition, self-increment, self-decrement, self-multiplication and self-division that mentioned in part 4 is also a type of assignment:

Self-operators	Expressions in assignment
Self-increment by i	`num = num + i`
Self-decrement by i	`num = num – i`
Self-multiplication by i	`num = num * i`
Self-division by i	`num = num / i`

Thus, it is equivalent that you write `num = num + i` and `num` **`INCREMENT BY`** `i` in the pseudo-code.

6. Conditions

Conditions are the judgment criteria for the program, and their values determine the path of each step of program. All conditions have only two values: **TRUE** or **FALSE**.

A condition is composed of variables, constants and operators.

Example

```
num1 == num2
city EQUAL TO "Liverpool"
x BIGGER THAN 100
y IS DIVISIBLE BY 7
hasSuccessor AND x != 20
```

For convenience, there are some conditions that can simplify the writing in this pseudocode.

Conditions	Simplified expressions
`x SMALLER THAN 0`	`x IS NEGATIVE (x IS NOT POSITIVE)`
`x BIGGER THAN 0`	`x IS POSITIVE (x IS NOT NEGATIVE)`
`x [NOT] EQUAL TO 0`	`x IS [NOT] ZERO`
`x [NOT] MOD y == [!=] 0`	`x IS [NOT] DIVISIBLE BY y`
Determine whether x is (not) a Number	`x IS [NOT] NUMBER`
Determine whether x is (not) a String	`x IS [NOT] STRING`
Determine whether x is (not) a Boolean	`x IS [NOT] BOOLEAN`
Determine whether x is (not) a true Boolean	`x IS [NOT] TRUE`
Determine whether x is (not) a false Boolean	`x IS [NOT] FALSE`
Determine whether String x is (not) equal to y	`x IS [NOT] SAME WITH y`

So far, we have only involved the knowledge of basic data types, so it's inconvenient to show all the characteristic conditions here. You can see more styles of conditions in below parts.

Note: you can write "**NOT**" to deny the majority of conditions, see "[]" contents in above table.

7. Conditional statements

There are several conditional statements that being used in the pseudo-code.

- **IF** condition **THEN** … **END IF**

Example

```
DEFINE x = 5 AS NUMBER
IF x IS POSITIVE THEN
    PRINT x is positive
END IF
```

Output Result

```
x is positive
```

Note 1: you MUST add a "**END IF**" at the end of statements.

Note 2: there could have several sentence lines after the keyword "**THEN**", but you must write then in a new line.

Both notes are same to all **IF**-conditional statements.

- **IF** condition **THEN** … **ELSE** … **END IF**

Example

```
DEFINE x = -5 AS NUMBER
IF x IS POSITIVE THEN
    PRINT x is positive
ELSE
    PRINT x is non-positive
END IF
```

Output Result

```
x is non-positive
```

- **IF** condition **THEN** … **ELSE IF** condition **THEN**… [**ELSE IF** condition **THEN**… **ELSE**] … **END IF**

Example

```
DEFINE x = 0 AS NUMBER
IF x IS POSITIVE THEN
    PRINT x is positive
ELSE IF x IS NEGATIVE THEN
    PRINT x is negative
ELSE
    PRINT x is zero
END IF
```

Output Result

```
x is zero
```

Note: every **IF**-condition line must end with a "**THEN**"; "[]" means parts in this symbol could be omitted.

- Nested **IF**

Example

```
DEFINE x = 7 AS NUMBER
IF x IS POSITIVE THEN
    IF x > 5 THEN
        PRINT i love this number
    END IF
    PRINT x is positive
ELSE IF x IS NEGATIVE THEN
    PRINT x is negative
ELSE
    DO NOTHING
END IF
```

Output Result

```
i love this number
x is positive
```

Note that the keyword “**DO NOTHING**”, it is equivalent to write nothing in a line.

- **CASE** expression **OF** label1**:** [label2**:** label3**: OTHERS:**] … **END CASE**

Example

```
DEFINE country = "China" AS STRING
CASE country OF
    "America":
        PRINT USA
    "China":
        PRINT CN
    "Japan":
        PRINT JP
    OTHERS:
        DO NOTHING
END CASE
```

Output Result

```
CN
```

Note that after each label, you can write any sentences lines. But you must start a label in a new line. “**OTHERS**” means the expression is not equal to any label above. And it is NOT necessary to appear “**OTHERS**” in a CASE OF condition. In addition, please do not forget write “**END CASE**” at the end of the CASE OF.

