University for Information Science and Technology
Ohrid, Republic of Macedonia

An introduction to algorithms and pseudo-codes

Matthieu PUIGT
matthieu.puigt@uist.edu.mk

January 14, 2010

Foreword

A computer is like an Old Testament god, with a lot of rules and no mercy.

Joseph Campbell

These notes correspond to the first part of a subject called "Programming 1" which is given to first-year students of the University for Information Science and Technology (Ohrid, Republic of Macedonia) but it can be useful[1] to anyone who would like to know how to make algorithms (this is the first step before programming in a given language).

Since I did not find a clear and simple English-written book about this subject[2], this document proposes an introduction to algorithms and pseudo-codes. It is mainly inspired – and even almost just translated! – from the excellent French-written document **Algorithmique et programmation pour non-matheux – Cours complet avec exercices, corrigés et citations philosophiques** (`http://www.pise.info/algo/index.htm`), written by **Christophe DARMANGEAT** (Contact: `christophe.darmangeat@paris7.jussieu.fr`) that I thank for allowing me to translate his work and to put this translation online.

This document is a first version. The author thanks in advance anybody for finding errors or mistakes or for any comment:

`matthieu.puigt@uist.edu.mk`

[1] Well, I hope so.

[2] If you have some good references, please tell me: `matthieu.puigt@uist.edu.mk`

Contents

Chapter 1

Introduction to algorithms

1.1 What is an algorithm?

The word *algorithm* has an arabic origin, such as *Algebra*. But what is it? Well, have you ever opened a cookbook and been able to prepare a lunch according to its receipts? Have you ever read (or decrypted) the userguide of your cell phone, sometimes written in a foreign language, and then be able to use it? Yes? Congratulations, you have *executed* an algorithm.

Have you ever given indications to a tourist about the road to take for going somewhere? Better, have you help someone to find something, only thanks to a phone? Yes? So you *made* algorithms (and have been executed them).

Definition 1 *An algorithm is a series of instructions which yields to a given result when it has been correctly executed.*

If the algorithm is correct, then the obtained result will be the expected one but otherwise, it will have a random behaviour (and your tourist will go I do not know where) ...

However, since an algorithm is a series of instructions to solve a problem, why not giving a lonely instruction: "Solve this problem"? Unfortunately, it would be too easy. And, if we take the example of a lost tourist who asks for his road, you cannot say "Solve your problem yourself!". Indeed, if he asks this question, it is because he does not know the road! It is the same for computers: lot of people think that they are "stupid". In fact, they are just able to do understandable instructions (from their point of view). This implies that algorithms must only contain such instructions (and "Do it yourself" is not one).

1.2 Do I need to be good in Maths for making algorithms?

To this question, I say "No". Indeed, do we need to be good in maths for explaining to someone how to go somewhere? However, making algorithms requires some qualities provided by Maths:

- you need *intuition* in order to give the good series of instructions. But this intuition can be learned: in fact, the experience obtained after a considerable amount of work provides it.
- you need to be *methodical* and *rigorous*. Indeed, each algorithm must be *systematically* verified: thanks to a paper and a pen, **you must run your algorithm as a machine, step by step. This is the pledge of your progress.**

1.3 DNA and algorithms

DNA is an algorithm which is the foundation of Life: it is only composed of 4 elements but depending their number and their position, you can obtain an elephant or a bee.

It is the same for algorithms: computers are only able to understand 4 classes of *instructions*:

1. the affectation of variables,
2. reading / writing,
3. the tests,
4. and the loops.

An algorithm is only a combination of these 4 basic instructions. It can be short (a few instructions) or long (several thousands of lines) but its length is not an indication of its complexity: some algorithms are long and are easy to understand while some short algorithms are really complex!

1.4 Algorithms and programming

Why do I need to learn how to make algorithms if I want to learn programming? Why do I need a special language, distinct from programming languages that are understood by computers? Because an algorithm solves a given problem, *independently* of the specificities of a chosen language. As an example, when I am writing some lecture notes, I first think to the ideas of these notes, before writing it, in English or French, depending the tongue of my students. It is exactly the same with programs: algorithms provide the logical structure while programming languages provide the syntax of the program.

As a consequence, learning separately algorithms and programming is better because you may be able to distinguish the logical structure of a program from its syntax.

Lastly, a generation of programmers did not make this difference when they learned programming. Most of them are self-educated and do not clearly make the difference between what corresponds to a concept shared by all programming languages and the syntax of a language. As a consequence, even if their codes are good, they are quite unreadable. So let us learn the rules of programming!

1.5 How to write an algorithm?

Historically, several kinds of notations have been used for representing an algorithm. In particular, one graphical representation called *flowchart* has been often used in the past. A flowchart is a common type of diagram, that represents an algorithm, showing the steps as boxes of various kinds, and their order by connecting these with arrows. However, even if it looks simple to write algorithms by this way, it is not used now because:

1. When the algorithm is long, this representation is not easy to use.
2. It makes think with a particular way of programming called *unstructured programming* that we do not want to use.

That's why we generally use a series of conventions called *pseudo-code*. A pseudo-code looks like a real programming language, except that we have removed most of the syntax problems.

This pseudo code can slightly differ, depending your teacher or the book you are reading. I have my own notations and I live very well with them. I hope it will be the same for you!

1.6 How to execute an algorithm?

This is a recurrent question: how is it possible to execute an algorithm without computer? For a lot of people, computers are magic machines which are able to do things that nobody is able to do, even Chuck Norris! Of course, this is wrong: computers only do a list of *basic operations* that anybody is able to do. However, its main advantage is that it does it *faster* than any human (Chuck Norris included).

One easy way for executing an algorithm consists in making tables. Each column contains one of the following informations:

- the different values of one *variable* during the program,
- the values of a *condition* (true or false) in a *test* or a *loop*,
- and lastly, what appears on screen.

When an operation is done by the program, we change the value inside the corresponding column and the final values inside each column give you the states of each variable and condition at the end of the program.

When you execute an algorithm, you must be as **the Terminator** © of James Cameron: it never asks questions, it never tries to interpret orders, and it executes instructions exactly as they were programmed. **Never forget to be as a Terminator when you execute an algorithm. Do not try to interpret what the programmer wanted to do but execute what he exactly wrote.** In particular, when you will test your own algorithms, executing them by this way will **make you see your mistakes and progress.**

Chapter 2

Variables

2.1 What's for?

In a program, we will always need to store data. They can be found in your hard disk, written by the user, etc. They also can be partial of definite results obtained by the program. When we need to store an information during an execution of a program, we use a *variable*. These variables can be numbers, text, etc. As an image, a variable is a box that the program (the computer) finds thanks to a label (a name). If we want to see what is inside the box, we just have to look for the label.

In a real computer, there is a memory space, located by a binary address (e.g. 10010010). If we would program in a language that the computer directly understands, we should design our data by binary numbers. It would be the worst of the Worlds, wouldn't it?

> **Bad news:** these programs exist! They are called *assembly programming languages*.
>
> **Good news:** other languages also exist! More sophisticated languages (e.g. C/C++, Fortran, Matlab) do this difficult work for the programmer who only codes a series of instructions that will be translated in the computer language thanks to a *compiler* or an *interpreter*.

2.2 Declaration of a variable

When you have a new born child, the first thing you have to do consists in declaring his birth to authorities. It is the same with algorithms: the first thing to do before being able to use a variable consists in creating a box and associating a label to it. It is always done at the beginning of the algorithm, before instructions. The name of the variable (the label of the box) must obey to some rules of the programming language. However, an absolute rule says that the name of a variable:

- contains letters and numbers,
- does not use punctuation signs, and in particular spaces,
- begins by a letter.

The maximum length of the name of a variable depends on the programming language.

When you declare the birth of a child, you must also give a lot of informations about him (sex, precise date of birth, etc). In a program, it is the same: when we declare a variable, we not only create a box, but also explain what we are going to put inside. We are now going to see the list of types we can put in a variable.

2.2.1 Classical numerical types

Let us begin by a situation we often meet: the one of a variable which will receive numbers.

If we allocate a byte for coding a number, we will only be able to code $2^8 = 256$ different values. This implies that we can e.g. code integers from 1 to 256, or from 0 to 255, or from -127 to 128... If we reserve two bytes, we can use 65536 values, with three byte, 16777216, etc. And here is a new problem: must we also code decimal numbers? Negative numbers?

Well, the chosen way of coding (i.e. the type of variable) for a number will determine:

- the minimum and maximum values of the numbers which can be stored in the variable,
- the accuracy of these numbers (in the case of decimal numbers).

All the languages offer a set of numerical types, whose detail can slightly change from a language to other. But we generally find the ranges below:

Numerical type	Range
Byte	From 0 to 255
Simple integer	from -32768 to 32767
Long integer	From -2147483648 to 2147483647
Simple real	From $-3 * 10^{38}$ to $-1.40 * 10^{-45}$ for negative values From $1.40 * 10^{-45}$ to $3 * 10^{38}$ for positive values
Double real	From $-1.79 * 10^{308}$ to $-4.94 * 10^{-324}$ for negative values From $4.94 * 10^{-324}$ to $1.79 * 10^{308}$ for positive values

Why not declaring all numerical variables as double real? Indeed, we would be sure that we would not have any problem! Well, programmers do not want to waste memory because of the principle of *economy of means*: a good algorithm not only works but also saves all the memory which is possible. For some high-sized programs, the overuse of oversized variables provides some slowdowns or even crashes during execution. So we must take good habits now.

In a pseudo-code, we will not take care about the sub-kind of numerical variables. We will just say if they are numbers, while keeping in mind that in a real language, we will need to be more accurate.

In pseudo-code, the declaration of variables will read:

Variable g **as Integer;**
Or for example
Variables Luke, R2D2, Yoda **as Numbers;**

2.2.2 Other numerical types

Some programming languages allow other numerical types:

- the *money* type, which only allows two decimal places,
- The *date* type (dd/mm/yyyy).

We won't use these kinds of parameters in this document but now, you know they exist.

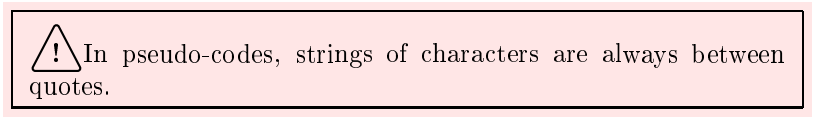
2.2.3 Alphanumerical type

Fortunately, the boxes that are variables can contain many things different from numbers. Without this possibility, we would have some difficulties for storing a family name for example.

We thus have an *alphanumerical* type (also called *character* type or *string* type).

In such variables, we store characters which can be letters, punctuation signs, spaces or digits. The maximal number of characters which can be stored in a lonely string variable depends on the used language.

A set of characters (which can also be a group of one or zero characters) is usually called a *string of characters*.



Why? In order to avoid both principal origins of possible mistakes:

- The confusion between numbers and a string of digits. For example, 123 can represent the number 123 (one hundred and twenty three) but also the series of digits 1, 2, and 3. And this is completely different! With the first one, we can make computations while it is impossible with the second one. As a consequence, quotes allow us to remove all ambiguity: 123 is the number one hundred and twenty three while "123" represents the series of digits 1, 2, and 3.
- But this is not the worst... Another mistake consists in making a confusion between the name of a variable and its value. This means that we won't make the difference between the label of the box and what it contains. We will talk about it later.

2.2.4 Boolean type

The last type of variable is called *Boolean*: we only store logical values TRUE or FALSE.

We can represent these abstract notions by what we want: English (TRUE and FALSE), French (VRAI and FAUX), numbers (1 and 0), etc. It does not matter. What is important is to understand that a boolean type is really economic in terms of memory costs since we only need one bit for storing a binary information.

The boolean type is usually neglected by programmers, inappropriately.
OK, it is not necessary and we can write almost any programs without using it. However, if it is accessible to any programmer, there is a good reason. Using boolean variables yields **readability** to your algorithms: it can help you a lot, but also people who will read or correct the algorithm.
So now it is sure, in Algorithms like in the languages, there are many ways to say the same thing. We will see later different style variations about the same solution.

2.3 Affectation instruction

2.3.1 Syntax and Meaning

Here is the time for our first real algorithmic manipulation! In fact, the variable (the box) is not a hard tool to manipulate. Unlike the Swiss army knife, you cannot do thirty-six thousands things with variables, but one and only one.

We are only able to *assign it*, i.e. *assign a value*. If we still continue the already employed metaphor, we fulfill the box.

Definition 2 *In pseudo-code, the assignment statement is denoted with the symbol* $\leftarrow$.

Thus:

```
Luke ← 12;
```

assigns the value 12 to the variable Luke.

This necessarily implies that Luke is a variable of numeric type. If it was defined in another type, we must understand that this instruction will cause an error. It is almost as if we were giving an order to someone with incompatible verb and complement, such as “Peel the pan”. Even with the best intentions, the housewife could simply put his dubiously task while reading this sentence. Then, a computer...

One can however assign without any problem the value of a variable to another variable. For example:

```
Luke ← Force;
```

means that the value of Luke is now that of Force.

Note that this instruction does not change the value of *Force*: **an assignment statement does not change what is on the right of the arrow.**

```
Luke ← Force+4;
```

If Force was 23, now the value of Luke is 27. As previously, Force is still 23.

```
Luke ← Luke+1;
```

If Luke was 27, it is now 28. The value of Luke is modified since it is also placed to the left of the arrow.

