

# ComIT

## ALGORITHMS

# THE TOPICS

- Purpose of programming
- Actions, interlocator and Algorithms
- Writing Algorithms
- Data and Information
- Pseudo Code
  - ✓ Structure of an algorithm
  - ✓ Types of Actions
  - ✓ Constants, Variables, Assignment and Expressions.
  - ✓ Logical Conditions
- Resolution of a first problem

# PURPOSE OF PROGRAMMING

- What is the purpose?
  - ✓ The resolution of computational type problems
- When is a problem computational and when not?
- When a computer can do the work to solve the problem.
  - ✓ It depends on the context

# SOLVING A REAL LIFE PROBLEM

- We get a flat tire
- We have to change the tire
- How do we do it?
  - ✓ We follow a series of steps or instructions.
  - ✓ If my hands are tied, how do I explain another person how to do it?.
  - ✓ What detail level do I need to use?

# SOLVING A REAL LIFE PROBLEM

For some people, solution will be:

1. Change the tire.

Others, will understand this:

1. Raise the car
2. Take out the flat tire
3. Bring the spare
4. Install the spare
5. Put the car back to the ground
6. Put the flat tire in the trunk

# SOLVING A REAL LIFE PROBLEM

But if I'm talking to someone without any expertise (only good will), maybe I'll have to be more specific:

For action 1:

1. Locate the Jack under the car
2. Bring the Jack
3. Raise the car with the Jack

For action 4:

1. Lift the spare tire and put into position
2. Tighten the lug nuts

# DEFINITIONS (I)

- Interlocutor is any entity (person or object) that can understand a stated method (set of actions) and carry out (Execute it).
- Action is a description of something that has to be done and changes the environment.
- Not every action can be executed by any interlocutor.
  - ✓ Primitive Actions
  - ✓ Non-Primitive Actions

# DEFINITIONS (II)

- **Algorithm:** It is an ordered sequence of primitive actions oriented to solve a problem or a certain question.
- **Language:** It is the one that we talk with the interlocutor to describe the algorithms.



# DEFINITIONS (III)

Writing algorithms consists of narrating a set of actions in a certain language to be performed by an interlocutor.

# WRITING ALGORITHMS

Let's make an algorithm to change a burned light bulb:

1. Set the ladder on place
2. Take out the light bulb
3. Install new light bulb
4. Store the ladder

# WRITING ALGORITHMS

And if I need more details?

1. Set the ladder on place
2. Climb up ladder
3. Take out light bulb
4. Step down from the ladder
5. Grab new light bulb
6. Climb up ladder
7. Set up new light bulb
8. Step down from the ladder
9. Store ladder

# WRITING ALGORITHMS

Let's think about how we think...

1. Set the ladder on place
2. Take out the light bulb
  1. Climb up ladder
  2. Take out light bulb
  3. Step down from the ladder
3. Install new light bulb
  1. Grab new light bulb
  2. Climb up ladder
  3. Set up new light bulb
  4. Step down from the ladder
4. Store the ladder

# WRITING ALGORITHMS

- First we think of it in the most general terms possible.
  - ✓ With a lot of abstraction
- Then we go on to break down each "big" step into a series of "little" steps
- Each large step is known by the term **module**
  - ✓ Each one fulfills a well defined function
  - ✓ Each of them, sometimes, can be done independently of the other

# WRITING ALGORITHMS

- This is part of modular thinking known as top-down programming philosophy
  - ✓ It's really all we have at hand to solve a problem when we do not know the details
    - ✓ It is not convenient to start with the details, because we can get lost!!!
  - ✓ We start from the most general information because it is easier for us.
    - ✓ And we go from the most general to the most specific
- It looks easy but it is actually hard
  - ✓ It requires a lot of practice
  - ✓ Creativity

# WRITING ALGORITHMS

- Just to think, **in our previous exercise**, how about electricity?
  - ✓ If I remove the light bulb and the lights are on, it might be the last algorithm I write in my whole life.
  - ✓ Knowing this:
    - ✓ I **assume** things (that lights are off)
    - ✓ Or I give explicit instructions to turn off the lights before removing the light bulb.
- Assumptions are facts that are not written but are part of my solution to the problem.
  - ✓ They help to define the scope of what we are doing and what not.

# EXERCISING

---

## **PRACTICE 1**




# QUESTION

---

What is data?

What is information?

# DIFFERENCES

- 43440436
-  43440436
- John's phone is 43330456

# DATA AND INFORMATION

- A data is a symbolic representation of a feature or property of an entity
  - ✓ It doesn't make sense by itself.
  - ✓ If we process or interpret it, then we have information
    - ✓ We associate data with something or someone.  
(Contextualize)
  - ✓ In computing, data is represented by a value.
    - ✓ That is, in essence, that data is a value about something.

# BASIC DEFINITION OF PROGRAM

---

A program contains a set of algorithms for processing and storing data.

# ALGORITHM = INSTRUCTIONS + DATA

- **Data:** a value (about something).
- **Instruction:** It is a single operation that can be performed by the machine.
- Instructions manipulate data.
- Instructions are executed in the order in which they are written, one after another, in sequence.
- The order in which they are executed is commonly called execution flow.

# STRUCTURE OF AN ALGORITHM

An algorithm has two large blocks:

```
algorithm <name>  
    Data Declaration  
    Actions  
end algorithm
```

# DATA DECLARATION

- Algorithms process data.
- Where do we store the data?
- These data are stored in elements that, according to their mutability criteria, are called:
  - ✓ **Constants:** its value can't change.
  - ✓ **Variables:** its value can change.

# VARIABLES AND CONSTANTS

Identifiable data elements having three properties:

- **A name** that identify them

```
var Number: x
```

```
var Number: total, discount
```

```
const Number: PI = 3,1416 (we have to  
define the value to the constants)
```

- **A type** that describes the use
- **A content:** the value they store



# VARIABLES AND CONSTANTS

- We declare them at the beginning of an algorithm

```
algorithm a
```

```
var Number : x
```

```
var Number : y
```

```
...
```

```
More declarations
```

```
Actions
```

```
end algorithm
```

- We can declare two or more in the same line

```
var Number: x, y;
```

# TYPES OF ACTIONS

- In the computer we do not have at our disposal all types of actions as we were doing describing the algorithms in natural language.
- We only have three basic types of actions and all the problems we want to solve must be described in terms of these.
- The Bohm-Jacopini theorem proves that any algorithm can be described with the three types of actions below.

# TYPES OF ACTIONS

- Sequence: It is a simple operation that runs as a step.
- Selection / Conditional: to make decisions.
- Iteration: They allow to repeat actions under a certain condition or a certain number of times.

# SEQUENCE

- They represent a direct operation to be carried out. They are called a sequence because they are executed one after another in a sequential order.
  - ✓ **We write one action per line**
- Different types:
  - ✓ Assignment
    - ✓ Operation to give value to a variable from an expression.
  - ✓ Input
    - ✓ Operation to enter characters by keyboard
  - ✓ Output
    - ✓ Operation to display characters on the screen
  - ✓ Algorithm invocation
    - ✓ An algorithm is invoked from another algorithm that runs as if it were an action.

# ASSIGNMENT

Operation to give value to a variable from an expression.

$$\langle \text{var} \rangle \leftarrow \langle \text{expr} \rangle$$

First, the expression on the right is resolved, and the result is assigned to the variable on the left.

$$X = 2 + 2$$
$$\text{my-variable} = 7 * 9$$

# EXPRESSIONS

- **Expression:** Combination of operands connected to operators that is evaluated and returns a single value.
- They are representations of a calculation needed to obtain a result that we want.

# EXPRESSIONS

EXPRESSION

X

variable

=

operator

2

operand

+

operator

5

operand

# OPERATORS

- Operators are special symbols that perform specific operations with one, two or three operands, and then return a result.
- Operators are used to construct expressions.
  - ✓ Mathematical: +, -, / y \*.
  - ✓  $2 + 5$
  - ✓  $3 * 4 + 1 / 2$



# OPERANDS

- They are the input elements that the operators need to make their calculation or operation.
- They can be variables, constants, literals and other expressions.
  - ✓ E.g.:
    - ✓  $(2 * 5 + 1) + (3 / 2)$  : Two expressions as operands of a sum.

# TYPES OF EXPRESSIONS

- Simple
  - ✓ **One operand**
    - ✓  $x$
    - ✓  $5$
    - ✓  $PI$
    - ✓ No calculation is required to obtain its result.
- Composed
  - ✓ set of operands linked with operators
    - ✓  $x + 5$
    - ✓  $5 * PI / x$
    - ✓ Calculation is required to obtain its result.

