Bob Walraet

Programming, the Impossible Challenge

North-Holland

PROGRAMMING,
THE IMPOSSIBLE CHALLENGE

BOB WALRAET
Cullinet Software
Brussels, Belgium

1989

NORTH-HOLLAND
AMSTERDAM · NEW YORK · OXFORD · TOKYO

No responsibility is assumed by the Publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

ISBN: 0 444 87128 4

Publishers:

ELSEVIER SCIENCE PUBLISHERS B.V.
P.O. Box 103
1000 AC Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:

ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

Library of Congress Cataloging-in-Publication Data

Walraet, Bob.
Programming, the impossible challenge / Bob Walraet.
p. cm.
Bibliography: p.
Includes index.
ISBN 0-444-87128-4 (U.S.)
1. Electronic digital computers--Programming. 2. Programming languages (Electronic computers) I. Title.
QA76.6.W3297 1988
005.1--dc19 88-7698
CIP

PRINTED IN THE NETHERLANDS

Table of Contents

PART 3 THE WORLD OF DATA

CHAPTER 15 THE YEARS OF FILES AND DATA-BASES

CHAPTER 16 THE YEARS OF DATA NORMALIZATION

CHAPTER 17 THE YEARS OF THE STRUCTURED QUERY LANGUAGE

CHAPTER 18 THE YEARS OF THE B.O.M.

CHAPTER 19 THE YEARS OF DATA STRUCTURING

PART 4 THE WORLD OF KNOWLEDGE

PART 1

THE WORLD OF PROGRAMS AND SYSTEMS

TOPICS

History of computer hardware
Assemblers
FORTRAN
Flow-charts
The Monitor
The macroprocessor
The subroutine
Operating systems
Overlay technology
Naming theory
COBOL
Algol60
Scope rules
PL1
Storage classes
Expressions
Stacking
Multiprogramming
Pascal
Decision tables
Concurrent processes
Deadly embrace
Semaphores
The banker's algorithm
Bourses
Goto-lessness
Structured programming
What is a program?
DREC structures
Coroutines
Cullinet's LRF
Recursion
Polish notation
Virtual storage
On-line Programming
Cullinet's ADSO
Fourth generation technology
Data Flow Diagrams
Menu-driven programming
Jobs and tasks
Job control language
C
BASIC
ADA
Information hiding
Rendez-vous techniques
Micro-programming

A LIST OF MILESTONES

In *PROGRAMMING LANGUAGES - THE FIRST 25 YEARS* [IEEE Transactions on Computers, December 1976], Wegner gives a list of milestones that greatly influenced the realm of programming. This list, slightly adapted and expanded, is printed below. The dates are somewhat approximate due to the fact that many of the development efforts have taken years to come about.

- *19th century: Ch. Babbage and Ada Lovelace's analytical machine*
- *40's: Türing Machines*
- *1944: the EDVAC report by von Neuman.*
- *50's: the first symbolic code assemblers.*
- *50's and 60's: macro-processors*
- *1958: FORTRAN*
- *50's: formal language theory*
- *60's: Compiler theory*
- *1960: ALGOL60*
- *1960: COBOL*
- *1964: PL1*
- *1960: LISP*
- *1963: SNOBOL*
- *1968: ALGOL68*
- *60's: functional programming*
- *60's: program semantics and verification*
- *1967: interactive APL*
- *1970: PASCAL*
- *70's: Structured Programming*
- *70's: information hiding and data abstraction*
- *70's: the concept of Software Engineering*
- *1971: PROLOG*
- *1973: C*
- *1979: ADA*
- *70's: the relational data model*
- *70's: data-base management systems*
- *70's: fourth generation technology*
- *70's: Expert Systems*

Chapter 1
THE YEARS OF ASSEMBLER AND FORTRAN

A more or less historical view of the simplicities of the 50's, including assembler, macroprocessor and Fortran.