CASE OF statement is similar to the **IF…ELSE IF…ELSE IF…ELSE…** condition**.**

In addition, please note that you **cannot define a new variable in condition structure (in the range of IF...END IF or CASE…END CASE)**.

8. Loop structures

Program statements of a sequential structure can only be executed once. If you want the same operation to be performed multiple times, you need to use a loop structure.

- **REPEAT** … **UNTIL** condition **END REPEAT / PEREAT UNTIL LOOP RUN** number **TIMES** condition **END REPEAT**

Example

```
DEFINE day = 0 AS NUMBER
REPEAT
    day INCREMENT BY 1
UNTIL day EQUAL TO 0
END REPEAT
OUTPUT day
REPEAT UNTIL LOOP RUN 7 TIMES
    day DECREMENT BY 1
END REPEAT
OUTPUT day
```

Output Result

```
day = 0
day = -7
```

Note 1: you MUST add a "**END REPEAT**" at the end of **REPEAT** loop.

Note 2: "**LOOP RUN** number **TIMES**" is a keyword for controlling the times of loop as a condition. The number should be a positive integer. Note that you can only add this condition after **REPEAT UNTIL** statement, not the **REPEAT … UNTIL** statement.

- **REPEAT WHEN** condition … **END REPEAT** / **WHILE WHEN** condition … **END WHILE**

Example

```
DEFINE day = 0 AS NUMBER
REPEAT WHEN day [IS] SMALLER THAN 3
   PRINT day
   day INCREMENT BY 1
END REPEAT
WHILE [WHEN] day [IS] BIGGER THAN 4
   OUTPUT day
   day DECREMENT BY 1
END WHILE
PRINT day + 1
```

Output Result

```
day
day
day
day + 1
```

Note 1: it is alternative for type **WHILE…** or **WHILE WHEN…** both styles are equivalent.

Note 2: you MUST add a "**END WHILE**" at the end of **WHILE** loop.

Note 3: "[]" means parts in this symbol could be omitted.

- **FOR** number **FROM** i **TO** j … **END FOR**

Example

```
DEFINE day = 0 AS NUMBER
FOR i FROM 0 TO 3
   day DECREMENT BY 1
END FOR
OUTPUT day
```

Output Result

```
day = -4
```

Note: it is no need to define the index number first, that is, the number **i** and **j**; **i** and **j** must be two integer numbers, "number" will automatically **increment (i < j) by 1**; if you make i > j, the code may run with error. If i = j, loop will only execute once.

- **WHENEVER … END WHENEVER**

Example

```
DEFINE day = 0 AS NUMBER
WHENEVER
    day INCREMENT BY 1
    OUTPUT day
END WHENEVER
```

Output Result

```
day = 1
day = 2
day = 3
day = 4
……
```

Note: this is an endless loop.

- **FOR EACH** loop with special usage

For the following loop keywords, they have particular usages in different situation.

FOR EACH MONTH OF A YEAR … END FOR (execute loop 12 times)

FOR EACH DAY OF A WEEK … END FOR (execute loop 7 times)

Here is all time duration words that can be used:

YEAR	WEEK	HOUR	SECOND
MONTH	DAY	MINUTE	

For each also can be used as:

FOR EACH ELEMENT OF [ARRAY] arrayName … **END FOR** (execute loop arrayName**.LENGTH** times, more details in part 10)

FOR EACH CHARACTER OF [STRING] stringName … **END FOR** (execute loop stringName**.LENGTH** times, more details in part 11)

……

- Loop can be **nested (Nested WHILE, nested REPEAT, nested FOR)**:

Example

```
WHILE WHEN day SMALLER THAN 100
    WHILE WHEN month SMALLER THAN 12
        day DECREMENT BY 1
    END WHILE
END WHILE
```

Nested loop will finish the deepest loop first… and then the outer loops.

In addition, there are two important keywords that being used in the loop structures: "**STOP LOOP**" and "**GO NEXT LOOP**".

"**STOP LOOP**" is mainly used in loop statements to jump out of the entire block. "STOP LOOP" jumps out of the innermost loop and continues to execute the statements below the loop.