# ASSIGNMENT

$\langle \text{Name Var} \rangle \leftarrow \langle \text{Expr} \rangle$

- Example:

✓  $X \leftarrow 5$

✓  $W \leftarrow X + 5 / \text{PI}$

✓  $Z \leftarrow 10 / 2 + 3 * (2 - 4)$

✓  $Z \leftarrow Z + 1$

- Example 2:

✓  $i \leftarrow 0$

✓  $i \leftarrow i + 2$

✓  $i \leftarrow (i * i) + 1$

✓  $i \leftarrow i - 5$

# INPUT

- **read** (X)
- The user who executes the program will enter by keyboard characters that will be saved in that variable.

# OUTPUT

- **print** "Welcome to my program"
- It shows in console the text.

# HELLO PROGRAM

```
algorithm a
    var NAME
    leer (NAME)
print ("Hello" + NAME + " how are you?")
End algorithm
```

# LET'S WORK A LITTLE BIT

- Let's develop an algorithm that requests the input of two numbers and calculate its product.
- Let's develop an algorithm that requests the input of two numbers and calculate its average.

# IF STATEMENT

It is an action to make a decision and execute under a condition certain actions.

We can branch to different sides of the algorithm

```
If <COND> then  
    Actions  
{else  
    Actions}  
End if
```

<COND> is an expression with a logical value.



# LOGICAL CONDITIONS

- These are expressions that may be true or false when evaluating.
  - ✓ They return logical values
- Relationship Operators
  - ✓ "<", "<=", ">", ">=", "==" y "!="
- Logical operators
  - ✓ "AND", "OR" y "NOT"

# IF STATEMENT

## Examples:

```
If x > 10 then
  x ← 0
else
  x ← 20
End if
```

```
If x == 10 AND y != 20 then
  x ← 0
  y ← 0
else
  x ← 20 + y
  y ← y + 20
End if
```

# IF STATEMENT

```
If <COND1> then
  Actions
Else If <COND2> then
  Actions
Else If <COND3> then
  Actions
...
else
  Actions
End if
```

# LET'S WORK

Let's develop an algorithm that requests the input of two numbers by keyboard and show which one is bigger.

EXERCISING

---

## PRACTICE 2

# ITERATION

- We use it to repeat an action or a set of actions a certain amount of times or under certain conditions.
  - ✓ Iterate comes from repeat
- The iteration action that we will use is called while and it is written like this:

```
while <COND> do  
  Actions  
end while
```

Where <COND> is a expression that returns True or False.

# ITERATION

- We evaluate the Condition
  - ✓ If it's True
    - Execute the actions
    - Jump to first step
  - ✓ If it's False
    - End
- The set of actions that are executed repeatedly is known as Cycle or Loop.
- An iteration is equivalent to a cycle or loop execution.

# EXAMPLE

I have to calculate the weight of all the students combined:

- For a student

```
var Number: weight, totalWeight;  
read(weight)  
TotalWeight ← weight
```

- If there are 2 students

```
var Number: weight, totalWeight;  
totalWeight ← 0  
read(weight)  
totalWeight ← totalWeight + weight  
read(weight)  
totalWeight ← totalWeight + weight
```



# EXAMPLE

- If we are 15

```
totalWeight ← 0
read(weight)
totalWeight ← totalWeight + weight
... (another 13 times)
read(weight)
totalWeight ← totalWeight + weight
```

- This is long and something doesn't seem right

```
totalWeight ← 0
while IHaveMoreStudentsToWeigh
  read(weight)
  totalWeight ← totalWeight + weight
end while
```

# EXAMPLE

---

This is very useful when I have to repeat a set of actions.

**And this happens a lot!**

The pending question is, how do I repeat the execution of such actions as many times as I want?

Let's write it in pseudo code...

# EXAMPLE

- The secret is to define the iteration condition correctly.
- A well-known way is to count the number of loops that we are doing and when we reach the desired amount we finish
  - ✓ We have to keep the amount of loops we took, and for this we use a variable.

```
count ← 0
while count < 5 do
    count ← count + 1
end while
```

- This variable is called counter

# EXAMPLE

**We had:**

```
totalWeight ← 0
while IHaveMoreStudentsToWeigh
  read(weight)
  totalWeight ← totalWeight + weight
end while
```

# EXAMPLE

**So applying this new concept:**

```
totalWeight ← 0
count ← 0
while count < 5 do
  read(weight)
  totalWeight ← totalWeight + weight
  count ← count + 1
end while
```

And we only have to vary the 5 for another value to contemplate more or less students.

In fact, we can use whatever expression we want.

# CONCLUSION (I)

Whenever there is a processing for a set of elements.  
We have to use an iteration that takes an element for each cycle and processes it.

```
While thereAreStepsToProcess do  
    doOne  
End While
```

It is not always easy, but the concept is always the same.

# CONCLUSION (II)

Generally speaking

Every time that we need to execute a certain number of steps.

It is convenient to use iterations and execute one step per cycle.

```
While thereAreSteps do  
    doOne  
End While
```

# ITERATION (III)

Inflating a tire is a good example of what is an iteration applied to real life.

```
while tirePressure < 30 do  
  tirePressure ← tirePressure + 1  
End while
```



# EXERCISE

---

Develop an algorithm that shows 10 times on screen a message that says "Hello my Friends".

# LET'S WORK A LITTLE BIT

---

Let's analyze and solve a practical problem

# EXERCISE

---

Develop an algorithm that adds the first 10 natural numbers.

# SOLUTION I

algorithm to add up first 10 Natural numbers

Number: sum

sum  $\leftarrow 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$

End algorithm

# SOLUTION II

algorithm to add up first 10 Natural numbers

Number: sum

sum  $\leftarrow$  0

sum  $\leftarrow$  sum + 1

sum  $\leftarrow$  sum + 2

sum  $\leftarrow$  sum + 3

sum  $\leftarrow$  sum + 4

sum  $\leftarrow$  sum + 5

sum  $\leftarrow$  sum + 6

sum  $\leftarrow$  sum + 7

sum  $\leftarrow$  sum + 8

sum  $\leftarrow$  sum + 9

sum  $\leftarrow$  sum + 10 // We have in sum what we  
were looking for

End algorithm

# NOTES

- The previous solutions are right, but...
  - ✓ If instead of adding the first 10 numbers, I want to add the first 20, or the first 5.
    - ✓ It was not considered
  - ✓ Even worse, what happens if the number of numbers to sum is determined at run time.
    - ✓ Not considered

# CONCLUSION

---

It is a correct algorithm, but very rigid and tied to a concrete situation.

Is that wrong?

No, not at all.

# NEW EXERCISE

- Develop an algorithm that adds the first  $N$  natural numbers.
  - ✓ Where “ $N$ ” is an arbitrary number
- Can this be done?
  - ✓ Of course.
  - ✓ But the previous rigid scheme no longer works.



# SOLUTION

```
algorithm sumNaturals
  var Number: sum
  sum  $\leftarrow$  0
  while thereAreNumbersToSum do
    sum  $\leftarrow$  sum + nextNumber
  end while
end algorithm
```

At each step of the iteration we add a number of the series.

- How many steps in total?
  - ✓ How do I iterate as many times as I need?
    - ✓ And how do I get the numbers needed to be added to the sum?

# HOW MANY ITERATIONS?

- If at each step I add a number, then in  $N$  steps I sum  $N$  numbers.
  - ✓ So, if I have 20 numbers, I'll have 20 iterations.
  - ✓ If I have 30 numbers, I'll have 30 iterations.
- We already know how many times I have to repeat the actions. Now let's see how I do this.

# HOW TO ITERATE?

- We already learnt this. But let's see it again
  - ✓ The secret is to correctly define the iteration condition.
  - ✓ For doing this, we have to count the amount of cycles, and decide when we want to stop at the wanted amount
    - ✓ We have to save the count of cycles somewhere
    - ✓ **We use a variable**

```
quantity ← 0
While quantity < 5 do
    quantity ← quantity + 1
End while
```

- This variable receives the name of Counter, because that precisely its role.

# HOW DO I GET THE NUMBERS?

- We have to generate the numbers somehow so that in each cycle we get a number that is different from the previous, so we add all the numbers.
  - ✓ One number per cycle.
- First number 1, number 2 in second cycle, and so on till N.