Let there be the machine

In the beginning there were the bars and the pebbles used to count and add numbers. Crude devices which were mechanized into the familiar *abacus* by the Greek and the Chinese. For centuries they remained the only computing devices used by mankind. In the 17-th century the German Wilhelm Schikard and the Frenchman Blaise Pascal created the very first *computing machines* based on revolving cogwheels. These machines used decimal numbers and were able to perform carry operations in such a way that the basic arithmetic operations (add and subtract) became possible at the mere turn of a crank. The machines underwent a number of improvements (such as automatic multiplication and division, a numeric keyboard and a paper roll printer) but the basic principle remained unchanged so that their descendants were still in use, not so many years ago, in almost all accountancy offices and banks. At the beginning of the 19-th century, however, another -parallel- development took place: in 1801 Jacquard used *punched cardboard strips* to control weaving machines. And then there was Charles Babbage, a British mathematician at Cambridge University. He was the real pioneer: in 1822 and 1848 he designed two mechanical *differential engines* that could process tables of numbers. In 1833 he created the very famous *analytical engine*, a real computer as we mean it today, which could read in *punched cards* comprising the input, perform any arithmetic operation, print the results, and *repeat any computation sequence* at will. The fact that Babbage could only use mechanical components to assemble his machine and had to rely on steam power is the reason why he did not succeed in building it. Nevertheless, he and his assistant, Ada Lovelace, described very many computation sequences that could solve a class of problems. These were truly the first *programs*! The ideas were in the air now; in 1890 Hollerith built statistical machines that used punched cards. They were used during the great census held in those years. The success was such that Hollerith founded a company which evolved into IBM. It took another fifty years before the next step came about: using telephone central relay technology, George Stibitz created a computing machine based on *relays*, which performed its operations considerably faster than any of its predecessors. During World War II, Türing and a team built Colossus, a machine intended to decipher the German ciphered messages. The trend was set now. In 1944, Howard Aiken built the first relay-based machine that incorporated Babbage's analytical principles. IBM took an interest and produced a host of devices that could together perform business computations. You may remember the *tabulators, card punches, sort-merging machines, duplicating machines, card readers* and *electric typewriters* which were the equipment of the very first *computing centres*. The tabulating machines were programmed by *wiring* the sequences of operations upon the now prehistoric *pin-board*. Once wired by the user, the pin-board was carefully archived in a pin-board cabinet, the first *program library* ever! Evolution took hold of these machines and the relays were replaced by *electronic valves*. Witness the birth of the **ENIAC** (Electronic Numeric Integrator and Computer), conceived by Mauchly and Eckert at the University of Pennsylvania (1944-46). So there it was: the Machine. A huge, somewhat awe-inspiring beast, full of lamps that flickered, and radiating a discomforting heat. It contained 18000 valves and needed an enormous amount of energy. In 1949, von Neuman, at Princeton University, created the **EDVAC** (Electronic Discrete Variable Computer), the very first computer that held a program in core and could execute it. The modern computer was born. It had the so-called von Neuman architecture: program and data held in core, without any difference. A great achievement. The unit of information handled by the machine was the binary digit (the *bit*) represented as an electronic signal that could be on or off, thus implementing the values 1 and 0. The memory of the machine was an assembly of such bits (better: groups of bits called *bytes*) which were laid out upon a revolving magnetic drum. The drum was described by means of a location map indicating exactly were each byte was registered. This location was called an *address*. Program execution consisted of reading and writing byte values at specified addresses of the memory. A byte could hold anything: an encoded program instruction (or part of it) or a data value. And it is a sad fact: the existence of a memory layout with addresses (of places in an array of core locations) forcing programs to manipulate one memory location at a time proves to be the greatest bottleneck in present day computers... But in the days of the first hardware generation, the concept was fantastic.