Now, let us go back to the role of quotes in strings and in particular, to the second confusion reported above. Now compare the following two algorithms:

Example 1:
Begin
Luke ← "DarthVader";
Yoda ← "Luke";
End

Example 2:
Begin
Luke ← "DarthVader";
Yoda ← Luke;
End

The only difference between the two algorithms lies in the presence or absence of quotation marks in the second assignment. And we see that this changes everything!

In the first example, what we affect to the variable Luke is the string of characters D–a–r–t–h–V–a–d–e–r. And at the end of the algorithm, the variable Yoda contains the value "Luke". On the contrary, in the second example, Luke does not contain any quote mark. Here, since it is written without quotation marks, Luke is the name of the variable. This means: "affetcs to variable Yoda the value of the variable Luke ". At the end of the second algorithm, the value of the variable Yoda is thus "DarthVader". Here, forgetting the quotation marks still provides a result but which is very different from the result produce by Algorithm n^o 1.

Because it is a common case, note that forgetting the quotation marks can provide error if what is placed to the right of the affectation sign does not correspond to any previously declared and affected variable.

This is a simple illustration but it resumes all the problems that occur when we forget the rules of quotation marks in strings of characters.

2.3.2 Order of the instructions

It is obvious that the position of the written instructions in the algorithm will play a key role in the outcome. Consider the two following algorithms:

Example 1:
Variable A **as Number;**
Begin
$A \leftarrow 34$;
$A \leftarrow 12$;
End

Example 2:
Variable A **as Number;**
Begin
$A \leftarrow 12$;
$A \leftarrow 34$;
End

It is clear that in the first case the final value of A is 12, the other is 34.

It is also clear that this should not surprise us. When indicating a route to someone, say "go straight for 1 km, then turn right" does not send people to the same place as when we say "turn right then straight ahead for 1 km".

Finally it is also clear that if one puts aside their pedagogical virtue, the two above algorithms are perfectly idiot, because they contain some inconsistency. There is no interest to assign a variable to assign different right after. In this case, we might obtain as well the same result by simply writing:

Example 1:
Variable A **as Number;**
Begin
$A \leftarrow 12$;
End

Example 2:
Variable A **as Number;**
Begin
$A \leftarrow 34$;
End

2.4 Expressions and operators

If we think about it, we find in an assignment statement:

- Left of the arrow, a variable name, and only that. In this algorithms' world which is full of doubts, this is one of the few golden rules that work every time: if you see on the left of the arrow other thing than a name of variable, you can be 100% sure that this is a mistake.
- Right of the arrow, is called an *expression*. This is another word that is misleading: in fact, this word is common in all-day-life, where it has many meanings. But in computing, the term expression refers to does one thing and that is one more thing very clear:

  Definition 3 *An expression is a set of values, linked by* operators *and equivalent to a single value.*

If this expression is quite ununderstandable, think about it during a few minutes and you will see that it correspond to simple things. For example, the following numerical expressions

```
7
5+4
123-45+844
Luke-12+5-R2D2
```

... are all valid if Luke and R2D2 have been declared as numbers. Because if it is not the case, the fourth expression does not have any sense. In the above examples, I only used the sum (+) and the substraction (-) operators.

Let us go back to affectation. Another condition (in addition to both previous ones) for an affectation instruction to be valid is that:

- Expression located to the right of the arrow belongs to the same class than the variable located to the left of the arrow. It is completely logival: you cannot properly store tools in a shopping bag or vegetables in a toolkit... except to cause a catastrophic outcome!

If at least one of the above three points is not respected, the computer will not be able to execute the affectation and will provide an error message. We are now going to define what we called an *operator*.

Definition 4 *An operator is a sign which links two values, in order to produce a result.*

Depending the type of the variables, the operators can differ. Let us see them.

2.4.1 Numerical operators

They are the four arithmetical operations that all of you know:

- the sum (+),
- the substraction (-),
- the product (*),
- the division (/).

Let us mention that the symbol $\wedge$ means power. 45 square will be written $45 \wedge 2$.

Lastly, we can use brackets, with the same rules than in Mathematics.

2.4.2 Alphanumerical operator &

This operator allows the *concatenation* of two strings of characters. For example:

```
Variables A, B, C  as Characters;
Begin
A ← "Dark";
B ← "side";
C ← A & B;
End
```

The value of C at the end of the algorithms is "Darkside".

2.4.3 Logical (or Boolean) operators

It consists in the connectives that we saw in Discrete Mathematics: AND, OR, NOT and XOR (exclusive OR, that is one boolean or other but not both). We will see them later.

2.5 Two last remarks

Now that we are familiar with variables, I draw your attention to the deceptive similarity of vocabulary between mathematics and computer science. In mathematics, a “variable” is usually unknown, which includes an unspecified number of values. When I write:

$$y = 3x + 2$$

the number of "variables" x and y which satisfy this equation is an infinity. When I write

$$ax^2 + bx + c = 0$$

the "variable" x design the solutions of this equation, that is zero, one or two values at the same time...

> ⚠ In Programming, at a given time, a variable exactly has one and only one value.

Strictly speaking, it can have no value at all (once it was declared, and since it was not affected. One should note that in some languages, the not yet assigned variables are automatically set to zero). But what is important is that this value precisely varies only when it is the subject of an assignment statement.

The second point concerns the sign of the assignment. In algorithms, as we have seen is the sign $\leftarrow$. But in practice, almost all languages use the equal sign. And there, for starters, the confusion with Mathematics is easy. In maths, $A = B$ and $B = A$ are two strictly equivalent propositions. But absolutely not with a computer, since this amounts to write A $\leftarrow$ B and B $\leftarrow$ A which are two very different things. Similarly, $A = A + 1$, which is an equation without solution in Mathematics, is a completely authorized action in Programming (and also extremely common). So beware !!! The best vaccination against this confusion is to properly use the $\leftarrow$ sign in pseudo-code. Once acquired good reflexes with this sign, you will not have any trouble passing to the $=$ of programming languages.

Chapter 3

Reading and Writing

3.1 What's that?

Let us imagine that we made a program which computes the square of a number, say 12. A solution should read:

```
Variable A as Number;
Begin
A ← 12 ∧ 2;
End
```

OK, we computed the square of 12. But if we want to computer the square of any different number, we must rewrite the program...

Moreover, even if the result has been computed by the machine, it is stored somewhere in the memory of the computer and the user will never know what is the square of 12...

That's why there are fortunately instructions which allow the computer to dialog with the user.

In a direction, these instructions allow the user to enter some values with the keyboard to be used by the program. This operation is called *Reading*.

In the other direction, these instructions allow the program to communicate some values to the user by printing them on screen. This operation is called *Writing*.

We should be surprised that programmers called these operations as they did. Indeed, when the user types on keyboard, we call it reading. On the contrary, when it reads what is printed on screen, we call it writing. In fact, we must change of point of view: when a program is run, it is the machine which is working, not the user! When a machine reads a value, it implies that a man will enter it. On the contrary, when we ask the machine to print a value on screen, it is done in order the user reads it. As a consequence, Reading and Writing instructions have been defined from the computer point of view. And everything becomes logical.

3.2 Reading/Writing instructions

It is really simple: when the user enters the (new) value of Luke, we will put:

Read Luke;


When a program meets a Read instruction, the execution stops, waiting for the value that the user will write.

Once the Enter button has been pressed, the execution goes on. In the opposite, in order to write something on screen, it is also simple:

Write Yoda;

Before writing a variable, I suggest you to always write some sentences on screen, in order to tell the user that he will need to enter something (otherwise, the poor user will waste all his time wondering what the computer is doing!)

Write "What is your name?";
Read FamilyName;

Reading and Writing are instructions which do not present some particular difficulties, once we have understood the problem of the direction of the talk (man → machine, or machine ← man).

Chapter 4

Tests

We previously saw that an algorithm is the combination of 4 structures. We already saw two of them, let us see the third one.

4.1 What is it?

Let us use our example of "algorithm of the lost tourist". It should look like: *"Go straight. When arrived to the second cross-roads, take the second street to your left. You are there"*.

Now, in case of doubt, it should become: *"Go straight. When arrived to the first cross-roads, if there is no panel which forbids you that, go straight and take the first next street on your left. Otherwise, in the first cross-roads, take the street on the right, take the first on the left and the first on the left. Lastly, go straight"*.

This second algorithm is an improvement of the first one since it proposes two ways for solving the given problem. It assumes that the lost tourist will be able to analyze the situation ("Is there a pannel in the first cross-roads?") on order to execute the series of instructions.

Computers are also able to do it! We are going to talk to our computer as to a tourist, by giving it several series of instructions, depending which situation occurs. This logical structure is called *Test*.

4.2 Structure of a test

There are only two possible forms for a test: the first one is simple while the second one is slightly more complex.

```
If Boolean Then
  Instructions;
EndIf
```

```
If Boolean Then
  Instructions1;
Else
  Instructions2;
EndIf
```

This calls some explanations.

A *Boolean* is an *expression* which is TRUE or FALSE. It only can be (there is only two possibilities)

- a Boolean *variable* (or an expression),
- a *condition*.

We will see below what is a condition in Programming.

Let us note that the structure of a test is relatively clear. In its simplest form, when arrived to the first line (If ... Then), the machine studies the value of the Boolean. If its value is TRUE, then the machine executes the series of instructions. This series can be very short or very long, it is not important. However, in the case when the Boolean is false, the computer directly skips to the instructions located after EndIf.

In the case of a complete structure, it is slightly more complex. If the value of the Boolean is TRUE, and after the execution of the series of Instructions1, when it arrives to the word "Else", the machine skips to the first instruction after the word "EndIf". On the contrary, if the value of the Boolean is FALSE, the machine skips directly goes to the first line after "Else" (without executing any line of the series of Instructions1) and runs the series of Instructions2. In all these situations, the instructions located after EndIf will be run.

In fact, the simplified form correspond to the case when one of both "branches" of If is empty. Hence, instead of writing "Else do nothing", it is simpler to write nothing. Let a complete IF... with both empty branches is considered as a very big clumsiness for a programmer, even if it is not a programming mistake.

Written as a pseudo-code, the program of our above lost tourist should read:

```
Go straight until the next cross-roads;
If There is no panel Then
   Go straight;
   Turn left at the next cross-roads;
Else
   Turn right;
   Take the next street to your left;
   Take the next street to your left;
   Go straight;
EndIf
```

4.3 What is a condition?

Definition 5 *A condition is a comparison.*

This definition is essential! It means that a condition is composed of three elements:

- a value,
- *a comparison operator*,
- another value

These values can belong to any type (numerical, characters, etc). But if we want our comparison to have a sense, these values must belong to the same type!

Comparison operators are:

- equal to...
- different to...
- strictly lower than...
- strictly higher than...
- below or equal to...
- above or equal to...

The set of three elements which constitute a condition makes a *mathematical proposition* which can be TRUE or FALSE.

The following table shows the main notations (and in particular the ones we will use in this document) for expressing the above operators:

Operators	Chosen notation	Other classical notations
equal to	$=$	$==$
different to	$\neq$	$<>$, $\sim=$, $!=$
strictly lower	$<$	
strictly higher	$>$	
below or equal	$<=$	$\leq$, $=<$
above or equal	$>=$	$\geq$, $=>$

Please note that these operators can be used with characters. Indeed, characters are coded by machine thanks to numbers (ASCII code), according to the alphabetical order. The capital letters are always placed before the lower-case letters. We thus obtain:

"t" $<$ "w";	TRUE
"Daddy" $>$ "Mum";	FALSE
"daddy" $>$ "Mum";	TRUE

By formulating a condition in an algorithm, we must be really carefully with some all-day-life words or some mathematical notations which yield to programming non-senses. For example, let us consider the sentence "Luke is between 5 and 8". We should translate it by:

$$5 < \text{ Luke } < 8$$

However, this expression which has a sense in English and in Mathematics **does not mean nothing in Programming**. Indeed, it contains two comparison operators (and so one too many) and three values (here again one too many). We are going to see how to translate this condition properly.

4.4 Composed conditions

Some problems sometimes need to formulate some conditions which cannot be expressed under the simple form we saw above. For example, let us consider again the situation when "Luke is

included between 5 and 8". In fact, this sentence does not hide one but **two** conditions. Indeed, there are two conditions linked by what we call a logical operator (or logical connective in Discrete Mathematics): the word AND.

As we explained before, Programming allow us to use four logical operators: AND, OR, NOT, and XOR.

- AND has the same meaning in Programming than in all-day-life. If we want "Condition1 AND Condition2" to be TRUE, we need both Condition1 and Condition2 to be TRUE. In all other situations, "Condition1 AND Condition2" will be FALSE.

- In order that "Condition1 OR Condition2", we need that at least one of these conditions to be true.

- "Condition1 XOR Condition2" slightly differs from the previous situation. It is true if at least one of the conditions is true **but not both**. So if Condition1 and Condition2 are TRUE, then "Condition1 XOR Condition2" is FALSE while "Condition1 OR Condition2" is true!

- Lastly, NOT provides the negation of a condition. NOT(Condition1) is TRUE if Condition1 is FALSE. On the contrary, NOT(Condition1) is FALSE if Condition1 is TRUE.

Usually, we can test the truth value of a condition thanks to a *truth table*. A truth table only test the validity of a composed condition according to the truth value of each condition.