# GENERATING NUMBERS

Let's see how we generate the numbers

```
quantity ← 0
while quantity < 5 do
  print (quantity)
  quantity ← quantity + 1
end while
```

# SOLUTION II

Let's assume I want to add the first 5 numbers.

```
algorithm to sum Natural numbers
  var Number: sum, number
  number ← 0
  sum ← 0
  while number < 5 do // when number is 5, I'll
    be added all the numbers
    number ← number + 1 // generate next number
    sum ← sum + number // I add it to SUM
  End while
end algorithm
```

2 very useful techniques:

**Accumulator:** Variables that accumulate values along the cycles.

**Counter:** **Accumulator** that increments by 1.

# SOLUTION II

```
algorithm to sum Natural numbers
  var Number: sum, number
  const Number: N=5
  number  $\leftarrow$  0
  sum  $\leftarrow$  0
  while number < N do // when number is 5, I'll
    be added all the numbers
    number  $\leftarrow$  number + 1
    sum  $\leftarrow$  sum + number
  End while
end algorithm
```

If I want to add the first 15 natural numbers simply change the value of the constant N by 15.

# CONCLUSION

---

It is very important to know how to handle iteration actions. These are the ones that allow you to extend an algorithm. The accumulator and counter techniques presented for this case are applied in many other cases.



# ADDING OUTPUT

- So far we only made internal "calculations" and there is no visible result to the user, only internal processing.
- It is in our interest to show on screen the result of our operation.
- That is why we are adding an output to report the result.

# ADDING OUTPUT

Adding output to inform the result:

- algorithm to sum 10 Natural numbers
  - var Number: sum, number
  - const Number: N=10
  - number  $\leftarrow$  0
  - sum  $\leftarrow$  0
  - while number < N do
    - number  $\leftarrow$  number + 1
    - sum  $\leftarrow$  sum + number
  - End while
    - print("The sum of first "+ quantity +  
"natural numbers is " + sum)
- end algorithm

# ADDING INPUT

- If we add a constant we could modify the amount of values to add by modifying only that constant. However, this would mean a modification of the algorithm each time.
- How can I do to sum an arbitrary number of numbers defined at run time without modifying the program.
- An input of data would provide the solution.

# ADDING INPUT

- algorithm to sum N Natural numbers
  - var Number: sum, number, **quantity**
  - **read(quantity) // I'm making sure it works for any number**
  - number  $\leftarrow$  0
  - sum  $\leftarrow$  0
  - while number < quantity do
    - number  $\leftarrow$  number + 1
    - sum  $\leftarrow$  sum + number
  - End while
- print("The sum of first "+ quantity + "natural numbers is " + sum)
- end algorithm

# TO THINK

- First way of solving this exercise:
  - ✓  $\text{sum} \leftarrow 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$
- It is completely and logically right.
  - ✓ But it only solves a particular problem.
- It is better to have a more generic algorithm to solve the case of sum “N” numbers.
- In programming, we reward generic solutions that can be applied to multiple contexts and problems.

# CONCLUSION

- The computer does nothing magical, but only we program it to imitate us, the way we do things.
  - ✓ Only faster
  - ✓ Much more accurate
  - ✓ It doesn't have "bad days"...

# EXERCISES

---

Let's make an algorithm that calculates the multiplier of the first "N" natural numbers.

Let's make an algorithm that sums the first "N" even numbers.

# CONFUCIUS SAID

- I hear and I forget.
- I see and I remember.
- I **do** and I understand.

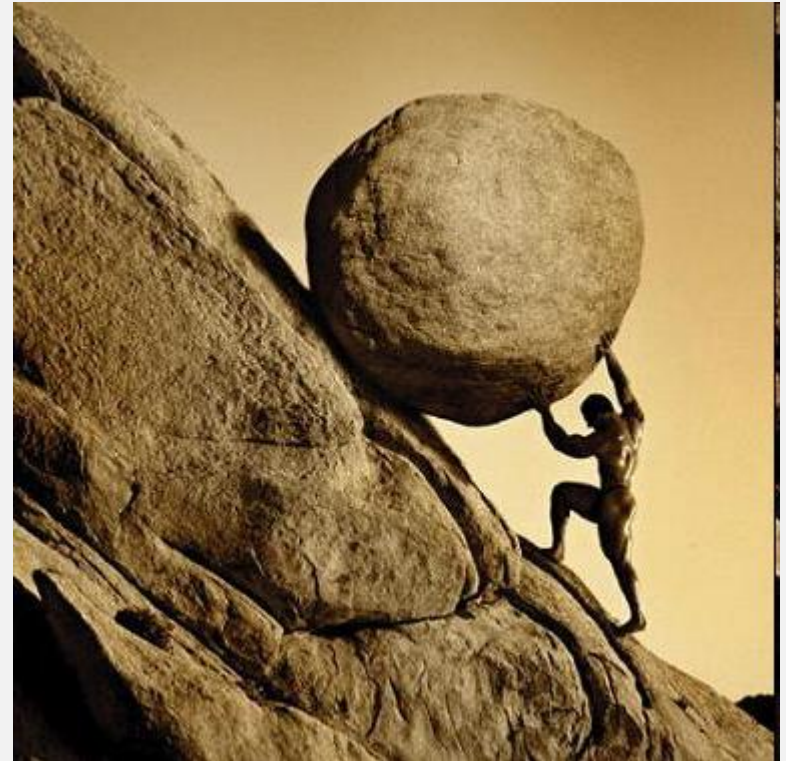




I WILL INSIST

**DO**

**It is the easiest way**



THE END

---

**QUESTIONS?**

# ComIT

## ALGORITHMS II

# THE TOPICS

- More Flow Control structures
  - ✓ Multiple selector
  - ✓ For
- Datatypes
  - ✓ Applied to Variables, Constants, Expressions and Operators
  - ✓ Type compatibility
  - ✓ Conversions
- Operators Precedence
- Identifiers and Reserved Words
- Nomenclature

# MULTIPLE SELECTOR

This action is used to select an alternative among several based on the value returned by an expression.

```
switch <expr> do
  <value 1> : Actions
  <value 2> : Actions
  ...
  <value n> : Actions
  {default: Actions}
end switch
```

What we do is to specify for each possible value of the expression what action or actions to follow.

# MULTIPLE SELECTOR

1. We evaluate the expression
2. Based on the return value, we decide the actions to execute
3. Execute actions
4. End

```
switch <expr> do
    <value 1> : Actions // when expr is <value 1>
    <value 2> : Actions // when expr is <value 2>

    <value n> : Actions // when expr is <value n>
    default: Actions // for other values
end switch
```

# MULTIPLE SELECTOR

- Each alternative is known as Case.
- You can not put in the cases ranges or conditions but only concrete values.
  - ✓ But you can put several values separated by commas
- Not contemplated values can be treated in the case **default**, which is optional.

# MULTIPLE SELECTOR

Example:

```
switch grade do
  1,2,3 : print ("Very bad")
  4,5   : print ("Mmm.....")
  6,7   : print ("Good")
  8,9   : print ("Very good!")
  10    : print ("Excellent!!!")
  default : print ("Invalid grade")
end switch
```



# MULTIPLE SELECTOR

You can define several actions for a Case

```
switch grade do
  1,2,3 : print ("Very bad")
  4,5   : print ("Mmm.....")
        grade ← 6 // I love this teacher 😊
  6,7   : print ("Good")
  8,9   : print ("Very good!")
  10    : print ("Excellent!!!")
  default : print ("Invalid grade")
end switch
```

# MULTIPLE SELECTOR

The multiple selection ("switch") is another way of **if... then**.

Generally speaking, we use it when we have to take 3 or more different paths based on different values of return from an expression.

EXERCISING

---

## PRACTICE 2

# FOR

## FOR

It is an action that provides a compact way of iterating using a counter.

We use it when we know the number of cycles beforehand.

```
For <Var> ← <expr. Initial value > to  
<expr. Final value> do  
    Actions  
end para
```

# FOR

It is a easier than **while** to perform iterations when we know beforehand the number of turns to give.

```
for <var> ← <expr. initialvalue > to <expr. final  
value > do  
    Actions  
end for
```

## ***Equals***

```
<var> ← <expr. initialvalue >  
while <var> < <expr. final value > do  
    Actions  
    <var> ← <var> + 1  
end while
```

# FOR

Example:

```
var Number : ac ← 0, i
for i ← 1 to 5 do
    ac = ac + i
end for
```

## Warning!

The variable we use as a counter within the For must be pre declared.