The older tabulating machines were, so it appeared, great at adding and counting (binary) numbers. Therefore, if a problem could be broken down into a flow of numbers and a series of computations upon them, it could be *computed* on the machine. Problem solving was in essence thought to be the *writing* of sequences of purely arithmetic operations upon the binary digits. And this was the very first misunderstanding: the (en)coding of the (numerical) problem solution was seen as the important aspect. The need for **designing** the solution was totally ignored; indeed, one was not even aware that this could be a need. In fact, the machine called for a court of tinkerers around itself, not of thinkers. How could this have been different? Indeed, machines were so limited that the problem solution was highly conditioned by the very way in which the coding was to be expressed. Everything was machine-oriented, and coders were cheerfully bending problems to suit the Machine. One computed, sorted and tabulated quite happily. The Machine beamed and almost purred. When the Machine took on the von Neuman philosophy and discarded its pin board coding interface to replace it by **punched cards** that could be read by a specialized apparatus -cards that could be produced very cheaply, cards that could be stored- a new era seemed to open. The Machine itself seemed to feel it, as it became faster, became bigger but also less monstrous. Initially, punched cards contained the programs in pure *binary code*, created by the programmers. A tedious and error-prone activity. There was a rather limited attempt at improving the quality: pseudo-code of a more symbolic nature, read in and interpreted by a small program residing in the machine. A rather inefficient way of using the machine. But very soon, the binary code work was doomed to vanish altogether as **assembler** came about.

In retrospect, that was a normal, predictable step in computer evolution. Assembler language: merely a set of *symbolic codes* to replace the operations previously coded in bits. Assembler: a program, loaded from binary cards into and running on the Machine, almost organically linked to it, able to translate the symbolic code sequences into pure binary code. Nothing very smart about assemblers, but comfortable animals as compared to pin board coding. Now, behold the birth of the assembler fanatic! Writing assembler programs was considered to be a *higher level* speciality, reserved for Data Processing wizards. This was mainly because the rise of assembler language coincided with the extension of the machine's basic instruction set. Programming (understand: the writing of assembler programs) became an art. A totally self-justified one. The assembler programmers of those days (late 50's, early 60's) wanted problems to solve, not because of the need to solve them, but for the pure delight of writing assembler code. The Data Processing people, even more than before, concentrated on *programming* (the word that replaced coding) and still did not worry about solution design. Huge books were written by smart authors, who delivered to the crowds their knowledge of assembler "tricks", standard programs (soon called *routines*) doing standard things, catalogs of prestidigitation and legerdemain. Sophistication increased as assemblers allowed symbolic addressing and as the machine took on register-based addressing. Assembler programs were now an irresistible must. Pin board coders were considered senile, of another age, and promptly discarded. Assembler was for the second generation. And very unfortunately, the assembler programmer became totally infatuated with his "principles of operation" and lost sight of objective and perspective. Something was very wrong indeed.

The first to feel it were the numerical analysts. They did not want to lose time on pure programming. They had immense problems to solve and needed to express solutions easily. They had some other, more special needs as well. In fact, what they wanted the machine to do was the computation step by step of very involved arithmetic sequences. A process of repetition was the core of the numerical problems. And such repetitions (or *iterations*) were, more often than not, to contain other iterations, possibly at rather deep level of nesting. Iteration step width had to be easily modifiable. Computations themselves had to be updatable in their tiniest details. In fact, programs had to be "legible" and maintainable and transferable to other machines. Moreover, in this context, the accent was upon values, not upon where the data was stored in the machine (a concept designated by *address*). All these criteria could not be truly obeyed by assembler language. Something new came about: **FORTRAN** or the FORmula TRANslator.

Machines kept evolving at a fast rate. The *valve-based* **first generation** machines (IBM 600 series) gave way to the **second generation** machines which were *transistor-based.* Five years later, the creation of *integrated circuit technology* was the start of the **third hardware generation,** which characterizes the era of the sixties. The **fourth hardware generation** is based on the *(Very) Large Scale Integration* technology (LSI and VLSI). Currently the Japanese are working at the **fifth generation** hardware.

First there was machine code and assembler followed

Let us come back to the beginning. Before we investigate FORTRAN, let us look at some machine code first. Machines of the pioneer era were simple (programming them was, however, not so simple...). They had a behaviour very much inspired by the mechanical desk calculators. And they had memory, which was a nice new feature. Memory allowed the machine to *record* data, but most importantly, it allowed **programs** to reside in the machine. Von Neuman had had that brilliant idea: to the machine, whatever resided in memory was data. But according to the way in which the programmer activated the machine, some piece of data would be interpreted as a program and executed.