The above comments about operators can be resumed as a table:

C1	C2	C1 AND C2	C1 OR C2	C1 XOR C2	NOT C1
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE

The great joke of the programmer: It consists in writing in a test a **condition which will be always true or always false**. As a consequence, it implies that we can write several thousands of lines in a test that we know in advance that we will never execute them.
For example, in "IF Luke $<$ 10 AND Luke $>$ 15 Then...", it will be really hard to find such a number which is in the same time lower than 10 and higher than 15.

4.5 Nested tests

Graphically, it is really easy to show an IF structure as a rail switch. An IF thus opens two ways, corresponding to two different treatments. But there exist a lot of situations where two ways are not sufficient. For example, the program which must give the state of water with respect to its temperature must choose between three possible states (solid, liquid and gaseous).

A fist solution reads:

```
Variable Temp  as Integer;
Begin
Write "Enter the water temperature :";
Read Temp;
If Temp <= 0 Then
  Write "It's ice.";
EndIf
If (Temp > 0) AND (Temp < 100) Then
  Write "It's liquid.";
EndIf
If Temp >= 100 Then
  Write "It's steam.";
EndIf
End
```

You can see that it is a bit tedious. All the conditions look almost the same and force the machine to check three successive tests while all of them are about the same subject: water temperature (the value of the variable Temp). It should be more rational to *nest* the tests as:

```
Variable Temp  as Integer;
Begin
Write "Enter the water temperature :";
Read Temp;
If Temp <= 0 Then
  Write "It's ice.";
Else
  If Temp < 100 Then
    Write "It's liquid.";
  Else
    Write "It's steam.";
  EndIf
EndIf
End
```

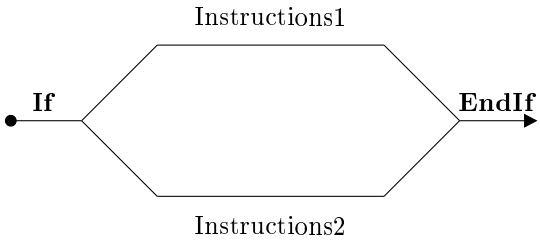
We really improved the first algorithm: instead of three conditions (with a composed one), we now only have two simple conditions. But also we especially saved some memory on the execution time of the computer. Indeed, if the temperature is below zero, it now write "It's ice" and **directly** skips to the end, without loosing time to check other possibilities (which are false by definition).

This second version is not only simpler to write and more readable, it is also more efficient.

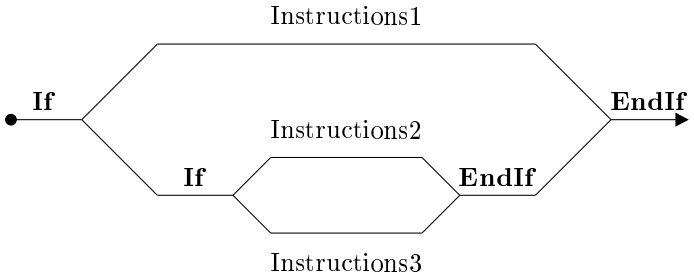
> Nested tests structures are an indispensable tool for the simplification and the optimization of algorithms.

4.6 Let's talk about rail switches

As we previously noticed, an IF structure can be compared to a rail switch. The main way is separated in two and the train can go in one or the other way. Both ways then meet again, when arrived to ENDIF. We can draw it like this:



But in some cases, we do not need two but three, four (or more!) ways. In the case of the water state, we needed three ways for our "train". Then, we did not have any choice and we used two nested tests. This structure can be drawn as:



Let us be clear: this structure is the only possible from a logical point of view (even if we can put the lowest instruction above and the highest one below). But from a Programming point of view, the pseudo-code admits another simplication. Thus, it is possible (but not mandatory) to write it like this:

```
Variable Temp  as Integer;
Begin
Write "Enter the water temperature :";
Read Temp;
If Temp <= 0 Then
   Write "It's ice.";
ElseIf Temp < 100 Then
   Write "It's liquid.";
Else
   Write "It's steam.";
EndIf
End
```

In the case of nested tests, ELSE and IF can be merged in an unique ElseIf. We consider that it is a unique test block, ended by a unique EndIf.

ElseIf allows to simulate rail switches with more than two ways. We thus can put ElseIf conditions ones after others in order to simulate a switch with as many ways as we want.

4.7 Boolean variable

Until now, in order to write our tests, we only used *conditions*. But you must recall that there exists a type of variable (Boolean variables) which can stock the TRUE and FALSE values. In

fact, we can enter some conditions in variables, and then test the value of these variables.

Let us consider another time the example of water. We should write it as:

```
Variable Temp as Integer;
Variables A, B as Booleans;
Begin
Write "Enter the water temperature :";
Read Temp;
A ← (Temp <= 0);
B ← (Temp < 100);
If A Then
   Write "It's ice.";
ElseIf B Then
   Write "It's liquid.";
Else
   Write "It's steam.";
EndIf
End
```

A priori, this technique is not of interest: we did not lighten the starting algorithm, adding two additional variables.

- But let us recall that a Boolean variable only need one bit in order to be stored. From this point of view, the weighting is not important.
- In some situations, especially when composed conditions are really heavy (with lots of AND and OR everywhere), this technique can help the programmer, making the algorithm more readable. Boolean variables can also be useful as *flags*. We will talk about this technique in Chapter 8.

Chapter 5

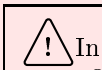
Still about Logic

5.1 Should there be an AND? Should we put an OR?

One remark to start: in the case of composed conditions, the brackets play a fundamental role.

```
Variables A, B, C, D, E  as Booleans;
Variable X  as Integer;
Begin
Read X;
A ← X>12;
B ← X>2;
B ← X<6;
D ← (A AND B) OR C;
E ← A AND (B OR C);
End
```

If X= 3, then we see that D is TRUE while E is FALSE.
However, if there are only AND or only OR, then the brackets do not play any role.



In a composed condition employing operators AND and operators OR, the presence of brackets has an influence on the result, as in the case of a numerical expression involving multiplications and additions.

This leads to another property of AND and OR, much more interesting.
Spontaneously, we often think that AND and OR are mutually exclusive in the sense when a given problem is expressed thanks to an AND or thanks to an OR. Yet, this is not so obvious.
When should you open the window of the room? Only if conditions require it, i.e.

```
If it is too hot AND it does not rains Then
   Open window;
Else
   Close window;
EndIf
```

But this small rule should be formulated as:

If it is not too hot **OR** it rains **Then**
  Close window;
Else
  Open window;
EndIf

Both formulations are strictly equivalent. Which brings us to the following conclusion:


Any test structure requiring a composed condition using the operator AND can be equivalently expressed thanks to the operator OR, and vice versa.

This is less surprising than it seems at first. For convincing yourself, get an eye on truth tables, and you will notice the symmetry between the one with AND and the other with OR. In both tables, there are three out of four cases which yield to a result, and one out of four which yield to the inverse result. So there is nothing surprising that a situation which is expressed with one of the tables (one of the logical operators) should also be expressed with the other table (the other logical operator). The whole trick is to know to properly perform this transition.

Of course, one cannot purely and simply replace AND by OR; it should be too easy. The equivalence rule is the following (one can check it on the example of the window):

If A **AND** B **Then**
  Instructions1;
Else
  Instructions2;
EndIf

Equivalent to:

If NOT A **OR NOT** B **Then**
  Instructions2;
Else
  Instructions1;
EndIf

This rule is called the de Morgan law, from the name of the English mathematician who formulated it.

5.2 Beyond the logic: a question of style

The goal of this somewhat provocative title (but justified) is to put on light a fundamental fact in programming, fact that some previous remarks have probably make you suspect: there is never one right way to treat alternative structures. And more generally, there is never one right way to treat a problem. Between the different possibilities, which are not better the ones than the others, the choice is a matter of *style*.

That's why, with some habit, one recognize the style of a programmer as surely as if it was a literary style.

Let us still consider our comparison operators AND and OR. In fact, we see that we could totally do without them! For example, taking the example of the window of the room:

```
If it is too hot AND it does not rains Then
  Open window;
Else
  Close window;
EndIf
```

is exactly equivalent to:

```
If it is too hot Then
  If it does not rains Then
    Open window;
  Else
    Close window;
  EndIf
  Close window;
EndIf
```

In this last formulation, we no longer use a composed condition made (but at the price of additional nested test).

And like everything which can be expressed by an AND can also be expressed by an OR, we conclude that OR can also be replaced by an additional nested test. We may thus state this general style rule:

In a complex alternative structure, composed conditions, nested test structures and the use of Boolean variables open the possibility of different style choices. Heavier conditions lighten test structures and the number of necessary Booleans; using additional Booleans lighten conditions and test structures, and so on.

If you understood what was above, then you know everything there is to know on test in order to handle any situation.

Unfortunately, we're not quite at the end of our troubles, it remains a last logical structure to consider, not least...

Chapter 6

Loops

Here we are! The fourth and last structure: the *loop*. If you want to be brilliant in a cocktail party, you can also speak about *repetitive structures* or *iterative structures*. Here, because we do not need to impress someone, we will simply talk about loops.

Loops are generally a difficult point for novice programmers. Indeed, while it is usually quite simple to understand how loops work, it is usually quite long to obtain the reflex providing their clever use in a given problem.

We can say that in fact, loops are the only true logical structures which caracterize a loop. If you e.g. use Excel, you may use things which are equivalent to variables (cells, formula) and to tests (the IF function...). But loops do not have any equivalence. It only exists in programming languages.

Using loops is the main difference between the programmer and the (even aware) user.

So for your future difficulties, there are only three solutions: rigor, patience and still rigor!

6.1 What is it for?

Let us consider a keyboard input (reading instruction), where the program e.g. asks a question to which the user must answer by Y (yes) or N (no). But earlier or later, an asleep or funny user risks to enter an unexpected answer. Then, the program can stop, because of an execution error (because the type of response does not correspond to the expected type of variable) or by a functional error (it runs until the end but produces some wild results).

As a consequence, in each a bit serious program, we put what we call an *input control*, in order to check if the data entered with keyboard correspond to those which are expected.

A first idea should consist in using an IF structure. Let us see how it reads:

```
Variable Resp  as Character;
Begin
Write "Do you want coffee? (Y/N)";
Read Resp;
If (Resp ≠ "Y") AND (Resp ≠ "N") Then
  Write "Error! Retry.";
  Read Resp;
EndIf
End
```

Great! It works... if the user only fails once and types a correct value when he is asked the second time. If we want to improve our program, we must add another IF. And so on, we can add hundreds of IF and write an unreadable program, we will never be sure that a obstinate user won't make the program fail!

The solution which consists in putting lots of IF is thus a dead-end. The only solution consists in using a *loop structure* which looks like:

```
While Boolean
  ...
  Instructions
  ...
EndWhile
```

The principle is simple: a program arrives on a While line. It checks the value of the Boolean (which can be a Boolean variable or more frequently a condition). If this value is TRUE, then the program runs all the instructions which follow until it meets the EndWhile line. It then goes back to the While line and re-checks the value of the Boolean, and so on. The round turns off when the Boolean takes the FALSE value. In this case, the program skips instructions until the line after EndWhile.

Let us illustrate it with our input control problem. A first approximation of the solution consists in writing:

```
Variable Resp as Character;
Begin
Write "Do you want coffee? (Y/N)";
While (Resp ≠ "Y") AND (Resp ≠ "N")
  Read Resp;
EndWhile
End
```

Here, we have the skeleton of a correct algorithm. However, it still needs some extra information in order to be executable.

Its main problem is that it provides an error in each run. Indeed, the Boolean expression which is located after While asks the value of the variable Resp. Unfortunately, this variable, even if it has been declared, has not been affected before the entrance of the loop. We thus test a variable which has not any value, which provides an error and the immediate stop of the execution. In order to avoid this situation, we only have one choice: we need the variable Resp to be already affected before we get into the loop. For that, we can do a first read of Resp before the loop. In this case, it will only be used in case of mistake by the user. The algorithm thus reads:

```
Variable Resp as Character;
Begin
Write "Do you want coffee? (Y/N)";
Read Resp;
While (Resp ≠ "Y") AND (Resp ≠ "N")
  Read Resp;
EndWhile
End
```

Another solution which is often used does not consist in reading but in arbitrarily affecting the variable before the loop. Arbitrarily? Not completely, since this affectation must force the entrance in the loop. This affectation must thus be done so that the Boolean will be set to TRUE and we will go into the loop for a first round. In our example, we can affect Resp with any value, except "Y" and "N". Because in this case, the execution would skip the loop and Resp would never be read with keyboard. A solution thus reads:

```
Variable Resp as Character;
Begin
Write "Do you want coffee? (Y/N)";
Resp ← "X";
While (Resp ≠ "Y") AND (Resp ≠ "N")
  Read Resp;
EndWhile
End
```

You must know this way because it is often used.

You must note that both solutions (first read of Resp outside the loop or affectation of Resp) both make the algorithm efficient, but show a quite important difference in their logical structure.

Indeed, if we choose to first read the value of Resp, the following loop is only used for checking if the user made a mistake. If the user does not make a mistake, the algorithm will skip the loop.

On the contrary, with the second solution (with a first affectation of Resp), the entrance in the loop is forced and its execution is mandatory at least once. For the user point of view, there is not difference. But for the programmer point of view, it is important to understand that the pathways of instructions won't be the same in one case and another.