# REMEMBERING

- **Data** is represented by a **value**.
- We have **numerical** values, **logic**, **characters** and **strings**.
- So 1, 22, 334 are numbers and have arithmetic properties.
- T or F, are logic values with its own properties.
- ‘A’ , ‘%’ , ‘\$’ , ‘|’ , are symbols or characters.
- “Hello” , “Bye” , “Very nice” , are strings.

# NATURE = TYPE

- From these observations we can see a "nature" of the data that leads us to classify them into data types.
- Data with similar characteristics and properties can be put in the same groups, that we call types.



# NATURE = TYPE

- Data types define a set of values and, in an associated way, a set of operations to work with those values.
- So far we have worked with types of numerical data but there are more.

# TYPES

- Number:
  - ✓ Operands: Numbers
  - ✓ Operators: +, -, /, div, mod and relationals.
- Boolean:
  - ✓ Operands: T or F.
  - ✓ Operators: “AND”, “OR”, “NOT”, “==” y “!=”.
- String
  - ✓ Operands: strings (delimited with “”)
  - ✓ Operators: “+” (concatenate) and relationals.
- Char
  - ✓ Operando: symbols (delimited with ‘ ’)
  - ✓ Operators: relational

# ARE THERE MORE TYPES?

- **Yes there are, but generally speaking, those are composition of the previous ones**
  - ✓ Compound data types
- For now we will start working on processing data of basic types and then moving on to escalate into problems that require more complex types.

# TYPES

- A datatype has:
  - ✓ A set of values
  - ✓ A set of operators / operations applicable to these values.

# TYPES IMPORTANCE

- Data types are a concept that appear throughout everything we've been doing so far.
  - ✓ From declaring a variable to creating expressions.
- In general, we speak of types of data and just types indistinctly, unless otherwise specified.

# VARIABLES, CONSTANTS, EXPRESSIONS AND OPERATORS

- Variables and constants store data
  - ✓ Data has a specific type → variables and constants **are from a specific type**, to indicate the type of the stored value.
- Expressions **also return values of specific types**:
  - ✓ Numeric expressions return numeric values.
  - ✓ Boolean expressions return boolean values
  - ✓ String expressions return string values.
- Operators **link operands of specific types**

# VARIABLES AND CONSTANTS

- This is how we declare variables:
  - ✓ `var Number : x`
  - ✓ `var String : name`
  - ✓ `var Boolean : isOn`
  - ✓ `var Char : inputKey`
- We can declare several variables in one line
  - ✓ `var Number : x , y, z`
- We can initialize the variables when we create them
  - ✓ `var Number: x  $\leftarrow$  0, num  $\leftarrow$  2, w  $\leftarrow$  65`
  - ✓ `var Boolean : thereIsPeople  $\leftarrow$  T`

# DECLARATION AND ASSIGNMENT

- You can not declare two variables with the same name.
- A variable has a single type.
- You can not assign a variable of a type to a value corresponding to another type.



# TYPE COMPATIBILITY

- It is a concept that we will expand throughout the course.
- Some pseudo code languages differentiate the value numbers this way:
  - ✓ Float
  - ✓ Integer
- Where it is admitted that a float is assigned an integer but not the other way around.

# IMPLICIT CONVERSION

- Without explicitly saying it, we are converting a data type.
- `var Integer : x ← 2.5`
  - ✓ x is going to be 2
- But we are not doing this in our pseudo code
  - ✓ That's why we use Number and not Integer 😊
  - ✓ In Java there are conversions

# EXPLICIT CONVERSION

- `var String : cad ← 2`
  - ✓ This can't be done
  - ✓ “2” is different from 2.
- But we can convert one into the other using functions.
  - ✓ We'll see those functions later

# EXPRESSIONS

Combinations of operands and operators that return a single value of a certain type.

We can only combine operands with compatible operators.

Is this OK:  $5 + T$ ? or  $\text{!}T - \text{"Orange"}$ ?

Operators take operands from specific types

# OPERATORS

- **Operators are described by**
  - ✓ **Number of operands** they take
    - ✓ Unary
    - ✓ Binary
  - ✓ **Operand Type**
  - ✓ **Return value Type**

# OPERATORS

- Arithmetic Operators
  - ✓ /, \*, div, mod, + and -.
  - ✓ Take numeric operands and return numeric values
- Relational Operators
  - ✓ <, <=, > y >=
  - ✓ Take numeric, char or string operands and return boolean values
  - ✓ ==, !=
  - ✓ Take same type operands in both sides and return boolean values.
- Logic/Boolean Operators
  - ✓ “NOT”, “AND” and “OT”
  - ✓ Take boolean operands and return boolean value.
- String Operators
  - ✓ +
  - ✓ Take string operands or numeric+string and return strings.

# OPERATOR “+”

- Works as operator for string and numeric type operands.
- How can we distinguish that it is for an arithmetic sum or for concatenation?
- We know that
  - ✓  $5 + 5 = 10$
  - ✓ “A” + “B” = “AB”
- What’s the result from the following?
  - ✓ “A” + 5 = ?
  - ✓  $5 + \text{“A”} = ?$

# OPERATOR “+” (II)

- Both return a string.
  - ✓ “A” + 5 = “A5”
  - ✓ 5 + “A” = “5A”
- But...
  - ✓ “a” + 5 + 5
  - ✓ What’s the return value?
- We’ll see....



# OPERATORS PRECEDENCE (I)

- Which operators have higher priority for linking operands.
  - ✓ They determine the order of evaluation of the terms of an expression.
- Example: let's use parenthesis
  - ✓  $2 * 5 - 4$
  - ✓  $2 + 4 / 2$

# OPERATORS PRECEDENCE (II)

1. NO , - (*change of sign*)
2. /, \*, mod, div
3. +, -
4. <, >, <=, >=
5. ==, !=
6. AND
7. OR

Logic Operators

Relational Operators

Arithmetic Operators

# OPERATORS PRECEDENCE (III)

- What happens when two operators with the same level of precedence appear sharing a single operand?
- Example:
  - ✓  $2 / 1 * 2$ 
    - ✓  $(2 / 1) * 2 = 4?$
    - ✓  $2 / (1 * 2) = 1?$

# OPERATORS PRECEDENCE (VI)

- Associativity of operators is what determines the order of evaluation.
  - ✓ Binary operators are associative from left to right.
    - ✓ This means that the shared operand is always taken by the operator on the left.
    - ✓ We will associate (putting brackets) and evaluating the operators from left to right.

# OPERATORS PRECEDENCE (V)

Example:

$a - b + c - d$  equals to  $((a-b) + c) - d$ .

We evaluate from left to right. First “a-b” then + c, and finally - d.

$a - b + c - d * e$  equals to  $((a-b) + c) - (d * e)$ .

# OPERATOR “+” (III)

- Do you remember about “a” + 5 + 5
  - ✓ ((“a” + 5) + 5)
- Then it returns
  - ✓ “a55”
- If I do “a” + (5 + 5)
- Then I have
  - ✓ “a10”

# OPERATORS PRECEDENCE (VI)

Example:

$3 == 3 == 8 == 8$

How do you evaluate it?

$((3 == 3) == 8) == 8$

This doesn't make sense

How do we parenthesize to make it right?

# OPERATORS PRECEDENCE (VII)

- $(3 == 3) == (8 == 8)$ 
  - ✓  $T = T$ , which gives us  $T$  😊
  - ✓ `var Boolean : log ← (3 == 3) == (8 == 8)`
  - ✓ `log ← (T) == (T)`
- Conclusion: **precedence matters!**



# IDENTIFIERS (I)

An identifier is a name (variable, constant, algorithm etc.) that allows us to refer to an element to be used in our program. They are lexical elements of our language.

# IDENTIFIERS (II)

The rule for declaring valid identifiers is as follows: The word or name must begin with a letter or a "\_" and then an arbitrary sequence of letters, numbers or a "\_".

# IDENTIFIERS (III)

- Identifiers are case sensitive.
  - ✓ Example: variables `dough`, `Dough` and `DOUGH` are considered different.
- We usually follow certain rules that facilitate the reading and maintenance of computer programs.

# RESERVED WORDS

A reserved word is every sequence of characters that is defined with the end of specifying what we want to do within the program. Its meaning can not be redefined and new reserved words can not be created.

The word Var refers to variable and serves to declare variables.

The word Number refers to the numeric data type.

# RESERVED WORDS

- **Reserved Words**
  - ✓ Predefined Constants
    - ✓ T, F
  - ✓ Operators
  - ✓ Predefined Functions
  - ✓ Instructions names
  - ✓ Declarations names

.