[NOTE: The assembler and machine characteristics given in this section and the following do not reflect the intricacies of programming for the first generation of machines (IBM series 600 and 700); indeed, my intention is not to concentrate on these pre-historic hardwares, but rather to give an aperçu of assembler philosophy from the language side; therefore I have prefered to use examples in the spirit of the third generation of machines, those of the 60's].

Execution of a program was controlled by a special *register* the so-called **program counter**, which holds the *address* in memory (the cell number as it were) of the next instruction to be executed. The machine also had one basic rule: unless otherwise forced, the execution of the instruction pointed to by the program counter would cause an increase of the counter so that it would now point to the following instruction (the next cell as it were), and execution would proceed from there. Other special memory cells also existed: the **registers**. Typically, there was an **accumulator**, used for plain arithmetic and a **B-register** used for extended arithmetic such as multiplication and division. The **instructions** were **words**, i.e. *numbers* that could be contained in a fixed number of cells (for instance four contiguous cells, something which would later be called a 4-*byte* instruction). An instruction was composed of an **operation code** and the address of a memory cell serving as the object of the operation. For electronic reasons, everything in the machine was binary based. Early machines had cells that could hold a value of maximum 2^6-1 (i.e. 63) and they were called *octal* machines because $2^6 = 100_8$, which in turn means that an octal cell can contain octal numbers comprised between 00 and 77. In such a machine, one *digit* is necessarily comprised between 0 and 7. Later machines were going to increase the cell size so that 2^8-1 (i.e. 255) became representable. These machines are *hexadecimal* and use digits between 0 and F (the hexadecimal digits A to F stand for the decimal values 10 to 15).

Let us now come back to the instructions of the machine. For the sake of the example, we will assume an octal machine with 4-byte instructions. A typical instruction is the one that loads the contents of a given machine word into the accumulator. The machine word whose **contents** have to be loaded is 4 bytes long and is designated by the cell address of its leftmost byte (high order byte). This is the **operand**. The layout of such an instruction could be as indicated in *figure 1*. Cell 1 holds the **operation code** (i.e. 01_8), cell 2 is not used (and therefore set to 0), and cells 3 and 4 contain the **address** of the operand. Quite often, the accumulator and the B-register were the first and second word of memory, so that they could be *addressed* by the instructions as need arose. Later machines, however, would have explicit registers, for instance 8 of them (on octal hardware) or 16 (on hexadecimal hardware), numbered 0 to 7 (or 0 to F). The instruction layout was modified so that one byte in cell 2 could now indicate a register as a second operand. Our typical

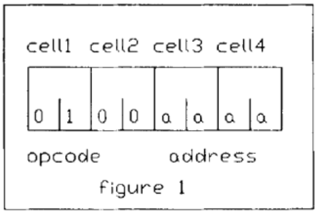


figure 1

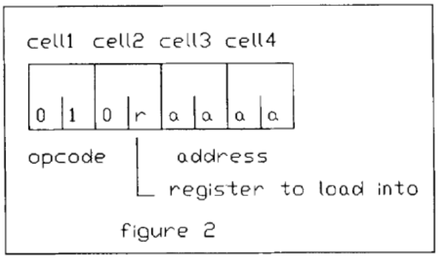


figure 2

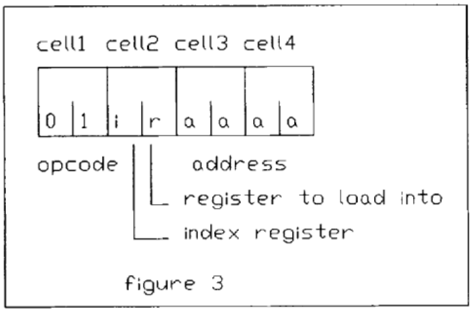


figure 3

load instruction could now be as in *figure 2*, which now means: load the contents of address **aaaa** *into register r*. But registers were also used to allow more involved addressing schemes. For instance, our load instruction could be even further complicated by the appearance of a so-called **index register.** This is one of the registers but it is used to keep an address. The address of the cell where to load from is now computed as the sum of the *value* held in **i** and **aaaa** (see *figure 3*). On some machines the addressing scheme could be even more involved: for instance, if the address computed as indicated above gave a negative number, then the cell located at the *absolute value* of this address was interpreted to contain the address of the operand (and so on, if this was also a negative value).