In order to finish, let us remark that we could improve our solutions by adding extra sentences printed on screen. A possible improvement of the first solution thus reads:

```
Variable Resp as Character;
Begin
Write "Do you want coffee? (Y/N)";
Read Resp;
While (Resp ≠ "Y") AND (Resp ≠ "N")
  Write "You must answer by Y or N. Retry.";
  Read Resp;
EndWhile
Write "Correct input.";
End
```

For the second solution, it should be something like:

```
Variable Resp as Character;
Begin
Write "Do you want coffee? (Y/N)";
Resp ← "X";
While (Resp ≠ "Y") AND (Resp ≠ "N")
  Read Resp;
  If (Resp ≠ "Y") AND (Resp ≠ "N") Then
    Write "You must answer by Y or N. Retry.";
  EndIf
EndWhile
End
```

The great joke of the programmer: It consists in writing in a loop for which the value of the Boolean is always false. As a consequence, it implies that we can write several thousands of lines in the loop that we know in advance that we will never execute them.
More funny, one can find some programs where the value of the Boolean is always true (and never becomes false). Then, the program will never stop! We call this situation **Infinite loop** and in this situation, we only have a solution: killing the execution of the program.

6.2 Looping while counting or counting in loops

Knowing *in advance the number of rounds* in a loop is an often met situation. However, the WHILE loop does not *a priori* know this number (since the number of rounds only depends on the value of a Boolean).

That's why another loop structure has been proposed. The following algorithm

```
Variable Toto as Integer;
Begin
Toto ← 0;
While Toto < 15
  Toto ← Toto +1;
  Write "Round number :", Toto;
EndWhile
End
```

can also be written as:

```
Variable Toto as Integer;
Begin
For Toto ← 1 to 15
  Write "Round number :", Toto;
Next Toto
End
```

Let us be clear: **the "FOR ... NEXT" is not mandatory.** Indeed, we can program all the loops only with a "WHILE" structure. The "FOR" structure is only provided in order to help the programmer, avoiding him to manage the value of the variable which is used as a counter (we also speak of *increment*).

Said differently, the "FOR ... NEXT" structure is a special case of "WHILE": the one when the programmer can count in advance the number of necessary rounds inside the loop.

Let us note that in the "FOR ... NEXT" structure, the statement of the counter can be updated as the programmer wants. In most of the cases, we need a variable to be increased of 1 in each round of the loop. In this case, we do not add nothing to the "FOR" instruction. By default, the computer understands that it will need to add one to this increment after each round, beginning by the first value and finishing by the second one.

But if you want a special progress, e.g. 2 by 2 or 3 by 3, or -1 by -1, or -10 by -10, it is not a problem: you just need to tell it in your instruction "FOR" by adding the word "Step" and the value of this step.

We now can give the general form of a "FOR ... NEXT" structure. Its general syntax is:

```
For Counter ← Initial to Final Step StepValue
  ...
  Instructions;
  ...
Next Counter
```

Of course, when we use a negative step, the initial value of the counter must be **higher** than its finishing value if we want the loop to be executed. In the contrary, we will simply write a loop in which the program will never go into. Please note that this is the second and last difference with the "WHILE" structure. For example, if we write this kind of "FOR" loop:

```
For i ← 1 to 10 Step −1
  Write "Luke, I am your father."
Next i
```

As we wrote above, the program will never go inside the loop. However, if we write this code it with a "WHILE" structure, it reads:

```
i ← 1;
While i =< 10
  Write "Luke, I am your father."
  i ← i−1;
EndWhile
```

And in this case, the value of the Boolean is true and we thus go inside the loop. However, since the value of i is decreasing in each round, we will never leave the loop. In other words, we here meet an **infinite loop!**

The WHILE loops are used in the situations when we must process a treatment for which we do not know in advance the quantity. For example:

- the input control,
- the rounds in a game (while the game is not over, we retry)
- the read of recordings for which the size of the file is unknown.

The FOR structures are used in situations when we must process a treatment for which programmer knows in advance the quantity. We will see in the following chapters series of elements called arrays (see Chapters 7 and 9) and string of characters (See Chapter 10). Depending the situation, the study of the elements of these series will be done by a FOR or a WHILE loop. This will only depends if the number of elements to be studied (i.e. the number of necessary rounds in the loop) can be previously computed by the programmer or not.

6.3 Nested loops

Like Matryoshka dolls which contain smaller dolls, like test structures which can contain other test structures, a loop can contain other loops.

```
Variables Tic, Tac as Integers;
Begin
For Tic ← 1 to 15
  Write "It's been there."
  For Tac ← 1 to 6
    Write "It will be here."
  Next Tac
Next Tic
End
```

In this example, the program will write once "It's been there" and then six times "It will be here", and so on fifteen times. At the end, we will thus have $15 \times 6 = 90$ rounds in the second loop (the one in the middle), and so 90 instructions writing on screen the message "It will be here".

Note the major difference with the following structure:

```
Variables Tic, Tac as Integers;
Begin
For Tic ← 1 to 15
  Write "It's been there."
Next Tic
For Tac ← 1 to 6
  Write "It will be here."
Next Tic
End
```

Here, it will be printed on screen fifteen consecutive "It's been there", and then six consecutive "It will be here", and that's all.

As a consequence, loops can be *nested* (Case number 1) or *successive* (Case number 2). However, contrary to Poems, they **cannot be nested**. It would not have any logical sense, and moreover, a few languages would allow you to write this kind of aberrant structure.

```
Variables Tic, Tac  as Integers;
For Tic ← ... to ...
  Instructions;
  For Tac ← ... to ...
    Instructions;
  Next Tic
  Instructions;
Next Tac
```

So this is the first time and the last time you will see this kind of algorithms in this document and I hope it will be the same for you: you will never write this kind of things...

6.4 Another thing you must not do!

Let us study the following algorithm:

```
Variable A  as Integer;
Begin
For A ← 1 to 15
  A ← A *2;
  Write "Round number ", A;
Next A
End
```

In this case, the FOR instruction adds 1 to A in each round of the loop. However, inside the loop, we multiply by 2 the value of A. It is obvious that this kind of manipulations completely perturb the normal process of the loop and provide an error or an unexpected result.

The great joke of the programmer: It thus consists in manipulating, inside a FOR loop, the variable which is used as a counter in this loop. This technique must be **absolutely avoided!**

Chapter 7

Arrays

Good news! I already tall you that there were only four logical structures in programming. Well, we studied them all. Bad news: you still have a lot of things to learn...

7.1 What's for?

Imagine that we simultaneously need 12 values (e.g. some marks for computing an average mark). Until now, the lonely method that we know consists in declaring 12 variables, e.g. $N1$, $N2$, ... , $N12$. After a series of 12 distinct "READ" instructions, we will have an atrocity like:

$$\text{Avg} \leftarrow (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12;$$

Wow, it is tedious! Now imagine that we have a problem with thousands values to treat, e.g. computing the mean age of all Macedonian students in the country. Then the above approach leads to suicide!

Icing on the cake, if in addition, there is a situation when we do not know in advance how many values we will have to deal with, then we are completely down.

That's why programming allows us **to merge all these variables in one**, in which each value is designated by a number. In good English, we should say "Mark number 1", "Mark number 2", and so on. This is really easier as you can imagine.

> A set of values which hold the same name of variable and identified by a number is called an **array** or an **indexed variable**.
> The number which is used to identify each value in an array is called – Surprise ! – an **index**.
> Each time one must consider an element of an array, we put the name of the array, followed by the index of the element between brackets.

7.2 Notation and algorithmic use

In our example, let us create a table called Mark. Each individual mark will be denoted Mark(0), Mark(1), ... Yes, be careful, indices in arrays generally begin at 0 (and do not start from 1).

An array must be declared as such by specifying the number and type of values it will contain[1]. By following what is proposed in many programming languages, we arbitrarily decide that:

- the "boxes" are numbered from zero, i.e. the lowest index is zero.
- When declaring an array, we specify the highest index value (e.g. if we have 12 marks, the highest index is 11). At the beginning, it is not so natural but with practice, you will manage with.

```
Table Mark(11) as Integer;
```

We thus can create arrays containing all kinds of variables: numerical types, characters, Boolean, etc. However, except in a very few languages, we cannot mix different kinds of values in the same array.

The major advantage of arrays is that we can treat arrays thanks to loops. For example, in our computation of average mark:

```
Table Mark(11) as Number;
Variables Avg, Sum as Numbers;
Variable i as Integer;
Begin
For i ← 0 to 11
  Write "Type the mark number ", i;
  Read Mark(i);
Next i
Sum ← 0;
For i ← 0 to 11
  Sum ← Sum + Mark(i);
Next i
Avg ← Avg / 12;
End
```

NB: We made 2 successive loops for the sake of readability but we could also make only one which would do everything at once.

General remark: The index which designates the elements of an array can be directly expressed as a number, but also as a variable or a computed expression.

In an array, the value of an index must always be:

- equal to at least 0 (except if the first the first element of an array is indexed by 1, as it is done in some rare languages). But as we previously stated that we would start from 0 (as it is e.g. done in C or Visual Basic).
- an integer. In any language, Mark(3.14159) never exists!
- lower than or equal to the highest index of the array (number of elements minus one since we begin at zero). If an array Blabla has been declared with 25 elements, a code line writing Blabla(32) will provide an error.

I re-re-repeat it but if we use a programming language where indices start from zero, we must take care of it in the declaration.

[1] In practice, depending the programming language, the declaration varies: some languages need the number of elements while other want the highest index. As a consequence, in pseudo-code, it is just a choice of conventions.

```
Table Mark(13) as Number;
```

... will create an array with 14 elements (0 will be the lowest index, 13 the greatest one).

The great joke of the programmer: It thus consists in making the confusion between the **index** of an element of an array and the **value** of this element. The third house of the street does not necessarily contain three people, and the twentieth one twenty people. In programming, it is the same: there is no link between i and Trick(i).

7.3 Dynamic arrays

It is often met that we do not know in advance the number of elements that will comprise an array. Of course, one solution would consist in declaring a huge array (with say 10000 elements, why not?) in order to be sure that "everything will go inside". But first, we will never be sure that it will be sufficient, and moreover, because of the immensity of the memory space reserved – and mostly unused and wasted – we will lose a lot of speed during execution of the program, and even sometimes we won't be able to run it.

Thus, in order to solve this kind of situations, we can declare an array without specifying its number of elements. Then, in a second time, inside the program, we will fix this number thanks to an resizing instruction: **Resize**.

Please note that **while we did not give the number of elements in an array, this array cannot be used.**

For example, if we want to type the marks for computing the average, but we do not know how many marks there will be, the beginning of the algorithm will be as:

```
Table Marks() as Number;
Variable Nb as Integer;
Begin
Write "How many marks do you need to enter?" ;
Read Nb;
Resize Marks(Nb-1);
...
```

There is nothing hard in this technique but it is one of the fundamental tools that can be used in programming.

Chapter 8

Tricky techniques

This chapter won't present a new kind of data, a new set of instructions or something like that.

Its goal is to detail some programming techniques which have two common points:

- their knowledge is mandatory,
- they are a bit tricky.

And what are some kilograms of Aspirin with respect to the indescribable happiness that provide the understanding of intricacies of algorithm?

8.1 Sorting an array: the selection sort

First of these tools, and one of the most often met situations by the user: sorting an array.

During its brilliant career, how many times does the programmer needs to sort some values inside an array? It is unimaginable. As a consequence, instead to re-discover the wheel, it is better to learn some experienced techniques, even if they seem a bit hard at the beginning.

There are several ways for sorting an array. In this document, we will only see two: the selection sort and the bubble sort.

Let's begin by the selection sort. Imagine that you must sort an 12 elements array according to increasing values. The selection sort method works as follows: we first put the lowest element, then the following lowest one, and so on. As an example, if we start from:

45	122	12	3	21	78	64	53	89	28	84	46

we first start by looking for the lowest element among the 12 values of the array. In this example, it is the number 3 which is placed in the 4th position. Then, we exchange it with the first element of the starting array (45).

3	122	12	45	21	78	64	53	89	28	84	46

We then look for the lowest element in the array, but this time, **starting from the second position** (since the first one element of the array is the least one, we do not consider it anymore). Now, it is in the 3rd position (12). We then exchange it with the second element (122). It reads:

3	12	122	45	21	78	64	53	89	28	84	46

We then look for the lowest element of the array, starting from the 3rd element (the two first elements are sorted, so we do not touch them), and after permutation with the 3rd element of the array, we obtain:

3	12	21	45	122	78	64	53	89	28	84	46

And so on, until the penultimate.

If we write it in English, we should explain it like this:

- Main loop: let us consider the first element of the array as the starting point, and then the second one, etc, until the penultimate one.
- Secondary loop: from this starting point, let us look for until the end of the array the least element. Once it has been found, we exchange it with the starting point.

This reads:

```
Main loop: the starting point is shifted in each iteration.
For i ← 0 to 10
  We assume that the lowest element of the table is in the i-th
  element.
  PosMini ← i;
  We then consider the following elements.
  For j ← i+1 to 11
    If t(j) < t(i) Then
      PosMini ← j;
    EndIf
  Next j
  Now, we know who is the least element. We only have to do
  the permutation.
  temp ← t(i);
  t(i) ← t(j);
  t(j) ← temp;
  The i-th element is in the correct position. Let us study the
  following one.
Next i
```

8.2 An example of Flag: research in an array

We will now discuss about the use of the boolean variables: the so-called *"Flag"* technique.

The flag will stay down as long as an expected event won't occur. And, as soon as this event takes place, the flag rises (the value of the Boolean variable changes). As a consequence, the final value of the Boolean variable allows the programmer to know if the event took place or not.