# NOMENCLATURE (I)

**Nomenclature:** Writing style that is applied to sentences or compound words.

- There are good practices for us to write the algorithms.
- Facilitates the reading of algorithms from other people
- We will use it when we program
- The grade depends on this too!

# NOMENCLATURE (II)

We usually use lower case for names.

But, these are the most commonly used nomenclatures

1. lowerCamelCase
2. UpperCamelCase
3. UPPER\_CASE

# NOMENCLATURE (III)

## 1. lowerCamelCase

- ✓ First word in lowercase and successive words with first capital letter, without spaces.
- ✓ Used for:
  - ✓ Variables
  - ✓ Algorithms and Subalgorithms

Examples:

```
algorithm sum
```

```
algorithm determineBigger
```

```
var Number : x
```

```
var String : goalDifference;
```



# NOMENCLATURE (IV)

## 2. UPPER\_CASE

- ✓ All capitalized words, separated by "\_".
- ✓ Used for:
  - ✓ Constants

Examples:

```
const Number : PI = 3.14
```

```
const String : END_GREETING = "THANK YOU  
FOR USING"
```

# NOMENCLATURE (V)

## 3. UpperCamelCase (PascalCase)

- ✓ All words capitalized, no spaces.
- ✓ Used for:
  - ✓ Data Types

Examples:

`Number, String, Char, Boolean.`

# NOMENCLATURE (VI)

- Indentation
  - ✓ Actions belonging to an algorithm are tabulated to the right
  - ✓ All actions that are within a control structure are tabulated to the right
    - ✓ Increases legibility

# NOMENCLATURE (VII)

- Identifiers

- ✓ The names used should be clear

- ✓ `var` Number : two  $\leftarrow$  3 // it's confusing

- ✓ `num`  $\leftarrow$  beatles / ROLLING\_STONES

- ✓ `algorithm` printScreen

- ✓ `x`  $\leftarrow$  2

- ✓ `end algorithm` // IT DOESN'T PRINT  
ANYTHING

- ✓ The general rule is: The smaller and descriptive the name, the better.

EXERCISING

---

## **PRACTICE 3**

THE END

---

**QUESTIONS?**

# ComIT

## ALGORITHMS III

# THE TOPICS

- Modular Programming
  - ✓ Subalgorithms
    - ✓ Definition
    - ✓ Invocation
  - ✓ Variable Scope
  - ✓ Return Value
  - ✓ Parameters and Arguments
  - ✓ Sending Parameters / Arguments
  - ✓ Declaration and Definition
  - ✓ Predefined Funciones
  - ✓ Modularity



# INTRODUCTION (I)

- So far we worked writing all the actions of my program within one algorithm.
  - ✓ “1 program = 1 algorithm”
- If we have a bigger program and everything is written into a single algorithm, it will end up being an overwhelming elephant.
  - ✓ An algorithm with more than 30 actions begins to become unreadable for most people.

# STRUCTURED PROGRAMMING

- The programming way that we were studying is called structured programming.
  - ✓ It has a writing discipline
    - ✓ Declaration of variables at the beginning
    - ✓ Standard Nomenclature Usage
  - ✓ 3 types of actions
    - ✓ No inendit cycles, no usage of unconditional selectors
  - ✓ One start and one end

# INTRODUCTION (II)

- To address larger developments we need an approach that allows us to handle the intrinsic complexity that every large problem has.
- We will see an approach that consists of separating a program into several algorithms.
  - ✓ “1 algorithm = N subalgorithms”
    - ✓ Each one solves a smaller part (subproblem) that is, of course, easier to raise and build.
    - ✓ Combined they solve the problem in its entirety

# MODULAR PROGRAMMING

This is known with the name of modular programming.

We do not stop applying the principles of structured programming.

# INTRODUCTION (III)

- Modular programming proposes to decompose a program into smaller parts called modules
  - ✓ Same as we did in natural language
    - ✓ We use “the top-down” philosophy
- Each module performs a specific task and works independently from the rest.
- Within the pseudo code, a module is known with the name of subalgorithm.

# SUBALGORITHMS

- Subalgorithms group under a name a set of actions.
- It is, in essence, an algorithm that also incorporates the ability to be used from another algorithm.
  - ✓ Also called “non-primitive actions”

# SUBALGORITHMS - PROS

- It allows to approach the development of a complex algorithm, by dividing it into several subalgorithms simpler to do.
  - ✓ All together solve the original problem.
- It reduces the size of an algorithm and makes it more readable.
- Each subalgorithm can be developed and tested independently by different people.
- Teamwork is improved by being able to clearly divide tasks.
- Work done on a project can be re-used in other projects

# SUBALGORITHMS (I)

Subalgorithms have the same form as the algorithms, only it contains a few brackets after the name and, optionally, parameters and a value of return.

```
algorithm <name> ({<parameters>}) {: <type>}  
    // data  
    // actions  
    {return <expr>}  
end algorithm
```



## SUBALGORITHMS (II)

```
algorithm insertNameAndGreeting ()  
    var String: name;  
    read(name);  
    print ("Hello " + name);  
end algorithm
```

```
algorithm insertAreaAndShowRadio ()  
    var Number: sup;  
    const Number: PI = 3.1416;  
    read(sup);  
    print (sup * PI * PI);  
end algorithm
```

Now we need to know how to use this.

# USE OF SUBALGORITHMS

- Subalgorithms are used by invoking them from another algorithm.
- To invoke we write the name of the subalgorithm followed by the parentheses.
  - ✓ Invocation is a sequence action
- The algorithm from which it is invoked is known as the invoking module

```
algorithm simple
    // ...
    insertNameAndGreeting()
    // ...
    insertAreaAndShowRadio ()
end algorithm
```

# USE OF SUBALGORITHMS

Invoking (or calling) a subalgorithm means executing it

The execution flow continues inside the subalgorithm, executing its actions sequentially, and when it finishes, it returns to the invoking module and continues executing the subsequent actions.

# SUMMARY

---

A subalgorithm is also an algorithm that in addition:

- Incorporates the ability to be invoked
- Can have a set of parameters
- Can return a value

# VARIABLES' SCOPE

- So far we had variables and constants declared only within a single algorithm.
  - ✓ What happens when we have a set of algorithms?
    - ✓ Can one algorithm access the variables of another?
    - ✓ What happens if in two algorithms I have two variables with the same name?
  - ✓ The answer, next 😊

# VARIABLES' SCOPE

- To answer this we have to introduce the concept of scope of a variable.
- It is understood by scope of a variable, to the part where the variable is accessible and can be used.
- The variables defined in each algorithm or subalgorithm belong only to it and are not shared with each other.
  - ✓ Variables are local to the module where they were defined.

# VARIABLES' SCOPE

- Meaning that:
  - ✓ Two variables of different scopes are two different variables, even if they are declared with the same name.
  - ✓ If within one algorithm I invoke another subalgorithm that has variables with same name there will be no problems of ambiguities, since they are different.
- There are “global variables” but we won’t use them for now

# VARIABLE DEFINITION

A variable:

- **Has a name** that identifies them
- **Has a type** that describes their use.
- **Has a content** that is the value or data that they store.
- **Belongs to a scope**, where they can be accessed.



# VARIABLES' SCOPE

```
algorithm a1
  var Number: x
  x ← 10
  a2()
  print (x)
end algorithm
```

```
algorithm a2 ()
  var Number: x
  x ← 20
  print (x)
end algorithm
```

What are we going to see on console?

# RETURN VALUE

Subalgorithms can return a value.