Entering a program into the machine's memory was not exactly an obvious matter. Using the pin board was one way. Another way was to use toggle switches on the machine itself: one first set an address of a cell which was to receive an instruction, and next one set the toggles to the binary value of the instruction. A very cumbersome way of doing this. So, it was quite a relief when card readers came about that allowed reading of punched cards upon which the instructions could reside in some form coded (and punched) by means of a typewriter-like keyboard. Of course, now the machine needed a *built-in* program that could handle the card reader and **load** the read in instructions into the correct memory cells. That built-in program was the seed of today's gigantic operating systems... The comfort brought by the card reader incited programmers to wish for even more comfort. And that was the motive for having assemblers. In fact, assemblers are programs, residing in the machine, and able to read instructions written not in binary or octal or hexadecimal code, but in some symbolic language and translate them into the binary code, which could then be punched onto cards for later usage. Typically, an assembler would replace our load instruction **0100aaaa** by the symbolic equivalent **LD address.** An immense improvement. But the real trick of assembler was in the way it allowed programmers to specify the address part of the instruction. An assembler *statement* has the following layout: **label opcode symbolic-address.** The symbolic address could be a mere number, literally the address of a memory cell. So, **LD 734** stands for **01000734.** But, assembler also allowed an address to be given a name, by using a *pseudo*-instruction such as **label DEF value.** Thus **ABC DEF 635** gives the symbolic name ABC to a certain cell and moreover places the value 635 in that cell. Thus, the **label** ABC stands for the address of a cell and can be used in the address part of an instruction: **LD ABC** loads the contents of cell ABC. Obviously, also instructions could be so labelled. This allows branching instructions to have such a label as a target. The question as to which cell exactly corresponds to ABC is solved by assembler, in the sense that it assigns monotonously increasing addresses to each line (statement or pseudo-instruction) of the program. Very soon, assembler came up with another pseudo-instruction, **ORG address**, which allowed programmers to force the assembler to abandon its address numbering scheme, accept an explicit address given by ORG and go on numbering from there. A way of working with explicit addresses. The revolution of assembler was that it created the *impression* of readability. Programs could be read and were found very self-documentary indeed. This, of course, was an illusion. Assembler language was a mere one- to-one transliteration of binary code, in which addresses were given names. And that was all there was to it. Assembler language programs did not at all allow the casual reader (and for that matter, not even the attentive one!) to see from the code what problem the program was supposed to solve. Problem **visibility** was very very low. FORTRAN was going to improve things in that respect.