All of this may seem to you a bit hazy but it should light up thanks to a extremely frequent example: the search of the occurrence of a value in a table. We will use this example for correcting a frequently made mistake among beginning programmers.

Consider an array with, say, 20 values. We must write an algorithm entering a number with keyboard and which tells the user about the presence or absence of this value in the array.

The first step consists in writing Read/Write instructions and the loop which goes inside the loop.

```
Table Tab(19) as Number;
Variable N as Number;
Variable i as Integer;
Begin
Write "Enter the value to look for";
Read N;
For i ← 0 to 19
  ???
Next i
End
```

We still have to fill the interrogating points inside the FOR loop. Of course, we will have to compare N to each element of the array: if two values are equal, then N belongs to the table. This will be translated by a IF ... THEN... ELSE. And that's the reasoning hastily by the programmer who is completely wrong in writing:

```
Table Tab(19) as Number;
Variable N as Number;
Variable i as Integer;
Begin
Write "Enter the value to look for";
Read N;
For i ← 0 to 19
  If N=Tab(i) Then
    Write "N belongs to the table";
  Else
    Write "N does not belong to the table";
  EndIf
Next i
End
```

And wham, this algorithm is a real disaster.

We just have to execute it for seeing it. Indeed, we have two possibilities: the value N belongs to the array, or on the contrary, it is not inside. **But in all the cases, the algorithm must return only one response, regardless to the number of elements which are in the array. However, the above algorithm prints on screen as many messages there are values inside the array (and so, in this example, 20 times!).**

As a consequence, there is a manifest error of design: writing the message must not be inside the loop. It must be outside. We know if the value was or not in the array when **scanning the array is completely done**. We therefore rewrite this algorithm while putting the test after the loop. We will use a Boolean variable that we will call Found.

```
Table Tab(19) as Number;
Variable N as Number;
Variable i as Integer;
Variable Found as Boolean;
Begin
Write "Enter the value to look for";
Read N;
Found ← FALSE;
For i ← 0 to 19
  ???
Next i
If Found Then
  Write "N belongs to the table";
Else
  Write "N does not belong to the table";
EndIf
End
```

We still have to manage the variable Found. It is done in two stages:

- a test inside the loop, indicating when the variable Found must become TRUE (i.e. when the value N is met in the array). And be careful: the test is not symmetrical. It does not contain an ELSE condition. We will talk about it later.
- The default affectation of the variable Found, whose starting value must be FALSE.

The complete algorithm then reads:

```
Table Tab(19) as Number;
Variable N as Number;
Variable i as Integer;
Variable Found as Boolean;
Begin
Write "Enter the value to look for";
Read N;
Found ← FALSE;
For i ← 0 to 19
  If N=Tab(i) Then
    Found ← TRUE;
  EndIf
Next i
If Found Then
  Write "N belongs to the table";
Else
  Write "N does not belong to the table";
EndIf
End
```

Meditate a little on this matter.

The challenge is to understand that in a search, the problem does not read the same, depending the way we consider it. We can resume it as follows: **it is sufficient that N is equal**

to a single value of Tab in order to belong to the array. On the contrary, it must be **different from all the values of Tab to not belong to it.**

That's why we must use a Boolean variable, **a "flag" which can stand up, but never falling down**. And this technique of flag (that we could nickname "Boolean variable asymetrical management") must be used each time that we find this kind of situation.

In other words, knowing this type of reasoning is as indispensable as learning to use it when needed.

8.3 Sorting an array + Flag = Bubble sort

The original idea of bubble sort is to say that a table sorted in ascending order is an array where **every element is smaller than the one which follows.** This idea which looks completely obvious is deeper – and more useful – than it looks.

Indeed, let us consider each element of a table, and let us compare it with the following element. If this order is not good, we permute both elements. And we repeat until there is no permutation to be done. The highest elements thus little by little "go up" from the last positions, which explain the name of this method (like bubbles in Champagne bottles – programmers are sometimes poets).

How does the bubble sort require the use of a flag? Because we never know in advance how many bubbles lift must be performed. In fact, all we can say is that we will perform the sort until there are no elements to permute. This is typically an "asymmetrical" situation: the fact that two elements are misclassified in a table is sufficient to say the the array is not sorted. On the contrary, we need all the elements to be properly arranged in order to have a sorted table.

We baptize the flag "PermutDone" because this Boolean variable will tell us if we just performed a permutation or not during the last scan of the array (in other case, it means that the table is sorted, and so that we can stop the bubble machine). The main loop will thus be:

```
Variable PermutDone as Boolean;
Begin
...
While PermutDone
  ...
EndWhile
End
```

What are we going to do inside the loop? Taking the elements of the array, from the first one to the penultimate, and processing a permutation if needed. This reads:

```
Variable PermutDone as Boolean;
Begin
...
While PermutDone
  For i ← 0 to 10
    ...
    If t(i) > t(i+1) Then
      temp ← t(i+1);
      t(i+1) ← t(i);
      t(i+1) ← temp;
    EndIf
  Next i
EndWhile
End
```

But we must not forget a capital detail: the management of our flag. Thie idea is that this variable will tell us the fact that there is at least one performed permutation. We thus must:

- assign the TRUE value to it when one single permutation has been done (if only one has been performed, we must restart at least once).
- put it to FALSE in each round of the main loop (when we restart a bubble round, there is no permutation which has been done),
- last point, we must not forget to run the main loop, and for that, we must give the TRUE value to the flag at the beginning of the algorithm.

The complete solution is:

```
Variable PermutDone as Boolean;
Begin
...
PermutDone ← TRUE;
While PermutDone
  PermutDone ← FALSE;
  For i ← 0 to 10
    If t(i) > t(i+1) Then
      temp ← t(i+1);
      t(i+1) ← t(i);
      t(i+1) ← temp;
    EndIf
  Next i
EndWhile
End
```

And I repeat: understanding and mastery of the principle of the flag are part of the well armed programmer's arsenal.

8.4 Binary search

We are now going to finish this chapter with a famous search technique, which reveals its usefulness when the number of elements is very high. For example, let us imagine that we have a

program which checks if a word exists in a dictionary. We can assume that the dictionary has been previously entered in an array (with one word per location). This can lead us to, say, 40000 words.

A first way for doing it consists in successively examining the words of the dictionary, from the first one to the last one, and compare them to the word to check. This works but it will be very long: if the word is not in the dictionary, the program will know it after 40000 rounds in the loop! And even if the word is in the dictionary, the response will need in average 20 000 rounds. It's a lot, even for a computer.

But there is another way to look for which is much more clever and which takes advantage to the fact that in a dictionary, the words are sorted according to the alphabetical order. Moreover, a human who looks for a word in a dictionary never reads all the words, from the first one to the last one: he also uses the fact that words are sorted.

For a machine, what is the most rational way to look for in a dictionary? It consists in comparing the word to check with the word which is in the middle of the dictionary. If the word is to check earlier in the alphabetical order, we know that we will look for it inside the first half of the dictionary. Otherwise, we now know we must seek it in the second half.

From that, we take the half of the kept dictionary, and we restart: we compare the word to check with the one which is in the middle of the piece of the left dictionary. We remove the bad half and we restart, and so on.

By cutting our dictionary in two, and then again in two, etc, we will work with pieces which only contain one word. And if we did not find the good word, this implies that the word to check is not in the dictionary.

Let us look on what happens in terms of the number of operations to be done, by choosing the worst case: the one where the word is absent of the dictionary:

- At the beginning, we look for the word among 40000 words.
- After the first test, we only look for among 2000 words.
- After the second test, we only look for among 10000 words.
- After the third test, we only look for among 5000 words.
- After the fourth test, we only look for among 2500 words.
- ...
- After the fifteenth test, we only look for among 2 words.
- After the sixteenth test, we only look for among 1 word.

And then, we know that the word does not exist.

Moral: We obtained our response in 16 operations against 40000 previously! There is no photo on the difference of performance between the barbary technique and the smart one. Caution, even if it is obvious, I repeat it with force: the binary search can only be performed on data that have been previously sorted.

And now I explained how to do, I let you translate it in Pseudo-code.

Taking the risk to repeating myself, understanding and mastery of the principle of the flag are part of the well armed programmer's arsenal.

Chapter 9

Multi-dimensional arrays

9.1 Why multiple dimensions?

You will probably ask :"Was one alone dimension not sufficient to our fully happiness?" Well, I will answer that you will see that with two or more, it is the Nirvana!

Imagine we want to model a checker game, and in particular the movement of a pawn. I recall that a pawn which is on a black square can move (in a simplified way) on four adjacent black squares.

With the tools that we have already seen, the simpler would consist in modeling the board as an array (called Square for example). Each "box" of the array is a square of the board, which e.g. contains 0 if it is empty and 1 if there is a pawn. We assign index numbers from 0 to 0 to the first line, from 10 to 19 for the second one, etc until 99.

If a pawn is placed in the box number i of the array, i.e. Square(i)= 1, we can move it to adjacent diagonal squares. This will force us to make some computations: the square located just below the square number i will have the number $i - 10$. The valid squares are thus the ones with indices $i - 9$ and $i - 11$.

Of course, we can make a complete program like this. But however, it does not provide some clearness.

As a consequence, it would be easier to model a checkers board by... a board!

9.2 Two-dimensional arrays

Programming yields us two ways for declaring arrays in which values are not located thanks to one but thanks to *two coordinates*.

Such a table thus reads:

```
Table Squares(9,9) as Integer;
```

This means: save for me a memory space of 10×10 integers, and when I will need one of these values, I will locate it thanks to two indices (like in the Naval War, or Excel – except that in Excel, we use letters and numbers and not only letters). For our pawn, things are going to light up. The square which contains the pawn is not Square(i,j). And the four accessible boxes in the table are: Square(i-1,j-1), Square(i+1,j-1), Square(i+1,j+1), Square(i-1,j+1).

There is no qualitative difference between a 2D-array with coordinates (i, j) and a mono-dimensional array with coordinates $(i * j)$. Exactly as in the checkers game we just talked about, any problem can be modeled with one way or the other. However, one or other of these techniques seems more natural to a given problem, and thus makes easier (or harder if we do not choose the good option) to write an algorithm and improves its readability.

Another remark: one classical question about 2D-arrays consists in knowing if the first (resp. the second) index represents rows (resp. columns) or if it is the contrary. I won't answer this question **because it makes non-sense**. "Rows" and "Columns" are visual, graphical concepts which apply to real-word objects. Indices in the tables only are logical coordinates pointing RAM memory addresses. If it does not convince you, think to the Naval Battle game : must the letters and numbers resp. design the rows and the columns? No matter! Each player can even choose a different convention, no matter! However, when a convention is chosen, the player of course keeps it until the end of the game.

9.3 N-dimensional arrays

If you understood the principle of 2D-arrays, there is no problem to manipulate arrays with 3, 4, or even 9 dimensions. It is exactly the same thing. If I declare an array `Luke(2,3,4,3)`, I reserve a memory space containing $3 * 4 * 5 * 4 = 240$ values. Each value is located by four coordinates.

The main difficulty of the systematic use of these arrays with more than 3 dimensions is that, when he proposes his algorithm, the programmer usually loves to make some small draws, to imagine loops in his mind, etc. But, even if it is easy to concretely imagine a one-dimensional array, even if it is feasible to do with two dimensions, it becomes the prerogative of a privileged minority for three-dimensional arrays and out of reach of any mortal beyond. That's life: human mind finds hard to represent things in space, and cries out when it deals with hyperspace (yes, this is the used name when there are more than 3 dimensions).

As a consequence, for practical reasons, arrays with more than three dimensions are not used by non-mathematician programmers. Indeed, because of their studies, mathematicians are used to manipulate spaces with N dimensions. But these are the only ones, and leave them in their corner, they are not people like us!

Chapter 10

Predefined functions

Some treatments cannot be performed by an algorithm. On the contrary, other ones only can be done after an untold suffering.

One example: computing the sinus of an angle. For approximating this value, the complexity of the algorithm is horrible! But, how does it work with small calculators that each of you know? You are provided some special keys, called *function keys*, which allow you to immediately know this result. With your calculator, if you want to know the value of the sinus of 35°, you will enter 35, then the `SIN` key, and you will have the result.

Any programming language thus offers a set of given *functions*. Some of them are essential because they can perform treatments that would be impossible to do without them. Other ones are used to help the user sparing him long and painful algorithms.

10.1 General structures of functions

Let us take the example of the sinus. Programming languages (which must be able to do the same thing than a calculator which costs 199 MKD) usually propose a `SIN` function. If we want to store the value of 35 in a variable A, we write

```
A ← sin 35;
```

A function is composed of three parts:

- the actual *name* of the function. This name cannot be invented! It must correspond to an language function. In our example, this name is `SIN`.
- Two brackets (an opening and a closing one) which are always **required, even if you do not write anything inside**.
- A list of values, which are essential for the proper execution of the function. These values are called *argument* or *parameters*. Some functions only need one argument. Others two, etc and lastly, some do not need any. Please not that even in this last case, brackets are mandatory! The number of necessary arguments for a given function cannot be invented: it is fixed by the language. For example, the sinus function only needs one argument (this is not surprising: this argument is the value of the angle). If you try to execute it while giving two arguments or no one, this will provide an error during execution. Please also note that these arguments belongs to a given type, and you must respect these types. For example, writing:

A ← sin("Luke, I am your father....");

is non-sense (and it is obvious!).