- ✓ We have to specify its type
- ✓ We have to return a concrete value
  - ✓ We use **return** as last sentence

```
algorithm calculateE() : Number
  var Number : ac, d, i
  ac ← 0
  d ← 1
  for i ← 1 to 20 do
    d ← d * i
    ac ← ac + 1 / d
  end for
  return ac + 1
end algorithm
```

## RETURN VALUE

In this way we can use a subalgorithm to form expressions, such as if it were an operand.

```
x ← calculateE() * 2 + 6 / 2 - 3  
x ← calculatePI()
```

## QUICK QUESTION

Is there any difference?

```
algorithm add4with4 () : Number
    var Number : res
    res  $\leftarrow$  4 + 4
    return res
end algorithm
```

```
Algorithm add4with4Version2 () : Number
    return 4 + 4
end algorithm
```

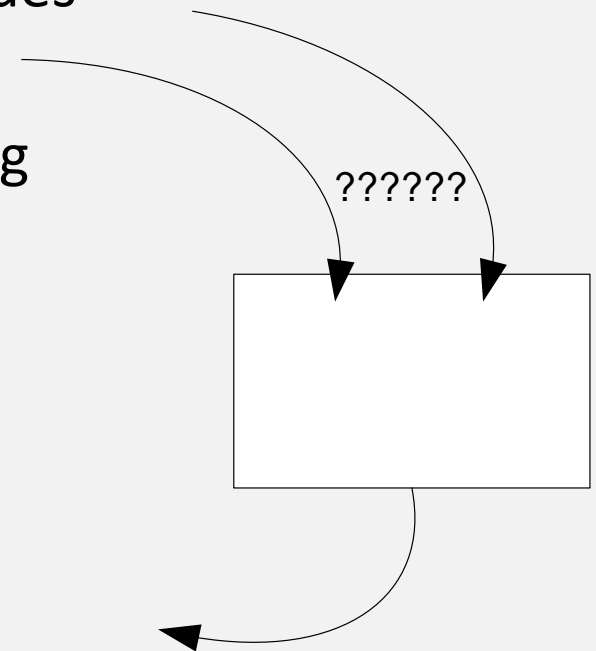
# INPUT VALUES (I)

We saw that subalgorithms have output values that we call return value.

It is the value that is returned to the invoking module.

Are there any input values that the subalgorithm receives when invoking it?

Values that you receive from the invoking module.



# INPUT VALUES (II)

Yes, and for that we need to...

- On the one hand, when defining it, specify in the subalgorithm that you are going to receive data of a certain type
  - ✓ **Parameters**
- On the other hand, when invoking it, passing concrete data
  - ✓ **Arguments**

# WHY PARAMETERS?

- Why do I add input values? To complicate our lives?
- Let's think about it:
  - ✓ Having a function that calculates the sine of an angle that we specify.
    - ✓  $\text{Sin}(x)$
  - ✓ Have a different function for each possible angle to calculate sine.
    - ✓  $\text{Sin}_1()$
    - ✓  $\text{Sin}_2()$
    - ✓ ...
    - ✓  $\text{Sin}_{360}()$

# WHY PARAMETERS?

- The idea of parameterizing a subalgorithm (adding parameters) is to extend or generalize the utility of it so that it adapts to various uses.
  - ✓ A subalgorithm can be generalized if it is correctly parameterized.
    - ✓ It is not always easy to realize that something must be a parameter



# PARAMETERS AND ARGUMENTS

- Let's take  $\sin(x)$ .
  - ✓ “**x**” is the **parameter**.
  - ✓ To use this function, we need to say what value “x” has.
    - ✓  $\sin(10)$  means that “10” is the expression used as **argument**.

# PARAMETERS AND ARGUMENTS

- Parameters are variables of a subalgorithm that will assume a value when that subalgorithm is invoked.
- The value that is actually assigned to a parameter of a subalgorithm when invoked is called argument.
- The order in which the parameters are declared and the arguments are passed, establishes the correspondence between them.

# PARAMETER DECLARATION

Parameters are declared as variables without the `var` and between parenthesis.

```
algorithm printTax(Number : amount)
    Var Number: tax
    tax ← amount * 0.0314
    print (tax)
end algorithm
```

**amount will have a value when the subalgorithm gets invoked**

# PARAMETER DECLARATION

If there is more than one parameter we separate them with comma.

```
algorithm printTaxAndText (Number : amount,  
String: text)  
    Var Number: tax  
    tax ← amount * 0.0314  
    print (text)  
    print (tax)  
end algorithm
```

# INVOKING WITH ARGUMENTS

- Arguments are expressions that go in parentheses to specify which value the parameters assume.
- If an algorithm has parameters, invoking it must include arguments.
- If an algorithm has no parameters, invoking it includes no arguments.

```
algorithm tax
    var Number: amount
    printNameAndGreeting()
    print ("Insert an amount")
    read (amount)
    printTaxAndText(amount, " the tax is ")
end algorithm
```

## IN SUMMARY

- Parameters appear in the definition of the algorithm as comma-separated variable declarations.
- Arguments appear in invocations of the algorithm as expressions separated by commas.

# PARAMETERS AND RETURN VALUES

Let's see how it would look like a subalgorithm with parameters that returns a value.

```
algorithm absoluteValue (Number : num) : Number
    var Number : res
    if (num < 0) entonces
        res ← -num
    else
        res ← num
    return res
end algorithm
```

This way you can generate very interesting subalgorithms that receive parameters and return a result.

# PARAMETERS AND RETURN VALUES

Another interesting sub algorithm that we can define with parameters and return value

```
Algorithm isPrime (Number : num) : Boolean
    var Number : i
    var Boolean: isIndivisible ← T
    i ← num - 1 // take previous Number
    while (i > 0) AND (isIndivisible) do
        isIndivisible ← (num mod i != 0)
        i ← i - 1;
    end while
    return (isIndivisible)
end algorithm
```



# VARIABLES' SCOPE

Parameters play the role of local variables within a subalgorithm. There can NOT be a local variable with the same name as a parameter.

```
algorithm a1 (Number : x)
    var Number : x
    x ← 10
    print (x)
end algorithm
```

**WRONG**

# SENDING PARAMETERS / ARGUMENTS

- When we have parameters and arguments:
  - ✓ On one side (arguments) I have expressions and on the other side (parameters) I have variables.
  - ✓ When invoking the subalgorithm occurs an assignment that we do not explicit, where the value returned by the expression is copied in the variable that is the parameter.

# SENDING PARAMETERS / ARGUMENTS

Let's see an example

Be a1 an algorithm that receives a Number type parameter.

```
algorithm a1 (Number : x)
  x ← x + 10
  print (x)
end algorithm
```

Another algorithm invokes a1 :

```
algorithm principal
  a1 (5 + 10 * 2)
end algorithm
```

What will be printed on screen?

## SENDING PARAMETERS / ARGUMENTS

- At runtime, this expression is resolved and terminated by calling a1 (25)
  - ✓ The parameter x of subalgorithm a1 is assigned 25
  - ✓ And in this way you begin to execute the algorithm a1 with the variable x with the mentioned value.

# SENDING PARAMETERS / ARGUMENTS

Another example:

Be a1 an algorithm that receives a Number type parameter

```
algorithm a1 (Number : x)
  x ← x + 10
  print (x)
end algorithm
```

Another algorithm invokes a1

```
algorithm principal
  var Number: x
  x ← 10
  a1 (x)
  print (x)
end algorithm
```

What will be printed on screen?

# SENDING PARAMETERS / ARGUMENTS

- At runtime, this expression is solved and terminated by calling `a1 (10)`, since `x` is 10.
  - ✓ It will happen that the parameter `x` of `a1` will be assigned a 10
  - ✓ And in this way you begin to execute the algorithm `a1` with the variable `x` with the correct value.
- But by finalizing `a1` and returning to the algorithm from which I invoke it, what will be printed on the screen?
  - ✓ The `x` defined as parameter in `a1` is different from the `x` defined as local variable in `main`, remember that two variables of different scopes are two different variables.

# SENDING PARAMETERS / ARGUMENTS

In-class exercise:

```
algorithm a1
  var Number : x ← 20
  a2(x)
  print (x * 2)
  x ← x * 2
  print (x * 2)
end algorithm
```

```
algorithm a2(x : Number)
  print (x)
  x ← x / 2
  print (x)
end algorithm
```

If we execute a1, what will I see in console?

# SUMMARY (I)

- Subalgorithm is a set of actions that perform a specific task.
  - ✓ It can optionally take input values, called parameters and provide an output value, called return value.
- To invoke an algorithm means execute it
  - ✓ This involves solving the passage of arguments and parameters (if there are parameters)
  - ✓ Then execute sequentially each of the actions
  - ✓ And when it ends up returning to the algorithm from which we invoked it, returning a value (if there is return value).



## SUMMARY (II)

- The algorithm from which another is invoked is known as the invoking module.
- The parameters and return value are the ways that a module has to communicate with the invoking module.
  - ✓ Parameters are input values
  - ✓ Return value is the output value
- All subalgorithms are invoked using parentheses
  - ✓ If there are parameters, arguments go in the parentheses
  - ✓ In case of not having parameters, only the empty parentheses go

## SUMMARY (III)

- Subalgorithms are classified into
  - ✓ **Functions:** have return value
  - ✓ **Procedures:** have no return value
- Functions can be used as one more operand within expressions.
  - ✓ Such as variables or constants.
    - ✓ `isPrime(2) AND amount == 1000`
    - ✓ `absoluteValue(-2) * 2 / absoluteValue (-(x + y))`

## SUMMARY (IV)

- The return value of a function does not have to be used in the invoking module.
  - ✓ I can write like this:
    - ✓ `// ... moreactions`
    - ✓ `isPrime(-2) // I just invoke it`
    - ✓ `// ... more actions`
  - ✓ Doesn't have much sense, but we can do it.
- To define which algorithm takes control when starting the program execution, it is clarified by omitting the `()` in its declaration or, writing "<name algorithm> is the main one" above everything.