"**GO NEXT LOOP**" applies to any loop control structure. The effect is to make the program jump immediately to the next iteration of the loop; it is not used very frequent, though.

Example

```
DEFINE day = 0 AS NUMBER
DEFINE multi = 2 AS NUMBER
REPEAT WHEN day SMALLER THAN 3
    multi INCREMENT BY 2
    day INCREMENT BY 1
    IF multi EQUAL TO 4 THEN
        STOP LOOP
    END IF
END REPEAT
OUTPUT day
```

Output Result

```
day = 1
```

Note that you must write "**STOP LOOP**" or "**GO NEXT LOOP**" in a new line and the line only contains these contents.

9. Define and use functions, return value and parameters

A "function" is a collection of statements that perform a function together. Use keyword "**DEFINE FUNCTION**" to start and define a function, in detail:

DEFINE FUNCTION name [**WITH PARAMETER** parameter1 **AS data type** [, parameter2 **AS data type**, …]]

… **END DEFINE** [**AND RETURN data type**]

Note**:** "[]" means parts in this symbol could be omitted.

Name is the name of the function; the rules of name are similar to that of variables.

Parameters looks like a placeholder. When the function is called, a value is passed to the parameter. This value is called an argument or variable. Parameters are optional and the function can contain no parameters; you must define the data type of parameters when defining function.

Functions may have **return values**. You should define the data type of return value when defining the function if the function has return value.

Note: after Pseudocode Translator version 3.0, the program has added a new way to add function (that is, click "+ add function" button and edit the contents of it in a new window). Thus, it is not support for user to input function in main interface.

The following is one example of function definition.

Example

```
DEFINE FUNCTION play WITH PARAMETER game AS STRING
   IF game == "card" THEN
      PRINT i am happy
   ELSE
      PRINT i am angry
   END IF
END DEFINE
```

Note that this function has no return values but has a parameter.

Example

```
DEFINE FUNCTION play WITH PARAMETER game AS STRING
   IF game == "card" THEN
      RETURN 0
   ELSE
      RETURN 1
   END IF
END DEFINE AND RETURN NUMBER
```

Note that this function has return value, and it MUST contain "**RETURN**" keywords in the function as an exit port of it.

When you want to use a defined function, the keyword "**EXECUTE**" will be used, in detail:

EXECUTE name**()**

EXECUTE name**(**value of parameter1, value of parameter2, …**)**

If you want to assign the return value to a variable, you should use the keyword **EXECUTE…AS…** to finish the assignment.

EXECUTE name**() AS** variable

EXECUTE name**(**value of parameter1, value of parameter2, …**) AS** variable

Example

```
EXECUTE play()
EXECUTE play(card, 3)
EXECUTE play(card, 3) AS x
```

Note 1: you must type "()" after the name of the function to represent that it is a function, even it has no parameters.

Note 2: when calling the function, you need to complete all the required parameters, the number of parameters must be consistent with that of its definition.

Note 3: you do not need to define the variable (which get the return value of function) before use **EXECUTE…AS…**, such the statement contains the definition of this variable. In addition, you cannot use **EXECUTE…AS…** on a non-return value function.

10. Array operations

Array is an important data structure. An array could store several variables or constants with same data types.

You can learn some basic concepts of array here: https://en.wikipedia.org/wiki/Array_data_structure.

In the pseudo-code, you can use "**CREATE ARRAY**" (recommend this) or "**DEFINE ARRAY**" keywords to start and define an array.

CREATE ARRAY data type name[1..space]

DEFINE ARRAY name[1..space] **AS data type**

CREATE ARRAY data type name = [element1, element2, element3, …]

DEFINE ARRAY name = [element1, element2, element3, …] **AS data type**

Space is the total space of an array. When defining an array, you must define its data type, space amount at the same time, or just list all elements of it.

You can use "**ADD INTO**" or "**LIST**" keywords to add elements after just defining the space of array.

Example 1

```
CREATE ARRAY STRING cities[1..3]
ADD "Liverpool" INTO cities
ADD "London" INTO cities
ADD "Edinburgh" INTO cities
```

Example 2

```
CREATE ARRAY STRING cities[1..3]
LIST cities = [Liverpool, London, Edinburgh]
```

If you want to use a value of array, you should mention arrayName[0], arrayName[1]… which represent the first, second, … element of array. You only can add the element with same data type.