And then there was FORTRAN

Many computer scientists of today condemn FORTRAN as an unsound language. I will, for the moment being, reserve my opinion. It is a fact that in the nothingness of the fifties, Fortran was an amazing novelty. It was the very first language that brought an amazing collection of new concepts. Paramount was the notion of **variable**, which was the start of *naming theory*. So, what was new there? Not much, but quite a lot. Let us come back to assembler, and see what a name stands for in that environment. All names in assembler are in fact labels of either the DEF instruction or the EQU instruction. Suppose we have an assembler line **ABC DEF 123** and another one **LD ABC**. So, the *value* 123 is loaded into the accumulator. The *name* ABC stands for the *value* 123. This is misleading, however. Suppose the DEF instruction was assembled at *address* 743; then the octal representation of the LD instruction is 01000743. This means that for the LD instruction, the name ABC acts like the address 743. If on the other hand we had written **ABC EQU 123** and **LD ABC**, this would have loaded the contents of cell 123 into the accumulator. In octal, the instruction would now have been 01000123. In this case ABC is the address 123. So, the interpretation of **LD ABC** depends on the way in which ABC is defined (by EQU or DEF). Both however stand for an address. But, if we write the assembler instructions **ABC EQU 123** and **LD =ABC** then the assembler first creates a *dummy* instruction **xxx1 DEF ABC** and really sees the LD as **LD xxx1** meaning that the value 123 gets loaded into the accumulator. In this case, ABC, although defined by an EQU, stands for a value. This is because the operand of a DEF is interpreted as a *value*. To say the least, this is confusing: the interpretation of a name depends on the usage that is made of it. In other words, the name does not unambiguously represent an object. Fortran changed this: a name stands for a **variable**, i.e. for a well-defined object that has a value of a precise type. And this is true whatever usage is made of the name. As a consequence, Fortran needed a statement to give a value to a variable. This was the **assignment statement**. Of course, a statement that serves to *assign* a value to a variable is powerful only if the value can be *computed*. Therefore, the assignment statement also defined the computable **expression**, usually as an arithmetic polynomial such as ABC*3+H35. In even more general terms, Fortran introduced the concept of **action**. An action is defined in terms of an operation that causes one or more variables to change value in a finite and deterministic way (of course, Fortran also had actions that changed no values but had another type of effect, such as printing on paper) . In essence, therefore, Fortran was the first true data manipulating language and a Fortran program is nothing more than a scheme of consecutive actions upon variables, submitted to rules of behaviour. Indeed, variables need to have values before they can be used in expressions. So a program must somehow start by giving values to variables. As long as a variable has not received a value, it is **unvalued** and it cannot undergo any other action than **valueing.** Moreover, for a variable to be able to receive a value, it must, in its internal machine implementation, be realized somewhere. It must have an address. We say that it must be **sited.** Now the Fortran compiler sited variables implicitly, which was very wise: the programmer had no need any more to even think in terms of addresses. And in many Fortran compilers, a variable was also valued by default to 0, so that no problems could arise. So Fortran did a very nice job in creating variables. But what about actions? These were covered by the **statements** of the language. And although this looked like a rather human concept, it was too vague. A good part of the problems that were going to arise later in programming techniques is due to exactly that vagueness. *Statement* is a very fuzzy notion; it implies nothing special, and in many cases not even an action. Fortran statements could be actions, but they could be other things as well. And this was very much inspired by assembler. In assembler programming, there existed instructions that allowed one to test certain conditions and the result of the test could be used in specialized branch instructions so as to transfer execution flow to some other place in the program. This was considered natural, since the data a program worked with varied from run to run, so that specific cases could appear for which alternative instructions were needed. And Fortran was in the same situation: it was not possible to imagine that all programs could be constituted of mere sequences of, say, assignment statements. One needed the breaks of sequence. And Fortran did it in a very simple way by introducing the GOTO and IF statements. Are these statements actions? Actually, they *do* something: they conduct program execution flow. But they are of a meaning quite different to those statements that modify values.

It is time now to have a look at programming in Fortran.

Fortran defined a program as a portion of **text** comprised between a card with the word PROGRAM and a card with the word END. The bulk of the contained text was to be composed of *executable* statements such as:

(1) The assignment statement, e.g. **x=732*(abc+def*3)/(klm+7)**. Clearly, this notation (with nested parentheses) is much more comfortable than assembler, which would need at least 10 instructions using 3 intermediate "variables" to achieve the same result.

(2) The branching statement **GOTO statement-label**, a statement which introduces the **label** notion. In Fortran each statement can be labelled by means of a *number*. This number does not at all represent an address. It is a mere *name* given to a statement by the programmer. The statement has the effect of transferring execution flow to the specified label. The GOTO statement has a second format, largely inspired by assembler, the so-called **computed GOTO**, as follows: **GOTO (label1,...,labeln),i**. This statement branches to *labelj* if at the moment of branching the variable i has value j. By using this statement, the programmer can create a so-called *case* structure; in other words he can set up specific actions triggered by specific discrete values of a *control variable*. There was yet a third format of the GOTO, the **assigned GOTO**, as follows **GOTO i,(label1,...,labeln)**. In this case i must have a label number as value and more precisely one of the labels between the brackets.