# DECLARATION AND DEFINITION

- There are two parts on any algorithm / subalgorithm
- Declaration
  - ✓ Header
    - ✓ `algorithm` <name> ({<parameters>}) {:  
    <type>}
- Definition
  - ✓ Body
  - ✓ Data and actions

# PREDEFINED FUNCTIONS

- Subalgorithms already defined in the system.
- In pseudo code we have predefined functions to perform operations, among them we have already seen two:
  - ✓ read (x)
  - ✓ print (x)

# PREDEFINED FUNCTIONS

- Arithmetic Functions:
  - ✓ `abs (<num expr> )`: Returns the absolute value of the passed numeric expression as parameter.
  - ✓ `sen (<num expr>):` : Returns the sine of the passed numeric expression as parameter.
  - ✓ `cos (<num expr>):` : Returns the cosine of the passed numeric expression as parameter.
  - ✓ `tan (<num expr>):` : Returns the tangent of the passed numeric expression as parameter.
  - ✓ `random ()`: Returns a random number between 0 (inclusive) and 1 (not inclusive)
  - ✓ `toString (<num expr>)`: Returns the representation as string of the passed expression as parameter.

# PREDEFINED FUNCTIONS

- String Functions:
  - ✓ `long (<text expr>)`: Returns the number of characters in the string passed as parameter.
  - ✓ `charAt(<text expr>,<num expr>)`: Returns the character in the string passed as first parameter in the position indicated by the second parameter. If such a position doesn't exist, an error occurs. We consider 0 as the beginning.
  - ✓ `toNumber(<text expr>)`: Returns the numeric value of a string. If the string does not equal a Number then an error occurs.

# MODULAR PROGRAMMING PHILOSOPHY

## Divide and Conquer

The program will take the form of a large algorithm that invokes other algorithms in a certain order to achieve our goal.

```
algorithm p
  a1 ()
  a2 (...)
end algorithm
```

```
algorithm a1 ()
  ...
end algorithm
```

```
Algorithm a2 (...) : ...
  ...
end algorithm
```



# MODULAR PROGRAMMING PHILOSOPHY

- Each algorithm should function as a module that is independent from others.
  - ✓ In other words
    - ✓ **High cohesion:** well defined tasks.
    - ✓ **Low coupling:** A change in one doesn't mean a change in the others.
  - ✓ This is not easy to do a requires a lot of practice.

# MODULAR PROGRAMMING - EXAMPLE

- A supermarket offers discounts of 30% for all purchases and 35% for those that exceed 1000 dollars\*
- Discounts are 500 dollars at the most.
- Let's create an algorithm so that the person can enter the value of a purchase and calculate how much would be the discount.

## EXAMPLE - WITHOUT MODULES

```
algorithm calculateNet
  var Number : gross, disc, net
  read(gross)
  if (gross < 1000) then // If spend is less
    than 1000 then I apply 30%
    disc =(gross * 30) / 100
  else // if spend is more than 1000 then I
    apply 35%
    disc = (gross* (35)) / 100
  end if
  if disc > 500 then // There won't be discount
    over 500 dollars
    disc = 500
  end if
  net ← gross - disc
end algorithm
```

# EXAMPLE - WITH MODULES

```
algorithm calculateNet
  var Number : gross, disc, net
  read(gross)
  disc ← calculateDiscount (gross)
  net ← gross - disc
end algorithm
```

```
algorithm calculateDiscount (Number : amount) : Number
  var Number : disc
  if(amount < 1000) then // If spend less than 1000
  then apply 30%
    disc ← (amount * 30) / 100
  else // if spend is more than 1000 then apply
  35%
    disc ← (amount * 35) / 100
  end if
  if disc > 500 then // There won't be discount over
  $500
    disc ← 500
  end if
  return disc
end algorithm
```

# EXAMPLE - MODULES + BEST PRACTICES

```
algorithm calculateNet
  var Number : gross, disc, net
  read(gross)
  disc ← calculateDiscount (gross)
  net ← gross - disc
end algorithm
```

```
algorithm calculateDiscount (Number : amount) : Number
  const Number : PERC_BASE = 30, PERC_AD = 35 // define
  constants
  var Number : disc
  if(amount < 1000) then // If spend less than 1000 then
    apply 30%
    disc ← (amount * PERC_BASE) / 100
  else // if spend is more than 1000 then apply 35%
    disc ← (amount * (PERC_AD)) / 100
  end if
  if disc > 500 then // There won't be discount over $500
    disc ← 500
  end if
  return disc
end algorithm
```

## EXAMPLE - IMPROVEMENTS

- If you were to have two purchases, then applying discount for the two would be as simple as invoking this algorithm twice and not writing all this for each sale.
- If tomorrow rules change (eg, discount is 20%), the only part that changes is the subalgorithm.
  - ✓ Remaining unchanged the rest of the program
  - ✓ Without subalgorithms you would have to change every part of the program where you used discounts.
    - ✓ **The change occurs in only one part and impacts the rest.**

# MODULARITY

- We don't always know how to break down a problem.
  - ✓ There is not an absolute truth
  - ✓ It depends on the particular problem, the knowledge that we have about it and the subjectivity of the developer

# MODULARITY

## Tip 1

- If, as we develop, we see portions of similar code.
- Then that code is candidate to be modularized in a subalgorithm.



# MODULARITY

## Tip 2

- When we see two pieces of code that perform the same operation but with different data.
- Data is set as parameter and the operation, now encapsulated in the subalgorithm, will work with said parameters.

# MODULARITY

## Tip 3

- If I can group a set of actions under a name.
- Then these actions are candidates to be modularized.

# MODULARITY

## Tip 4

- Keep in mind that a change that is made in a module is spread everywhere it is used.
  - ✓ This is a direct consequence of writing a process only in one place.
  - ✓ This facilitates subsequent modifications.

# MODULARITY

- The idea is to reuse modules in other contexts or other occasions within the same context.
  - ✓ Our goal is to get:
    - ✓ High Cohesion
    - ✓ Low Coupling

# HIGH COHESION

**Cohesion:** Measurement of the degree of identification of a module with a specific function or task.

The function that the subalgorithm does has to be well defined.

Example:

- If a module calculates a discount for an "X" amount, it should not print anything on the screen or ask for keyboard input.
  - ✓ It only has to do what it says it does

# LOW COUPLING

**Coupling:** Measure of interaction between the modules of my program.

- The idea is that a change in one module does not necessarily mean a change in another.
- Modules should be independent.
- Only communicating with each other through parameters and return values.

# MODULARITY

- It requires more thought
  - ✓ It requires the use of abstraction and thinking in more general terms as specific operations are carried out.
- It is less efficient
  - ✓ Breaking down a problem into parts involves adding a "bridge" between those parts for everything to work correctly, and this creates an additional burden.
  - ✓ Module invocations require additional time.

# MODULARITY

---

- We have already seen that separating a program into parts is not trivial.



# ABOUT PARADIGMS

- Modular programming is not exclusive to structured programming, but on the contrary, they are complementary.
- Let us observe, that within each module (subalgorithm) we continue to program in a structured way as we did with the first algorithms.
- Now we program thinking about the paradigms.

# EXERCISE

Algorithm to sum first N Numbers that are divisible by 3 with first N Numbers divisible by 5.

If  $N = 1 \rightarrow 3 + 5$

If  $N = 2 \rightarrow (3 + 6) + (5 + 10)$

If  $N = 5 \rightarrow (3 + 6 + 9 + 12 + 15) + (5 + 10 + 15 + 20 + 25)$

EXERCISING

---

## PRACTICE 4

THE END

**QUESTIONS?**

# ComIT

## ALGORITHMS IV

# THE TOPICS

- Programming languages
  - ✓ Machine Language
  - ✓ Assembly Language
  - ✓ High Level Languages
- Translators
  - ✓ Interpreters
  - ✓ Compilers
- Source Code
- Object Code
- Compilation, OS and Architectures
- Hardware Concepts for development

# PROGRAMMING LANGUAGES

- So far we have only worked with pseudo code that involves a fictitious interlocutor who takes what we write and executes it on a Machine (fictional as well).
- But what about a real computer?
  - ✓ Does it understand pseudo code?
  - ✓ What does it understand?
  - ✓ How do I actually program for it?