And entry, we find...

The great joke of the programmer: It is to assigning a function, whatsoever.
Any writing placing a function on the left of an Affectation instruction is absurd for two reasons:

- we know from Chapter 2 of this amazing document that we can only affect a variable.
- Then, because the goal of a function consists in producing a result, not receiving one.

Assigning a function will be considered as one of the worst algorithmic mistakes!

10.2 Text functions

One useful class of functions is one which allows us to manipulate strings of characters. We already saw that we could concatenate two strings of characters thanks to the & operator. But what we were not able to do, and that will be able now, consists in performing extraction of strings (less painful than dental extraction).

All languages propose the following functions, even if the name and the syntax may vary with respect to the language:

- `Len(String)` returns the number of characters in a string.
- `Mid(String,n1,n2)` returns a sample of the `n2`-characters long string, starting from the `n1`-th character.

The above functions if strings of characters are the only ones actually needed. However, in order to cancel time-consuming algorithms, languages also propose:

- `Left(String,n)` returns the `n` first characters of the string.
- `Right(String,n)` returns the `n` last characters of the string.
- `Find(String1,String2)` returns a number corresponding to the position of `string2` in `string1`. If `string2` does not belong to `string1`, then the function returns zero.

Examples:

`Len("Hi, how are you?")` returns 20 (keep in mind that spaces and punctuation are counted as characters).
`Len("")` returns zero.
`Mid("Zorro is back",4,7)` returns "`ro is b`".
`Mid("Zorro is back",12,1)` returns "`c`".
`Left("If only...",6)` returns "`If onl`"
`Right("If only...",6)` returns "`nly...`"
`Find("pure hapiness", "pure")` returns 1.
`Find("a pure hapiness", "pure")` returns 3.
`Find("pure hapiness", "techno")` returns 0.

It also exists in all programming languages a function which returns the character corresponding to a given ASCII code (Function `Asc`), and the inverse function (function `Chr`):

`Asc("N")` is 78.
`Chr(63)` is "?".

I emphasize: except if you are programming in a bit special language (like C which in fact treats strings of characters as tables), you could not do without both `Len` and `Mid` for processing the strings. But if computer programs usually have to work with numbers, they also frequently handle series of characters (strings). I know it becomes a refrain but knowing the basics about strings is more than useful, it is indispensable.

10.3 Numerical functions

In addition to the classical numerical functions which are also proposed by basic calculator (remember the introduction of this chapter) like `sin`, `cos`, `tan`, `sqrt`, etc, there are other functions that they do not necessarily provide but which are really useful.

10.3.1 Integer part

Three functions which are extremely popular allow to keep the integer part of any number (lower, upper or closer integer).

After: A1 ← Floor(3.14159), A1 is 3.
After: A2 ← Floor(3.94159), A2 is 3.
After: A3 ← Ceil(3.14159), A3 is 4.
After: A4 ← Ceil(3.94159), A4 is 4.
After: A5 ← Round(3.14159), A5 is 3.
After: A6 ← Round(3.94159), A6 is 3.

10.3.2 Modulo

This function allows you to recover the remainder of the division of a number by another number. For example:

A ← Mod(10,3); A is 1 because $10 = 3 * 3 + 1$
B ← Mod(12,2); B is 0 because $12 = 6 * 2$
C ← Mod(44,8); C is 4 because $44 = 5 * 8 + 4$

10.3.3 Generating random numbers

Another classical function is the one which generates a random number. Almost all the game programs need this kind of tool, whether to simulate a throw of dies of the chaotic movement of the spacecraft from the hell of death led by the horrible Zappa who wants to raid the Universe (fortunately, you are here to stop him!).

But generating numbers is not limited to games: it is useful in Mathematics, Physics, Signal Processing, Economics, etc which sometimes need stochastic models (they are models in which each variable can be estimated thanks to the others – i.e. thanks to calculations – but where we simulate an amount of chance by a "range" of chance.

For example, a demographic model assumed that a woman has an average x children during his life, say 1.5. But it also assumed that on a given population, this number can fluctuate between 1.35 and 1.65 (if we leave some uncertainty of 10%). Each year, i.e. each set of calculated values of the model, we will thus need to let the machine choose a random number between 1.35 and 1.65. In all the languages, this function exists and returns the following result. After:

```
Luke ← Rand();
```

we have $0 \leq \text{Luke} < 1$.

In fact, we realize that with a little practice, this `Rand` function allows us to generate any number in any range.

- If Rand() generates a number between 0 and 1, Rand() multiplied by Z generates a number between 0 and Z. As a consequence, it will suffice to estimate the "width" of the wanted range and multiply Rand() by this desired "width".
- Then, if the range does not begin by zero, it will suffice to add or subtract something to "calibrate" the range in the right place.

For example, if I want to generate a number between 1.35 and 1.65, the width of the range is 0.30. So $0 \leq \text{Rand}() * 0.30 < 0.30$.

It is therefore sufficient to add 1.35 to obtain the desired range. If I write:

```
Luke ← Rand()*0.30 + 1.35;
```

The value of Luke will be comprised between 1.35 and 1.65. And it is done!

10.4 Conversion functions

This is the last major category of functions which are here again available in all languages, because their duty is sometimes unavoidable.

Remember what we saw in Chapter 2: there exist different types of variables, which determine in particular the coding type that will be used. Let us take the digit 3. If I store it in an alphanumerical variable, it will be stored as a character, on one byte. However, if I store it as an integer, it will be coded with two bytes. And the configuration of the bytes will be completely different in both situations.

One obvious consequence, on which we already emphasized, is that we cannot do anything with anything and that we e.g. cannot multiply "3" by "5", if 3 and 5 are stored as characters. Until now, you may say that there is no scoop. But wait a minute thereafter.

Why not drawing the consequences, and properly storing numbers in numerical variables, characters in alphanumerical variables, as we always did?

Because there are some situations when we do not have the choice! There exists a way for storing (text files) where any data are stored as characters. So, if one wants to retrieve numbers and make make computations with, one shall be able to convert these strings of characters.

Also, all the languages propose a set of functions to make such conversions. We will find at least a function destined to the conversion of a string as a number (let's call it `Cnum` in pseudo-code), and another one converting a number as character (`Cchar`).

Chapter 11

Procedures and functions

11.1 Custom functions

11.1.1 What is it?

Especially if it is long, an application has a lot of chances to have to proceed the same treatments, or similar treatments, in several parts to its conduct. For example, entering yes or no (and the control which implies) can be repeated ten times in different times of the same application, for ten different questions

The most obvious (but also the less able) to program this kind of things consists in repeating the corresponding code as many times as necessary (the famous Copy/Paste that all of us know). Apparently, it does not provide headaches: when the machine has to ask the user, one copies the desired code lines while only changing what is necessary. But by processing by this way, one is preparing disillusioned days...

First because if the program structure written by this way may seem simple, it is in fact unnecessary clumsy. It contains repetitions, and if the program is long, it can be completely unreadable. However, being easily modified and so readable, even by those – and especially them – who did not write the code, is an essential criterion for a program! Once the program is not for yourself, but in the framework of an organization (business or otherwise), this need is felt acutely. Ignoring it is thus really serious.

In addition, at another level, such a structure yields considerable maintenance problems: in case of any code modification, one will have to hunt any appearance more or less identical of this code for correctly doing the change! And if one forgets one, wham, one let a bug.

We just choose another strategy, which consists in separating this treatment from the body of the program, and merge its instructions in a separated module. It then remains to call this group of instructions (which now exists in a single version) each time we need it. Thus, the readability is assured, the program becomes *modular*, and it suffices to do one single modification in the right place to make this modification having an effect in the whole application.

The body of the program is then called the *main procedure*, and these groups of instructions that we need are called *functions* and *sub-procedures* (we will see later the differences between them).

Let us take an example of questions to which the user must answer yes or no.

Bad structure:

```
...
Write "Are you married?";
Resp1 ← "X";
While (Resp1 ≠ "Yes") AND (Resp1 ≠ "No")
   Write "Enter Yes or No";
   Read Resp1;
EndWhile
...
Write "Have you got children?";
Resp2 ← "X";
While (Resp2 ≠ "Yes") AND (Resp2 ≠ "No")
   Write "Enter Yes or No";
   Read Resp2;
EndWhile
...
```

It's clear: there is a quasi-identical repetition of the treatment to be done. Each time, one asks an answer by Yes or No, with an input control. The only thing which changes is the name of the variable in which one stores the response. As a consequence, it must exist a trick.

We saw above that the solution consists in **isolating instructions** which ask a response by Yes or No, and in calling these instructions each time they are needed. As a consequence, this avoids duplications, and this cuts our problem in small autonomous pieces.

We will thus create a *function* whose role will consist in *returning* the answer (Yes or No) of the user. This word of "function" should not surprise us: we studied in Chapter 10 functions that are provided by the language, and we saw that their goal was to return one value. Well, this is exactly the same situation here, except that we will create our own function, called AnswerYorN:

```
Function AnswerYorN() as Character;
Variable Tip  as Character;
Begin
Tip ← "X";
While (Tip ≠ "Yes") AND (Tip ≠ "No")
   Write "Enter Yes or No";
   Read Tip;
EndWhile
Return  Tip;
EndFunction
```

We can also see the emergence of a new word: *Return* which tells which value must take the function when used by the program. This returned value (here, the value of the variable Tip) is somewhat contained in the name of the function itself, exactly how it was the case in predefined functions.

A function is always written **outside the main procedure**. Depending the languages, it can take different forms. But you must understand that these code lines are like satellites which exist outside the treatment itself. Simply, they are at its disposal and it can call them whenever it needs. If we consider back our example, once the function AnswerYorN has been written, the main program will contain the following lines:

Good structure:

...
Write "Are your married?";
Resp1 ← AnswerYorN();
...
Write "Have you got children?";
Resp2 ← AnswerYorN();
...

And it's done! We thus avoid unnecessary duplications, and if ever there was a bug in our input control, we would have to make only one correction in the function AnswerYorN to remove this bug from any application. Is not the life beautiful?

However, the most sagacious of you have noticed, in the title of the function in each call, the presence of *brackets*. As we said for predefined functions, these are required. And if you understand everything above, you must have some idea of what is going put in it...

11.1.2 Passing arguments

Let us consider another time the above example and let us analyze it. One prints on screen a message, and then we call a function AnswerYorN for answering the question. Then, later, one writes another message on screen and we call another time the same function for answering it, etc. This is an acceptable approach which can still be improved: since before each call of the function, there is a question, why not putting this message directly inside the called function. This implies two things:

- when we call the function, we must tell it which message it will print on screen before reading the answer.
- The function must be "told" it will receive a message, and be able to recover it for display.

In algorithmic language, we will say that the message **becomes an argument (or parameter) of the function**. This is certainly not new for you: we already used arguments with predefined functions. And, while building our own functions, we thus can construct our own arguments. This is how it is written:

Function AnswerYorN(Msg **as Character**) **as Character**;
Variable Tip **as Character;**
Begin
Write Msg;
Tip ← "X";
While (Tip ≠ "Yes") AND (Tip ≠ "No")
  Write "Enter Yes or No";
  Read Tip;
EndWhile
Return Tip;
EndFunction

Between brackets, there is now a variable, Msg, whose type is given and which tells the function that an argument should be sent during each call. About these calls, they are further simplified in the main procedure, thus reading:

```
...
Resp1 ← AnswerYorN("Are your married?");
...
Resp2 ← AnswerYorN("Have you got children?");
...
```

And this works!

A key note: here, we only passed one input argument. But of course, we can pass as many as we want and create functions with two, three, four, etc arguments. However, we must simply try to be not greedy and pass what we need (and only it, no more no less)!

In the above example, passing one argument in the function was elegant, but not essential. As a proof, it was well working in the first version. But we can imagine situations where we must absolutely design the function so as to transmit it a given number of arguments if we want it to do its task. Let us e.g. take all the functions which will do computations. Whatever they are simple or complex, one will have to send to the function the values thanks to what it will produce its result (simply remember the syntax of an Excel function for computing a sum or a product). It is also true with functions which will process strings of characters. Well, in 99% of the situations, when we will create a function, it should contain arguments.

11.1.3 Two words about functional analysis

As often in algorithms, if we use a tool as it has been designed for, all of this is not really hard. Custom functions are then logically deduced from the way we previously experienced with predefined functions.

The hardest, but also the most important, consists in acquiring the reflex to **systematically constitute well-suited functions when we have a given problem**, and know the good way to cut its algorithm in different functions to make it light, readable and efficient.

This part of reflection is called the *functional analysis* of a problem, and it is always by it that you must start: basically, in a first step, we cut the treatment in modules (functional algorithm), and in a second time, we write each module (standard algorithm). However, before coming here, we must find two other tools to use where the functions become unable to help.

11.2 Sub-procedures

11.2.1 Generalities

Knowing functions is good but in some situations, it is not useful. One can find situations in a program when we have to perform repetitive tasks which do not need to generate a particular value or, on the contrary, which have to generate more than one at once. If it is not clear for you, let us take two examples.

First example. Let us imagine that in my application, I need to clean several time the screen and re-print something like a small logo, on the upper left-hand corner. One might say that we need to create a function to do that. But which value would be returned by the function? None! Neither cleaning a screen nor printing a logo produce a result which can be stored in a variable. As a consequence, here is a situation when I need to repeat a code, but where this code does not have to produce a value.