Assigning such a label value to i is done by means of a specialized statement: **ASSIGN label-value TO i.** This statement is a true assignment but somehow labels were considered different to "usual" variables and this explains the special form of the statement. A rather messy situation, which would have to wait for PL1 to clean it up.

(3) The conditional statement **IF(conditional expression) statement** meaning that if the stated condition is true, the *statement* is executed. Otherwise, processing continues just after the IF statement. As an example consider:

IF (A.GT.BCD.AND.Z.EQ.3) GOTO 1002

This would once more cost about 10 assembler instructions with some intermediate labels. There is another format of the IF statement:

IF (numeric expression) label1,label2,label3

which means that if the numeric expression is negative, control is transferred to label1, if the expression is zero control goes to label2 and otherwise it goes to label3.

(4) Finally, Fortran also defined the iteration as a group of statements executed repeatedly under the control of a *counting* variable, also called a **control variable**. The structure is as follows:

```
                 DO label I = from,to,increment
                 ...     \
                 ...     | contained statements
                 ...     /
label            CONTINUE
```

It means: execute the contained statements with the variable I equal to the *from* value. Execute it a second time with variable I augmented by *increment*, and go on like that till variable I exceeds the *to* value, in which case processing continues just after the CONTINUE statement. Notice, and this is important, that the *DO label* and the statement having that *label* (it need not be CONTINUE) constitute **delimiters**, which define a **group** of statements. In this group there may be other groups, which allows for nested iterations. Moreover, a GOTO may not branch from outside to within an iterable group. These rules make the construct much more attractive than what could be achieved in assembler language.

Very comfortable and readable programming had become possible for the first time ever. That there were many problems in Fortran would appear only later. Even the most modern common languages have not yet cleaned up all the weaknesses. In fact, most of the known languages of today (Algol, PL1, Pascal, C, ADA) are true descendants of Fortran. Had Fortran been different, our present day languages would also have been different. The inheritance is heavy. Other languages such as LISP, PROLOG, APL, were based on totally different premises, as we will see, but they never had the same public audience. Many computer scientists consider this an unfortunate historical mistake.

Assembler's revenge

And yet, the deed was not done. Again, the machine grew in complexity. Its peripherals now included magnetic tape drives, allowing bulk input/output (the story of input/output will be told in chapter 15). And, of course there was Fortran. Fortran was comfortable, but it was totally incomprehensible to the machine. The machine wanted binary code, so its human worshippers still fed symbolic code into assemblers. A Fortran **compiler** was needed, a program that would read in Fortran statements and *translate* them into executable binary code. And such a compiler was naturally itself written in assembler. Since Fortran was essentially a simple language, the process of translation was not very difficult, apart from the treatment of expressions. Expressions could be very involved, with lots of parentheses. The pioneers had had a hard time in setting up a correct translator for such beings. There had been approaches which somehow counted parentheses and created a so-called *Klammergebirge*. The theory of languages had not yet been invented. Refer to chapter 11 for the story of that field. The Fortran compiler was something very new: for the first time computer people had created a program that constructed another program. A very powerful thought, but its richness was not perceived until later. One still needed assembler to create such specialized programs.

At the same time, other evolutions came about. The machine became undisciplined. Computer people had given it too many powers. Uncontrolled, the machine started wreaking havoc. It needed a Master. This was the **Monitor**. A program (developed in assembler, of course!) which resided in the machine and exercised control over various activities, such as device management (reading from the card reader, writing to the printer, handling tapes). Monitor also reshaped the machine's comprehension of happenings: Monitor wanted the machine to accept **jobs**, which it executed by **scheduling** them, one after the other. Jobs were programs, framed by a number of control statements (job control language) indicating the beginning and the end of the stack of cards the program was residing on, and also the designation of the devices this job needed to operate with. Jobs could be stacked as a **batch** in the card reader and Monitor would be able to differentiate them. So, the machine was tamed. Those were IBM's days of IBSYS and IBJOB (Monitor names, actually). Monitor distributed machine power to jobs. And what could have been anticipated happened: Monitor became a tyrant. An uncontainable trend towards huge **operating systems** was set. A cleavage immediately appeared: there were men and women alike who created, extended and worshipped the Monitor. They were the *systems programmers* and, because Fortran did not allow systems programming, they took an assembler oath and despised Fortran. These people were under the hardest of stresses: they had to keep the monitor operational. But the monitor was a very harsh mistress. It had needs, and due to its growing complexity, it resisted changes in very subtle ways. So it coaxed its developers into very ad hoc programming avenues, and as a result, the complexity grew and grew and grew. The systems developers became mere handworkers, having nervous breakdowns and stomach ulcers. They lost grip and discipline. Their programs became more and more difficult to read and maintain. So, they needed help in that wilderness. Help came in the guise of the **macroprocessor**. This was a very essential being, but once more, people failed to grasp its real implications.