Please pay attention on **the beginning index of an array is 0**.

cities[]	Liverpool	London	Edinburgh	Manchester	……

cites[0] = "Liverpool";

cites[3] = "Manchester"…

Note that you can omit double quotes ("") of string when writing the list elements of the string array.

Example

```
CREATE ARRAY STRING cities = [Liverpool, London, Edinburgh]
DEFINE x = cities[2] AS STRING
OUTPUT x
```

Output Result

```
cities[2] = "Edinburgh"
```

Note that here the OUTPUT statement cannot output cities[2] directly because that OUTPUT only can "output a defined variable or constant”.

However, **arrayName[1]**, **arrayName[2]**, … are similar to variables; they could be changed values, assigned and give operators to do some operation. For example, you can write `cities[1] = London` to finish the add element directly.

The following are some defined operations in the pseudo-code on array:

Operations on Array “arr”	Functionalities
FOR EACH ELEMENT OF ARRAY arr … **END FOR**	A loop that traversing all elements in array (i.e. traversing loop), it means a loop run arr.LENGTH times.
arr**[E]**, arr**[ELEMENT]**	An element (you must define first) of the array
arr**[E - 1]**, arr**[ELEMENT - 1]**	Direct precursor of current element of the array
arr**[E + 1]**, arr**[ELEMENT + 1]**	Direct successor of current element of the array
PRINT ALL ELEMENT OF [ARRAY] arr	Print all elements of array in sequence; must write it in a new line
SWAP arr**[i]**, arr**[j]**	Swap the value of arr[i] and arr[j]; must write it in a new line
SORT ARRAY arr	Sort all elements of array from small to big; Number is based on the numbers, String is based on the initials of strings, Boolean cannot be sorted.
arr**.LENGTH**	The value of the length of array
arr.[i..j]	The value of subarray from index i to j

Arrays are not only one-dimensional. For example, you can create a two-dimensional array:

CREATE ARRAY data type name[1..rowSpace][1..columnSpace]

DEFINE ARRAY name[1..rowSpace][1..columnSpace] **AS data type**

CREATE ARRAY data type name = [[element1, element2, …], [element3, element4, …], …]

DEFINE ARRAY name = [[element1, element2, …], [element3, element4, …], …] **AS data type**

Two-dimensional arrays are also support the keyword “**ADD TO**” and “**LIST**”.

Example

```
CREATE ARRAY STRING cities[1..3][1..3]
LIST cities = [[Liverpool, London, Edinburgh], [Beijing, Shanghai, Hongkong], [Tokyo, Osaka, Sapporo]]
```

This example creates an array like this:

Liverpool	London	Edinburgh
Beijing	Shanghai	Hongkong
Tokyo	Osaka	Sapporo

Thus, cities[0][0] is equal to “Liverpool” and cities[1][2] is equal to “Hongkong”, and so on.

Two-dimensional array can also use the operation:

- "**FOR EACH**" to transferring (the sequence is **row by row**, i.e. for the above example, the sequence is Liverpool-London-Edinburgh-Beijing-Shanghai-Hongkong-Tokyo-Osaka-Sapporo), and
- "**PRINT ALL ELEMENT OF ARRAY**" to print all elements, and
- "**SWAP**" to swap every to elements' values.

However, "**SORT**", and "**.LENGTH**" is NOT available for two-dimensional array; in fact, such three operation are just fit the one-dimensional array.

In addition, array can be a parameter or a return value of functions.

Example

```
DEFINE FUNCTION play WITH PARAMETER game AS STRING ARRAY
   CREATE ARRAY NUMBER happyOfPlayer = [0, 0, 0]
   IF game CONTAINS "card" THEN
      ASSIGN 1 TO happyOfPlayer[2]
   ELSE
      ASSIGN 1 TO happyOfPlayer[1]
   END IF
   RETURN happyOfPlayer
END DEFINE AND RETURN NUMBER ARRAY
```

The keyword of the data type is **NUMBER ARRAY**, **STRING ARRAY** and **BOOLEAN ARRAY**.

*When translating, the C language may use the pointer to replace the array; as C cannot make an array become a parameter.

11. String operations

We have introduced how to define a string in part 2. There are several operations in the pseudo-code that can handle the value or properties of strings. But please note that you should define string first before do these operations.