Second example. During my application, I have to enter several times an array of integers (but each time, the array is different). Here again, one is tempted to do these array inputs in one single function. But the problem is that a function is only able to return one single value at once. It is thus not able to return an array, which is a series of distinct values.

So in both situations, since we are unable to treat them with a function, should we need to use repetitive code which we just have so vigorously denounce the weaknesses? Mmmmmh? You can well imagine not. Fortunately, everything is provided, there is a solution which consists in using *sub-procedures*.

In fact, *functions* – that we already saw – *are finally a special case of sub-procedures* – that we are going to see: the one where *the calling procedure should return one value and only one*. In all other situations (where we do not return any value and where we return several ones), we must not use the particular and simplified form (the function), but the general one (the sub-procedure).

Let us talk about what is common to functions and sub-procedures but also about their differences. Here is how a sub-procedure reads:

```
Procedure Thingummy(...)
...
EndProcedure
```

In the main procedure, the call to the sub-procedure Thingummy is:

```
Call Thingummy(...);
```

Let us establish an initial inventory:

- while a function was characterized by the keywords **Function ... EndFunction**, a sub-procedure is identified by the key-words **Procedure ... EndProcedure**. I know, this remark a bit trivial...
- When a function was called, its (returned) value was always affected to a variable (or integrated in the computation of an expression). On the contrary, the call of a procedure is an *autonomous instruction*. "Execute the procedure Thingummy" is an order that is sufficient unto itself.
- Because of this reason, any function should contain the "Return" instruction. For the same reason, **the "Return" instruction is never used in a sub-procedure.** The function is a computed value, which returns its result to the main procedure. The sub-procedure is just a treatment. It "is worth nothing".
- Even when we understood the three first points, we are not completely at the end of our troubles!

11.2.2 The problem of arguments

Indeed, we have to examine what can be put inside the brackets, instead of the three points, both in the declaration of the sub-procedure and in its call. You can imagine: this is where we will find the tools which will allow the exchange of information between the main procedure and the sub-procedure (in fact, this last sentence is too restrictive: it would be better to say: "between the calling and the called procedures". Because a sub-procedure can of course call another one to accomplish its task).

As for functions, the values that flow from the calling procedure (or function) to the called sub-procedure are named *arguments* or *input parameters* of the sub-procedure. As we can see, whether for sub-procedures or functions, these things exactly play the same role (transmit an information from the code that gives instructions to the subcontractor code), they also have the

same name. Only small difference, we said this time that is was arguments or *input* parameters. Why?

Simply because in a sub-procedure, it may be necessary to seek return results results to the main program. However, here, contrary to functions, nothing is planned: the sub-procedure does not "return" anything (as we previously saw, it does not contain the "Return" instruction). These results that the sub-procedure must transmit to the calling procedure will thus also be conveyed by parameters. But this time, it will be parameters operating in the other direction (from the sub-contractor to the payer): we will thus call them *output parameters*.

This allow us to restate in other words the fundamental truth learned above: **any sub-procedure having one and only one output parameter can be written as a function** (and between us, this is a preferable statement because it is slightly easier to understand, and so to remember).

Until then, it is OK? Yes? So take an aspirin and continue the reading. If not, take an aspirin and restart from the beginning. In both cases, do not forget the big glass of water for the aspirin.

We still have to examine one detail which is of importance, as you can imagine: how can we do to make the language understand which parameters should work in input which ones in output...

11.2.3 How does it work all this?

In fact, if I say that a parameter is "in input" or "in output", I state something about its role in the program. I say what I want it to do, the way that I want it works. But programs themselves do not care of my desires, and this is not this classification that they use. This is all the difference between saying that an electric outlet is used to plug a razor or a coffee machine (which characterizes its role), and saying that the outlet is 220 V or 110 V (which characterizes its technical type, and which is an information which concerns the electrician). Like electricians, programming languages do not care about what will be the role (input or output) of a parameter. What they require is to know their voltage... Sorry, I mean the *way of passing* these parameters. There are only two:

- passing *by value*,
- passing *by reference*.

Let's look at what it is.

Let us consider our previous example with our function AnswerYorN. As we saw, nothing prevents us to rewrite this function as a procedure (since a function is only a special case of sub-procedure). For the moment, we will skip the question of how to return the response (contained in the variable Tip) to the main program. However, we will declare Msg as a parameter passed by value. This will read:

```
Procedure AnswerYorN(Msg as Character by value)
Variable Tip  as Character;
Begin
Write Msg;
Tip ← "X";
While (Tip ≠ "Yes") AND (Tip ≠ "No")
  Write "Enter Yes or No";
  Read Tip;
EndWhile
??? How to transmit Tip to the calling procedure ???
EndProcedure
```

The call of this sub-procedure will e.g. take the following form:

```
M ← "Are you married?";
Call AnswerYorN(M);
```

What will happen?

When the main program is on the first line, it affects the variable M with the label "Are you married?". The following line triggers the execution of the sub-procedure. This immediately creates a variable Msg. Since it was declared as a parameter passed **by value**, Msg will have the same content than M. This means that *Msg is now a copy of M.* The information contained in M have been completely copied (duplicated) in Msg. This copy will remain throughout the execution of the sub-procedure AnswerYorN and will be destroyed at its end.

An essential consequence of all or that is that if the sub-procedure AnswerYorN contained an instruction that would change the contents of the variable Msg, this would not have any impact on the main procedure in general, and on the variable M in particular. **As the sub-procedure only works on a copy of the variable which has been provided by the main program, it is not able, even if we would like it, to modify its value.** Said differently, in a procedure, **a parameter passed by value only can be an input parameter.**

It is also a limit (compounded by the fact that the information thus copied occupy twice more memory space) and a security: when one sends a parameter by value, we are sure that, even in case of a bug in the sub-procedure, the value of the transmitted variable will never be changed by mistake (i.e. erased) in the main program.

Let us suppose now that we declared another parameter, Tip, indicating that time that it will be passed *by reference*. And let us adopt now the following writing for the procedure:

```
Procedure AnswerYorN(Msg as Character by value, Tip
as Character by reference)
Begin
Write Msg;
Tip ← "X";
While (Tip ≠ "Yes") AND (Tip ≠ "No")
  Write "Enter Yes or No";
  Read Tip;
EndWhile
EndProcedure
```

The call of the sub-procedure would e.g. be:

```
M ← "Are you married?";
Call AnswerYorN(M,Resp);
Write "Your answer is ", Resp;
```

Let us study in detail the mechanism of this new writing. About the first line which affects the variable M, nothing differs from above. However, the call of the sub-procedure causes two very different effects. As we already said, the variable Msg is created and immediately affected by a copy of the contents of M, since it was passed by value. But for Tip, it is completely different. As this time the way of passing was by reference, **the variable Tip will not contain the value of Resp, but its address, i.e. its reference.**

Therefore, **any modification of Tip will be immediately redirected to Resp.** Tip is not an ordinary variable: it does not contain a value but only a reference to a value which is located elsewhere (in the variable Resp). This is thus a completely new kind of variable that is different from all that we already saw. Such a variable is named a *pointer*. All the parameters passed by reference are pointers, but pointers are not limited to parameters passed by reference (even if these are the only ones that we will see in this course). It should be understood that this kind of strange variable is directly managed by languages: **from the moment when a variable is considered as a pointer, any affectation of this variable will be automatically translated by the modification of the variable on which it points.**

Passing a parameter by reference thus represents two advantages. First, we save some memory space since the parameter in question does not duplicate the information sent by the calling procedure, but it only contains the address. Then, it allows to use this parameter as **both input and output**, since any modification of the value of the parameter will affect the corresponding variable in the calling procedure.

We summarize it thanks to a table:

	pass by value	pass by reference
input use	Yes	Yes
output use	No	Yes

But then, if the pass by reference presents both above advantages, why don't we systematically use it? Why do we bother with passes by value, which not only waste memory space but also forbid the use of the variable as an output paramter?

Well, just because we won't be able to use it as an output parameter, and that this limitation can also be an advantage. Said differently, it is a *security*. This is the warranty that whatever the bug that should affect the sub-procedure, this bug will never make disasters in the variables of the main program that it should not touch. That's why, when we wish to define a parameter that we know that it will be exclusively used in input, it is better to lock it, while defining it as passed by value. On the contrary, parameters will be passed by reference if and only if we absolutely need them in output.

11.3 Public and private variables

Let us summarize the situation. We just saw that we would cut a long process that eventually contains redundancies (our application) in different modules. And we saw that these informations could be transmitted between these modules according to two modes:

- if the called module is a function, by *return of the result*,
- in all other situations, by *transmission of parameters* (whatever these parameters are passed by value or by reference).

In fact, there is a third and final way to exchange information between different procedures and functions: it consists in not exchanging them, ensuring that these procedures and functions literally share the same variables. This implies to use special variables, readable and usable by any procedure or function of the application.

By default, a variable is declared inside a procedure or function. It is thus created in this procedure, and disappears with it. During all the time of its existence, such a variable is only visible by the procedure which saw its birth. If I create a variable Luke in a procedure Yoda which calls during its execution a sub-procedure R2D2, there is no way that R2D2 could access to Luke (and don't talk about modifying it). That's why these variables by default are said *private* or *local*.

But besides this, it is possible to create some variables that will be declared in a procedure, but, from the moment when they will exist, will be common variable to *all* the procedures and functions of the application. With such variables, the problem of the transmission of values from a procedure (or a function) to another one does not arise: the variable Luke, that exists for all the application, is accessible and editable from any code line of this application. No need to transmit or return it. Such a variable is thus called *public* or *global*.

Of course, the way to declare a public variable varies with respect of the programming language. In pseudo-code, we will use the keyword "Public":

```
Variable Public Luke as Number;;
```

So why not make public all the variables, and thus spare so tedious efforts to pass the parameters? This is simple, and this is always the same thing: **global variables need a lot of memory space**. As a consequence, the principle that must be used to choose between public and private variables shall be that of *the economy of means*: we only declare as public variables that must be. And each time it is possible, when we create a sub-procedure, we use the passes of parameters instead of the public variables.

11.4 Can we do everything?

To this question, the answer is obviously: yes, we can do everything. But this is precisely the reason why we may soon come to also do absolutely anything.

What is "anything"? For example, as we just saw, putting global variables everywhere, under the excuse that there will be as many parameters that we won't have to pass.

But we can imagine other atrocities.

For example, a function, whose an input parameter would be passed by reference, and modified by the function. This would imply that this functions does not produce one but two results. Said differently, under the aspect of a function, it would work as a sub-procedure.

Or conversely, one could imagine a procedure which would modify the value of one parameter (and only one) passed by reference. It would thus be a procedure that in reality would be a function. Even if this last example is not a serious drama, it participates to the same logic which consists in confusing the code while passing the tool for another one, instead of adopting the most clear and as readable as possible structure.

Finally, do not rule out the possibility of particularly vicious programers who, thanks to a mix of parameters passed by reference, global variables, procedures and function poorly chosen, would eventually give an absolutely illogical and unreadable code, in which hunting for mistakes is an achievement.

End of the jokes: the principle which may guide any programmer is both solidity and clearness of the code. **A well programmed applications is an application with a clear structure,**

whose different modules do what they say (and say what they do), and can be tested (or modified) one by one without perturbing the rest of the building. Therefore, you must:

1. **limit the use of global variables**. They must be sparingly and wisely employed.

2. **Group as distinct modules** of the pieces of code which have a given functional unity (programming by "blocks"). That is to do the hunt to (quasi-)redundant code lines.

3. to make **functions** of these modules **when they return a single result**, and **sub-procedures in all other situations** (which implies to *never* pass a parameter by reference in a function: either we do not need, whether it is not a function if it is really needed)

Following these hygiene rules is essential if we want that an application looks differently that the Postman Cheval's Ideal Palace[1]. Because an architecture in which we do not understand nothing is probably poetic, but there are situations when efficiency is preferable to poesy. And for those who still doubt it, programming (unfortunately?) belongs to these circumstances.

11.5 Functional algorithms

To close this chapter, here are some additional words about the general structure of an application. As we repeatedly said, it will be commonly formed of a main procedure, and of functions and sub-procedures (which will call themselves others functions and sub-procedures if needed, etc). The typical example is the one of a menu, or a summary, which "plugs" on different treatments, and so on different sub-procedures.

The *functional algorithm* of the application is the *cut* and/or the graphical *representation* of this general structure, whose the goal is the understanding at glance of which procedure does what, and which procedure calls other one. The functional algorithm is thus somewhat the construction of a skeleton of the application. It is placed in a more general and more abstract level than the normal algorithm, which details step by step the treatments done inside each procedure.

In the construction – and the understanding – of an application, both documents are essential, and constitute two successive stages of the elaboration of a project. The third – and last – stage consists in writing, for each procedure and function, the detailed algorithm.

An example of functional algorithm: the Hangman game

All of you know this game: the player must guess a word randomly chosen by the computer, with a minimum of tries. For that, he proposes alphabet letters. If a letter belongs to the word to be find, it appears. Otherwise, the number of false answers increases of 1. After 10 bad answers, the game is over (and lost). This small game will allow us to show the three steps of the realization of a little complex algorithm. Of course, one should ignore these three stages, and run like hell straight into the lion's den, i.e. directly write the final algorithm. But, except if you are particularly talented, it is better to follow the canvas below, because difficulties are easier to solve when we carve them...

[1] I let you look for what it is.

