Statements

Conditional statements

If statement

Conditionally executes another statement.
Used where code needs to be executed based on a condition.

Syntax :

```
if condition then statement-true end if
if condition then statement-true else statement-false end if
```

If condition yields true, statement-true is executed.

Example :

EZ	C++

variable Var is integer
Var = 5
If Var > 3 then
Print Var
end if
	Int var = 5;
	if(var > 3){
	Std::cout << var << std::endl;
	}

Output : 5

If the else part of the if-statement is present and condition yields false, statement-false is executed.

Example :

EZ	C++

variable var is integer
var = 5
If var > 3 then
print “variable greater than 3”
else
print “variable less than 3”
end if
	Int var = 5;
	if(var > 3){
	Std::cout << “variable greater than 3” << std::endl;
	}else{
	Std::cout << “variable less than 3” << std::endl;
	}

Output : variable greater than 3

Nested if-statements can also be used in order to test more than two cases.

Example :

EZ	C++

variable var is integer
var = 5
If var > 3 then
print “variable supérieur à 3”
else
If var< 0 then
print “variable négative”
else
print “variable positive inférieur à 3”
end if
end if
	Int var = 5;
	if(var > 3){
	Std::cout << “variable supérieur à 3” << std::endl;
	}else if(var < 0){
	Std::cout << “variable négative” << std::endl;
	}else{
	Std::cout << “variable positive inférieur à 3” << std::endl;
	}

when statement

Transfers control to one of the several statements, depending on the value of an expression.

Syntax :

```
when  expression is
      case constant_expr1
            bloc1
      end case
      case constant_expr2
            bloc2
      end case
      default
            blocdefault
      end case
end when
```

The body of a when statement may have an arbitrary number of case labels, as long as the values of all constant_expressions are unique. At most one default case may be present.

If expression valuates to the value that is equal to the value of one of constant_expressions, then control is transferred to the statement that is labeled with that constant_expression.
If expression evaluates to the value that doesn't match any of the case labels, and the default label is present, control is transferred to the statement labeled with the default label.

Example :

EZ	C++

variable x, y are integer
x = 0
 	
when y is
    case 1
        x = x +1
    end case
    case 2
        x = x +2
   end case
   default
      x = x + 3
   end case
end when
	Int x = 0, y;
	switch(y) {
	    case 1 : x = x +1;
	    break;
	    case 2 : x = x +2;
	    break;
	    case default : x = x + 3;
	    break;
	}

Output : x = 3

Loops

while loop

Executes a statement repeatedly, until the value of the condition becomes false. The test takes place before each iteration. The condition is an expression that must be convertible to a boolean value. If it yields to false at the beginning of a new iteration, the loop is exited.

Syntax : `while condition do statement(s) end while`

Example :

EZ	C++

variable x is integer
x = 0
while x < 10 do
    x = x +1
end while
	Int x = 0;
	
	while(x < 10) {
	    x = x +1;
	}

Output : x = 10

repeat… until loop

Executes a statement repeatedly, until the value of the condition becomes true. The test takes place after each iteration. The condition is an expression that must be convertible to a boolean value. If it yields to true at the end of an iteration, the loop is exited.

Syntax : `repeat statement(s) until condition`

Example :

EZ	C++

`variable x is integer`
`x = 0`
 	
`repeat`
`    x = x +1`
`until x < 10`
	`int x = 0;`
	
	`do {`
	`  x = x +1;`
	`} while(x < 10)`

Output : x = 10

for loop

Executes a statement repeatedly over a range of values, i.e as long as the variable declared in range_declaration is in the range specified by range_expression. Otherwise, the loop is exited. The variable declared in range_declaration must be of the same type as the element of the sequence represented by range_expression. The incrementation of the loop counter (typically, the range_declaration) may be specified by the iteration_expression introduced by the keyword step. If no iteration_expression is specified, the loop counter will be incremented by 1. The incrementation is done at the end of each iteration, before checking if the range is respected.