Operations on String "str"	Functionalities

`FOR EACH CHARACTER OF STRING str`
 …
 `END FOR`
	A loop that traversing all character in String (i.e. traversing loop), it means a loop run str.LENGTH times.

`str[C], str[CHARACTER]`	A character (you must define first) of the string
`str[C - 1], str[CHARACTER - 1]`	Direct precursor of current element of the string
`str[C + 1], str[CHARACTER + 1]`	Direct successor of current element of the string
`str.LENGTH`	The value of the length of string
`str [IS] [NOT] SAME WITH str2`	Determine that whether the content of str is (not) equal to that of str2(*)
`APPEND str2 [, str3, str4…] TO str`	Connect strings, append str2, (str3, str4 …) to str, append at the end of str; must write it in a new line

`str[i]`	The character at the index = i of string
`str.INDEX[character]`	The index of the character first appears in string.
`str.UPPERCASE`	All character of str with uppercase
`str.LOWERCASE`	All character of str with lowercase
`str.[i..j]`	The value of substring from index i to j (not include index j)

(*) Note: if you use "equal to" here, the syntax is correct but the running result may make you disappointed. You can see more information on this link: https://www.javatpoint.com/java-string-equals

Note that the index of string is similar to the expressions in array, starts at index=1.

For example, if `str = "Liverpool"`, thus, `str[0] = "L"` and `str[2..5] = "ver"`.

12. Common functions for Numbers and instructions for program

Here are several Math functions for handling Numbers that being used in pseudo-code, you can use them as a value of variable directly:

Operations	Functionalities
`ABS(x)`	Return the absolute value of x; x must be a Number
`MAX(a, b)`	Return the maximum number of a and b; a, b must be Numbers
`MIN(a, b)`	Return the minimum number of a and b; a, b must be Numbers
`CEILING(x)`	Returns the minimum integer greater than or equal to x
`FLOOR(x)`	Returns the maximum integer smaller than or equal to x
`ROUNDING(x)`	Return the rounding value of x
`POWER(x, n)`	Return x^n
`LOG(x, n)`	Return $\log_n x$
`LG(x)`	Return lg x
`EXP(x)`	Return $\log_e x$
`ROOT(x)`	Return the square root of x
`SIN(x)`	Return sin(x), x is the degree
`COS(x)`	Return cos(x), x is the degree
`TAN(x)`	Return tan(x), x is the degree
`ARCSIN(x)`	Return arcsin(x), x is the value [-1, 1]
`ARCCOS(x)`	Return arccos(x), x is the value [-1, 1]
`ARCTAN(x)`	Return arctan(x), x is the value $(-\infty, +\infty)$

*When translating to C, some of this function may be disable due to C has no such Math functions.

Example

```
DEFINE num = 4.5 AS NUMBER
DEFINE x = CEILING(num) AS NUMBER
OUTPUT x
DEFINE num2 = num + ABS(num) AS NUMBER
ASSIGN ROOT(num2) TO num2
OUTPUT num2
num2 = num PLUS 0.6
num2 = ROUNDING(num2)
OUTPUT num2
```

Output Result

```
x = 5.0
number2 = 3.0
number2 = 5.0
```

In addition, these are several instructions for program that could be used in the pseudo-code. You can write them in a new line and occupies the whole line, to instruct the program to do something.

Instructions	Functionalities
EXIT PROGRAM	Stop anything and exit the program at once
DO NOTHING	Equivalent to a null line
DO xxx	xxx could be any words, even plain English. The system would try to knowledge xxx is a function with no parameter, though you have not defined it yet. When translating, this line will become a comment (see part 13).

13. Comments

Use the symbol "**//**" or the keyword "**COMMENT**" or "**COMMENT:**" as the beginning of a line, you can add a comment line into the program (please mind the space).

The contents of comments could be anything, and it will not affect the execution of program; comments will be ignored by the program when running.

Please note that a comment requires a whole line space and it cannot be written after a normal sentence. For example:

```
// This is a correct comment.
PRINT I am just a cute output
```

```
PRINT I am just a cute output // This is NOT a correct comment.
```

When translating, the comments will be ignored by the program; in other words, comments will not be appeared in the translated sentences.

Comments can make it easier for you to record your program information, whatever you are a noob programmer or even a pro programmer; please form the habit of writing comments in your program.