Stage 1: Data dictionary

The goal of this stage consists in identifying information which will be needed to solve the problem, and to choose the best way of coding treat this information. This is an essential step of the reflection, that you must consider carefully... However, nine out of ten novice programmers shorten this reflection, when they even skip it. Punishment does not wait long: as the algorithm is build on bad foundations, the programer understands while writing it that for example, the choice of information coding yields to an dead-end. The precipitation is punished by the fact that one is obliged to start again from the beginning, and we have finally lost more time than was believed in win...

So, just before writing anything, the questions you must ask yourself are the following:

- which information will the program need to finish its task?
- for each of these information, what is the best coding? In other words, one that without wasting the memory space, will write the algorithm as simple?

Once again, do not hesitate to spend some time on these questions, because some mistakes, or some forgotten things can provide a lot of problems then. And conversely, the time invested at this level is largely caught at development time itself.

To play to Hangman, here is the list of information we need:

- a list of words (playing with a single word will be quickly boring),
- the word to guess,
- the letter proposed by the player in each round,
- the total number of bad answers,
- and lastly, the set of letters already found by the player. This information is capital for two reasons: first for knowing if the whole word has been guessed and then for printing at each round the state of the word (in each round, found letters are printed on screen while other ones are replaced by indents).
- To this list, since we are gentlemen, we might add a list containing all the letters already proposed by the player, whether correct or not. This will prohibit the player to propose a new letter previously played.

This list of information is perhaps not exhaustive: we will probably need some additional variables (loop counters, temporary variables, etc). But essential information is definitively there.

Now arises the problem of choosing the smartest method for coding. If for some information, it will be easy to do, for others we will have to make choices (and if possible clever ones!). Let us go!

- For the list of words to guess, it is a set of alphanumerical data. These information could be part of the body of the main procedure, and thus be stored in RAM in the form of an array. But it will be better to put these words in a `*.txt` file and to open it (we do not waste memory and if we need, we can add words without modifying the program). We will create a file called `Dict.txt` in which each word will appear in one line.
- The word to find is just a string stored in a variable `Word` declared as character.

- Moreover, the letter proposed by the player is a string which will be stored in a variable `Letter` declared as character.
- The number of false responses is a number which will be stored in `FalseResp`.
- The set of found letters is typically the kind of information that may be coded by several ways. Remember that while printing, we need to know for each letter of the word if it has been guessed or not. One possibility would consist in having a string of characters with all previously found letters. This is not a bad idea but, for the pleasure of change, we will make another choice: we will provide an array of Booleans, counting as many locations correspond to a letter of the word to find, and indicate by its value if the letter has been found or not (TRUE if found, FALSE otherwise). The correspondence between the elements of the array and the word to guess is immediate. The programming of loops will be facilitated. We will name our Boolean array `Verif()`.
- Finally, the set of all proposed letters il be stored in a string named "Propos".

We now have sufficiently thought to this problem for proposing the data dictionary itself.

Name	Type	Description
`Dict.txt`	Text file	List of words to guess
`Word`	Character	The word to guess
`Letter`	Character	Proposed letter
`FalseResp`	Integer	Number of false answers
`Verif()`	Array of Boolean	Previously found letters from the word to guess
`Propos`	Character	List of proposed letters

Stage 2: Functional algorithm

We can now proceed to the realization of the functional algorithm, i.e. cutting the problem in logical blocks. The purpose is multiple:

- facilitate the implementation of the final algorithm,
- save time and lightness while isolating sub-procedures and functions. And possibly avoid multiple repetitions of code which slightly differ from each others.
- Allow the division of labor among programmers, each being assigned to the specific programming of procedures or specific functions (this is essential when we enter in the word of professional programming).

In our case, a first block would consist in preparing the game (choice of the word to guess). Since the goal is to return one and only one value (the word randomly chosen by the machine), we can give this task to a function `ChoiceWord`. Please note that this division is only made for the sake of readability and not an absolute necessity: we might also do it in the main procedure.

The main procedure will have the shape of a While loop: indeed, while the party is not over, we restart the series of treatments which represent one round. But how can we precisely know if the game is over? It can end either because the number of incorrect answers reached 10, either because all the letters of the word have been guessed. The best will be to give the review of it to a specialized function, `GameOver`, which returns a numerical value (0 to signify that the game is in progress, 1 in case of victory, 2 in case of defeat).

Let us now turn the game.

The first thing to do consists in displaying on screen the current state of the word to guess: a mixture of found letters and dashes (corresponding to not yet found letters). All this can be supported by a specialized sub-procedure called `PrintWord`. About the initialization of variables, they will be done as usual in the main procedure.

Then we must enter a letter, being sure that the input is correct. Here again, a specialized procedure, `EnterLetter`, will be used.

Once the proposal done, we must check if it corresponds to a letter of the word to guess or not. This will be done by a sub-procedure called `CheckLetter`.

Finally, once the game over, we must print on screen the conclusion. To this end, we call a last procedure, `EndOfGame`.

To complete the functional algorithm, we must provide a table of the different functions and procedures, exactly as we did for the data dictionary. Here it is:

Name	Type	Description
`ChoiceWord`	Function	Choice of the word to guess in Dict.txt
`GameOver`	Procedure	Returns a value for showing if the game is in progress or ends (with victory or defeat)
`PrintWord`	Procedure	Prints on screen the state of the word
`EnterLetter`	Procedure	Proposes a letter to the computer (must tell if it has already been proposed)
`CheckLetter`	Procedure	Check if the proposed letter is present or not
`EndOfGame`	Procedure	Prints on screen the conclusion

In addition, one could also make a scheme of the way our application is working thanks to blocks, each one representing a function or a sub-procedure.

At this stage, the functional algorithm is completed.

Stage 3: Detailed algorithm

Normally, it only remains for us to handle each procedure separately. We begin by sub-procedures and functions, to complete the draft of the main procedure.

As an example, I give you the first function: `ChoiceWord`.

```
Function ChoiceWord() as Character
Table List()  as Character;
Variables NbWords, Choose  as Numbers;
Begin
Open "Dict.txt" on 1 in Lecture;
NbWords ← −1;
While not EOF(1)
  NbWords ← NbWords +1;
  Resize List(NbWords);
  ReadFile 1, List(NbWords);
EndWhile
Close 1;
Choose ← Round(Rand() ∗ NbWords);
Return List(Choose);
EndFunction
```

Chapter 12

Complementary concepts

12.1 Structured programming

Small return on a concept very quickly flown above: that of "structured programming". In fact, until now, we were doing structured programming without knowing it. So rather than explain what it is, I prefer to take the problem in the other end: how it is not.

In some languages (historically they are often old languages), programming lines are numbered. And the code lines are executed by the computer in the order of these numbers. So far, in itself, no problem! But the trick is that in all these languages, there is an instruction, denoted *Goto* in pseudo-code that directly sends the program to the specified line. Conversely, this type of language does not contain instructions as EndWhile that close a block.

Let's take an example with an "If ... Then ... Else" structure.

Structured programming

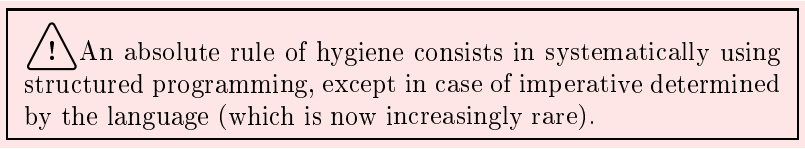
```
If Condition Then
  Instructions1;
Else
  Instructions2;
EndIf
Instructions3;
```

Unstructured programming

```
1000   If Boolean Then Goto 1200;
1100   Instructions 2;
1110   ...
1120   ...
1130   ...
1190   Goto 1400;
1200   Instructions1;
1210   ...
1220   ...
1400   Instructions3;
```

As you can see, a program written in such languages is presented as **a series of connections tangled into each other**. On the one hand, we can not say that this promotes readability of the program. On the other hand, is a major source of errors, because sooner or later you forget a "Goto" or you put an extra one, etc. As a consequence, when a complex algorithm exists, it can become a tangled jungle.

Conversely, structured programming, especially if care is taken to streamline the presentation through comment lines and practicing the indentation, avoids mistakes, and reveals its logical structure very clearly. The danger is that while most programming languages are structured, they still provide mostly the opportunity to practice unstructured programming. In this case, the lines are not designated by numbers, but some can be identified by names (called "labels") and we obtain a branch instruction.



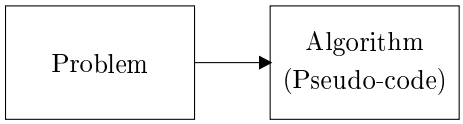
Said differently, even when a language offers you the possibility to make sprains to structure programming, don't seize it in any pretext.

12.2 Interpreting and compiling

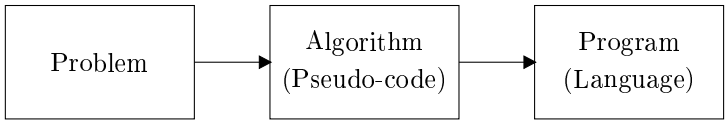
With this section, we leave actual algorithms for entering in the more technical field of practical realization. Or if you prefer, these lines are the culmination, the grand finale, the ultimate ecstasy, the great consecration of this document.

In all modesty, of course.

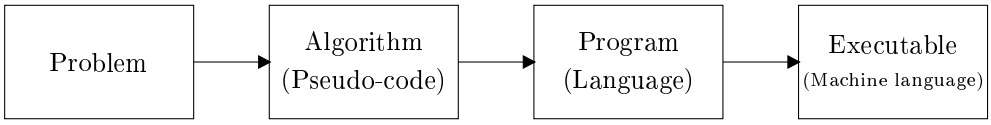
Until now, we have worked on the first stage of the implementation of a program: writing an algorithm.



In fact, if the algorithm is well written, without logical mistake, the next step must not provide any conceptual problem. We just have to do a simple translation.



From here, the work of the programmer is virtually finished (in reality, there is still an inevitable phase of testing, corrections, etc which is often very long). But by the way, for the computer, this is here that troubles begin! Indeed, no computer is in itself able to execute instructions as written in a particular language. The computer only understands one language, coded in binary (or hexadecimal) called the *machine* (or *assembly*) language.



This is the goal of any language: saving you from programming in binary (pure horror, you can imagine) and making yourself understandable for the computer in a (relatively) readable way.

That's why any language must proceed to a *translation* in machine language in order to make the program executable.

There exist two translation strategies, which are sometimes available in the same language:

- the language translates instructions as and when they arise. This is called *interpretation*.
- The language begins by translating all the program into the machine language, thus creating a second program physically and logically distinct from the first. Then and only then, it executes the second program. This is called *compiling*.

It goes without saying that interpreted language is more manageable: it can directly execute its code - and then test it - as and when we type it, without each time passing by the extra step of compiling. But it is equally clear that a compiled program runs much faster than an interpreted program: the gain is commonly a factor of 10 or 20 or more.

It must be clear that for a professional use, any application is programmed in a compiled language. However, in some companies, the first step of a work is done with interpreted code, in order to easily and quickly validate an idea. Once it is done, it is always translated in a compiled language, for the sake of speed.

12.3 A genuine pervert logic: recursive programming

You know how the programmers are: you can not give them anything without them trying to play with, and the worst is that they succeed.

The programming of custom functions provided development of a special logic, especially suited to the treatment of some mathematical problems or games: *recursive programming*. In order to explain you how it works, let us take the example of the computation of a factorial.

Remember that the value of the factorial of an integer n reads:

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

We programmed it with a FOR loop and it was done.

But another way of seeing the things, not more or less right, would say that the factorial of any number n reads:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{otherwise} \end{cases}$$

Said in English, it means that the factorial of a number is equal to the product between this number and the factorial of the previous number. Once again, it is not more or less right than with the FOR loop. It is just different.

If we program it, we can thus imagine a function called `Fact` which will compute the factorial. This function performs the multiplication of the number passed as an argument by the factorial of the previous number. And this factor of the previous number will of course be itself determined by the function `Fact`.

In other words, we will create a function which will **call itself a given number of times** in order to provide its outcome. This is the recursion.

There is another part of the recursive definition of the factorial that we did not use. Indeed, when $n = 0$, then $n! = 1$. Contrary to The Young and the Restless, the recursion will stop when n will be equal to zero.

This produces the following algorithm, which can be really useful (some really complex functions have a relatively simple recursive form and helped a lot of researchers to find new results).

```
Function Fact (N as Integer) as Integer
Begin
If N = 0 Then
  Return 1;
Else
  Return Fact(N-1) * N;
EndIf
EndFunction
```

You will note that the recursive process somewhat replaces the loop, i.e. it is an iterative process. You will also note that we take the problem upside down: we start from a number and we decrease back to 1 to compute the factorial. This reverse effect is characteristic of the recursive programming.

To conclude about recursion, three fundamental remarks:

- to address some problems, recursive programming is **very economical for the programmer**: it allows to do things properly, with a few instructions.
- However, it is **very expensive in terms of machine resources.** Because during execution, the machine will have to create as many temporary variables as "rounds" of pending functions.
- Lastly but not least, and this is the final joke, **problems that are formulated in recursive terms can also be formulated as iterative terms** in most of the situations[1]. So if the recursive programming is helpful for programmer, it is usually not necessary! But as future programmers, you must know these techniques on which one can always fall one day or another.

[1] Indeed, some mathematicians found some functions which can be defined with a recursion and not thanks to a loop. However, these functions are special cases and in almost all the situations you will meet, you will be able to translate the recursive function that you are using as a loop.