Syntax :

```
for range_declaration in range_expression do statement(s) end for

for range_declaration in range_expression step
iteration_expression do statement(s) end for
```

EZ	C++

`variable x, y, k are integer`
`x = k = 0`
`y = 10`
`for a is integer in x..y do`
	`int x = 0, y = 10, k = 0;`
	`for(int a = x; a = y; a++){`
	`    k = k + 1`
	`}`

`    k = k + 1` `end for`	

foreach loop

Executes a statement repeatedly over all the elements of a container.

Syntax : `foreach range_declaration in container do statement(s) end each`

Example :

EZ	C++

`v is vector of integer`
`foreach e is integer in v do`
`    e = 0`
`end each`
	`vector<int> v;`
	`for(auto e : v) {`
	`    e = 0;`
	`}`

Functions and procedures

Functions and procedures are entities that associate a sequence of statements (a *function / procedure body*) with a *name* and a list of zero or more *function/procedure parameters*. A function terminate by returning a result, whereas a procedure do not return anything.

Syntax

EZ	C++
Function syntax	

`function function_name (arg is arg_type) return return_type`
`return_type variable`
`    statement(s)`
`return variable`
`end function`
	`return_type function_name (arg_type arg) {`
	`return_type variable;`
	`    statement(s)`
	`return variable;`
	`}`

Procedure syntax	

`procedure procedure_name (arg is arg_type)`
`    statement(s)`
`end procedure`
	`void procedure_name (arg_type arg) {`
	`    statement(s)`
	`}`

A default value can be provided for function or procedure arguments. This value will be attributed to the argument if none is provided when the function or procedure is invoked.

EZ	C++
Function syntax	

`function function_name (arg is arg_type = value) return return_type`
`return_type variable`
`    statement(s)`
`return variable`
`end function`
	`return_type function_name (arg_type arg = value ) {`
	`return_type variable;`
	`    statement(s)`
	`return variable;`
	`}`

Procedure syntax	

`procedure procedure_name (arg is arg_type = value)`
`    statement(s)`
`end procedure`
	`void procedure_name (arg_type arg = value) {`
	`    statement(s)`
	`}`

When a function or a procedure is invoked, the parameters are initialized from the arguments (either provided at the place of call or defaulted) and the statements in the function body are executed.

EZ	C++
Function syntax	
`variable arg is arg_type` `variable var is var_type = function_name(arg);`	`arg_type arg;` `var_type var = function_name(arg);`
Procedure syntax	
`variable arg is arg_type` `procedure_name(arg);`	`arg_type arg;` `procedure_name(arg);`

Overload resolution

Several functions / procedures may have the same name as long as they have a different signature.

In order to compile a function / procedure call, the compiler must first perform name lookup, which, for functions and procedures, may involve argument-dependent lookup. If these steps produce more than one *candidate function*, then *overload resolution* is performed to select the function / procedure that will actually be called.
In general, the candidate function whose parameters match the arguments most closely is the one that is called.

EZ	C++
Declaration	

`procedure procedure_name (arg is type1)`
`    statement(s)`
`end procedure`
 	
`procedure procedure_name (arg is type2)`
`    statement(s)`
`end procedure`
	`void procedure_name(type1 arg){`
	`    statement(s)`
	`}`
	
	`void procedure_name(type2 arg){`
	`    statement(s)`
	`}`

Call	

`variable A is type1`
`variable B is type2`
`//type1 different from type2`
 	
`procedure_name(A)`
`procedure_name(B)`
	`type1 A;`
	`type2 B;`
	`//type1 different from type2`
	
	`procedure_name(A)`
	`procedure_name(B)`

Input and output streams

The print instruction can be used to display information on the output stream. Plain text can be concatenated with the “.” operator to variables whose type in fundamental (integer, real, string…)

Example :
```
n is integer = 3
print “I have “ . n . “ apples”
```

Output : I have 3 apples

The read instruction can be used to get a keyboard entry and store it into a variable. The program stays on hold while waiting for this entry.

Example :

```
variable saisie is string
read saisie
```