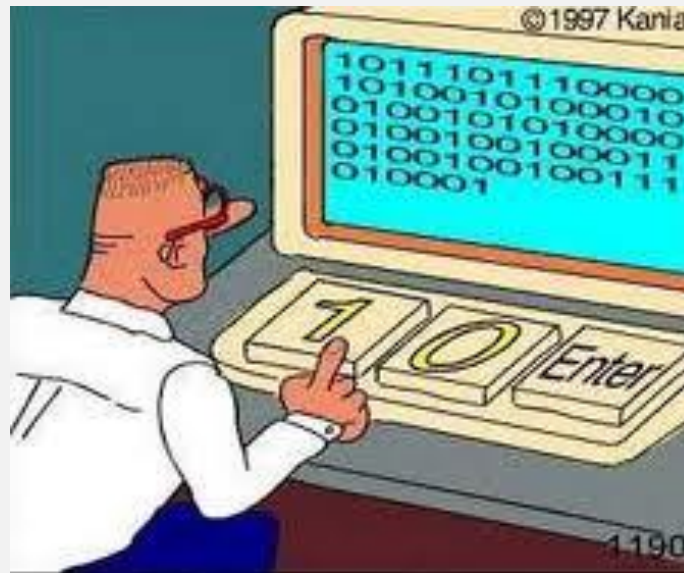
# MACHINE LANGUAGE (I)

- Each machine or machine family has its own language with repertoires of instructions and data types.
- Machine Language is the language of the specific architecture.
- The only language that the machine actually understands
- How it works?
  - ✓ It works directly over microprocessor instructions
  - ✓ Dependent on the machine architecture
    - ✓ No portability
  - ✓ Maximum efficiency



# MACHINE LANGUAGE (II)

This machine don't exist...



Working at machine language level consists of programming the hardware on a basic level.

# ASSEMBLY LANGUAGE

- Second language generation
  - ✓ Architecture dependent
  - ✓ Difficult programming although they are a little bit clearer than machine language, with names and symbols readable by humans.
- Can the machine understand it?
  - ✓ NO. Machines only understand their own language
- How does the machine work with it?
  - ✓ We translate from this language to the machine language, using a program called Assembler.

# PROGRAMMING OVER THE HW

- There are many types of hardware architectures and they are not handled in the same way.
- Programming directly on the hardware is a very complex task and in addition the same program must be done for as many hardwares as you want.

# HW ABSTRACTION

- In order to make programs compatible with several hardware architectures
  - ✓ The hardware manufacturers had to agree to make compatible models
  - ✓ Or we need to think another solution
- HW manufacturers didn't agree
  - ✓ Then we need to think another solution
- 😊

# HW ABSTRACTION

---

- The solution was to invent a new language that is independent from specific architectures and to provide a translator who does the work of converting what is written in that language to machine language.

# HW ABSTRACTION

---

If there is a translator for each architecture, I can write a program only once and translate it for each particular architecture.

# HIGH LEVEL LANGUAGE

- High level languages have 2 main goals
  - ✓ **Portability**
  - ✓ **Simpler programming**
- Does the computer understand it?
  - ✓ No
- What do we do for the computer to be able to understand?
  - ✓ We translate from high level Language to the Machine Language using a translation program, called translator.

# HIGH LEVEL LANGUAGE

This language is:

- More general
  - ✓ It's not specific for an architecture
    - ✓ Its generality is what moves it away from the specifics and allows portability
- Coherent
  - ✓ Each instruction must have its equivalence with one or more hardware instructions.



# QUICK QUESTIONS

- How was the translator programmed?
  - ✓ Assembly Language.
  - ✓ One translator per architecture
- Can there be different high level languages?
  - ✓ Yes.
  - ✓ If there is a translator for the language.

# HIGH LEVEL LANGUAGE

- There are a lot of high level languages
- For each of them, there is one or more translators
- Translators are classified into two types according to how they translate
  - ✓ Compilers
  - ✓ Interpreters

# COMPILERS AND INTERPRETERS (I)

- Compiler
  - ✓ Programs that receive something written in high level language and translate it completely to:
    - ✓ Machine language run by the machine.
    - ✓ Or an intermediate language that is not directly executable by the machine
  - ✓ Complete translation
- Generally speaking
  - ✓ Translates from a certain language to an inferior language.

# COMPILERS AND INTERPRETERS (II)

## **Interpreter:**

Programs that receive source code written in high level language, and then translate and execute instruction by instruction.

Partial translation.

# SOURCE CODE

- They all translate what is written in one language to another one.
- What is written in High Level language is generically called Source Code. This can be in:
  - ✓ Papers
  - ✓ Files
- **Source Program** = Set of Source Codes that conform a program.

# OBJECT CODE

- **Object Code** is the result of a compilation.
- **Executable Program:** It is a file that contains the program we want to execute.

# IN SUMMARY

- Higher level languages are more understandable by man.
- Lower level are more understandable by machines.
- Source Code Translation
  - ✓ Assembly Language.
    - ✓ Compiler: Assembler.
      - ✓ Source Code to Object Code (Machine).
  - ✓ High Level Languages
    - ✓ Compilers
      - ✓ Source Code to Object Code
    - ✓ Interpreter
      - ✓ Interprets one by one Source Code instructions

# COMPILATION, OS AND ARCHITECTURES

- Let's suppose we write a program in "C" language.
- In order for this program to be executed by a machine we need to translate it to Machine Language.
- So we have to have a compiler for that machine, right?.
  - ✓ It sounds right, but it's not how it works.



# OPERATIVE SYSTEM (OS)

- Before OS, everything was translated to Machine Code.
  - ✓ We had to load the program in a special way.
    - ✓ Using input and output routines (BIOS)
    - ✓ A lot of knowledge was required
  - ✓ Programs used to use resources anyway they wanted to
    - ✓ Programs could write any disk sector.
  - ✓ There was absolutely no control.

# OPERATIVE SYSTEM (OS)

- OS allows to
  - ✓ Control and coordinate the use of the hardware resources, so there are no issues there.
  - ✓ For programmers
    - ✓ We interact with OS to obtain HW resources.
  - ✓ For users
    - ✓ Turn on the computer and run the program with a “double click”

# OPERATIVE SYSTEM (OS)

- Between HW and Compiler there is now an OS.
  - ✓ Every part of our program that handles hardware resources is delegated to the OS to do it for us.
    - ✓ It is translated to Machine Code invoking a specific OS routine to operate with the hardware

# OPERATIVE SYSTEM (OS)

---

- Now we have Compilers for each OS
  - ✓ Not for specific architectures

# QUICK QUESTION

- Can we program for a specific architecture?
  - ✓ Yes, but you won't be able to run it on any machine with OS on it.
  - ✓ We have to hack the OS to use the resources.

# QUICK QUESTION

- Does an operating system run on different types of hardware automatically?
  - ✓ Of course not.
  - ✓ Each OS is built for a specific architecture.

# COMPILATION UNDER AN OS

- The result of compiling a high-level program is a file with an OS recognizable format containing Machine Code.
- The file is:
  - ✓ Dependent on HW
  - ✓ Dependent on OS
    - ✓ The uses OS native routines
- Program examples are:
  - ✓ .exe for Windows
  - ✓ .sh, .bin for Linux (some examples)
  - ✓ .app for Mac OS

# INTERPRETERS

- The story changes a little bit, since the interpretation does not involve the complete conversion of a Source program to an Object program.
- The conversion is done at the time of executing the program.
  - ✓ In Compilation we generate an Object Code that can be used only for a certain OS and Architecture.
  - ✓ A program that is interpreted can be run on any OS that has a compatible interpreter installed without a Compilation.
    - ✓ Flash Media Player
    - ✓ **Java**



# QUICK QUESTION

- Just as compiling we generate Object Code from Source Code, can you perform the reverse operation?
  - ✓ **Yes**
    - ✓ For the Assembly Language it is called disassembly
    - ✓ For any high-level language it is called decompile.
    - ✓ We went from a lower level to a higher level.
    - ✓ There are legal aspects to consider
      - ✓ Source Codes may be patented and an attempt to obtain a Source Code by these means may result in a claim

# QUICK QUESTION

- How was the Assembler programmed?
  - ✓ Machine Language
- And High-Level Compilers and Interpreters?
  - ✓ Assembly Language
  - ✓ Other high-level languages

THE END

---

**QUESTIONS?**