Example

```
COMMENT A small program to count the number of characters in the word.
DEFINE word = "congratulations" AS STRING
DEFINE num AS NUMBER
ASSIGN word.LENGTH TO num
//if the length of word is equal to 0, print some information.
IF num EQUAL TO 0 THEN
    PRINT does there have a word?
ELSE
    COMMENT: output the number's value.
    OUTPUT num
END IF
```

Output Result

```
number = 15.0
```

14. Case study: Write Binary Search in the pseudo-code style

Binary search is a search algorithm that finds the position of a target value within a **sorted** array.

Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.

Example

```
DEFINE FUNCTION binarySearch WITH PARAMETER arr AS NUMBER ARRAY, aim AS NUMBER
    IF arr.LENGTH IS ZERO THEN
        RETURN -1
        COMMENT: -1 means not found the aim.
    END IF
    DEFINE ends = arr.LENGTH AS NUMBER
    DEFINE mid = FLOOR(ends DIVS 2) AS NUMBER
    IF arr[mid] > aim THEN
        RETURN EXECUTE binarySearch(arr.[0..mid - 1], aim)
    END IF
    IF arr[mid] < aim THEN
        RETURN EXECUTE binarySearch(arr.[mid + 1..ends], aim)
    END IF
    RETURN mid
END DEFINE AND RETURN NUMBER
```

Note that the step **RETURN EXECUTE** `function` used a concept called "recursion", where a function being defined is applied within its own definition.

To see more details about binary search, please read: https://en.wikipedia.org/wiki/Binary_search_algorithm.

15. Case study: Write the pseudo-code to find Greatest common divisor and Least common multiple

In mathematics, the greatest common divisor **(gcd)** of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4; the least common multiple **(lcm)** is the smallest positive integer that is divisible by both two integers.

The relationship between the **gcd** and **lcm** of two integers a, b is as follows:

$$\operatorname{lcm}(a,b) = \frac{|a \cdot b|}{\gcd(a,b)}$$

And you can use Euclidean algorithm to find out the **gcd** of two integers: Divide two numbers, take the remainder and repeat the division until the remainder is 0, the previous divisor is **gcd**.

When you know the gcd of two integers, you can calculate the lcm of them by the above formula.

Example

```
DEFINE FUNCTION gcd WITH PARAMETER a AS NUMBER, b AS NUMBER
    IF b EQUAL TO 0 THEN
        RETURN a
    ELSE
        RETURN EXECUTE gcd(b, a MOD b)
    END IF
END DEFINE AND RETURN NUMBER

DEFINE FUNCTION lcm WITH PARAMETER a AS NUMBER, b AS NUMBER
    DEFINE mul = a TIMES b AS NUMBER
    EXECUTE gcd(a, b) AS num
    DEFINE outputs = mul DIVS num AS NUMBER
    RETURN outputs
END DEFINE AND RETURN NUMBER
```

To see more information about gcd and lcm, please read: https://en.wikipedia.org/wiki/Greatest_common_divisor and https://en.wikipedia.org/wiki/Least_common_multiple.

16. Case study: Write Bubble Sort in the pseudo-code style

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Step by step example (from Wikipedia):

Take an array of numbers " 5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required;

First pass:

(**5 1** 4 2 8) → (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) → (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) → (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) → (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second pass:

(**1 4** 2 5 8) → (**1 4** 2 5 8)
(1 **4 2** 5 8) → (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third pass:

(**1 2** 4 5 8) → (**1 2** 4 5 8)
(1 **2 4** 5 8) → (1 **2 4** 5 8)
(1 2 **4 5** 8) → (1 2 **4 5** 8)
(1 2 4 **5 8**) → (1 2 4 **5 8**)

Write the bubble sort in the style of the pseudo-code:

Example

```
DEFINE FUNCTION bubbleSort WITH PARAMETER arr AS NUMBER ARRAY
    FOR i FROM 1 TO arr.LENGTH - 1
        FOR j FROM 0 TO arr.LENGTH - 1 - i
            IF arr[j] > arr[j + 1] THEN
                SWAP arr[j], arr[j + 1]
            END IF
        END FOR
    END FOR
    RETURN arr
END DEFINE AND RETURN NUMBER ARRAY
```

Sort is very important in algorithm. Although you can use "**SORT ARRAY**" keyword to sort an array directly in pseudo-code, it is useful to acknowledge several sort algorithms and try to implement them.

END OF DOCUMENT