The wicked flow-charts this way came

Meanwhile, part of the truth had hit some of the Computer worshippers. They realized that program design was called for. A method would be welcome. And in

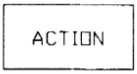


figure 4

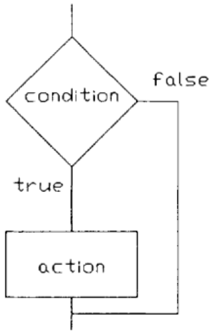


figure 5

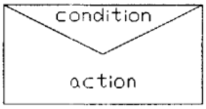


figure 6

search of something -anything- they unearthed an oldie: **flow-charting**. All of a sudden, a discipline was inflicted upon the Data Processing world. The flow-chart was alive and well and meant to stay. A flow-chart expresses the *flow of control* through program execution. In this context, the notion of **action** is all-important. An action is anything that modifies the value of variables (or has some more esoteric side effect such as Input/Output). The action therefore is made equivalent to one or a group of program statements. This easy equivalence explains the unbelievable success flow-charting had. Flow-charting uses a graphical syntax, in which the action is expressed as a rectangular box (*figure 4*). The chart also *connects* the boxes by means of vertical descending *arrows*, thus signifying control flow. A rule that people manipulated is that although many arrows can enter a box, only one arrow can leave it. This means that there always is one and only one action that follows a given action (even if it only were the conventional action **stop** which ends the program). Flow is synonymous to sequence, and therefore it was taken to be synonymous to programming also: the writing of a sequential list of statements. But pure sequence was not enough. One also needed to express that some actions could be executed only if a certain condition held. The flow-chart uses a lozenge and some arrows to indicate just that (*figure 5*). Now, looking at the drawing, one may wonder if that is a truly natural representation. Does this flow-chart indeed indicate a conditional statement? Or does it rather show conditional skipping of an action? Wouldn't it be more natural to represent a conditional action as the structure in *figure 6*? Using the lozenge introduces a new type of connective arrow, one that goes horizontally, then descends and again goes horizontally towards the next action. Nothing prevents us, of course, from bypassing many sequential action boxes with this kind of connective line. Another usage of such graphical structures is in representing Fortran's iteration. Iterations are necessarily conditional: at each step of the iteration some test is required to verify whether a next step is required. Thus, the flow-chart *naturally* expresses this in one of two ways, as *figure 7* indicates. Both ways use the lozenge to indicate the test. Both somehow enclose the iteration body by using a connective line that actually branches back (goes upwards). Of course, the two structures do not mean the same. If the iteration condition is such that iteration is no longer needed,

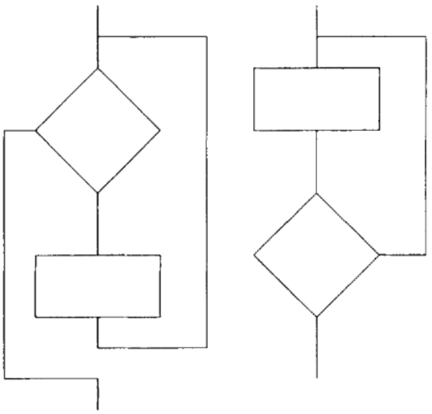


figure 7

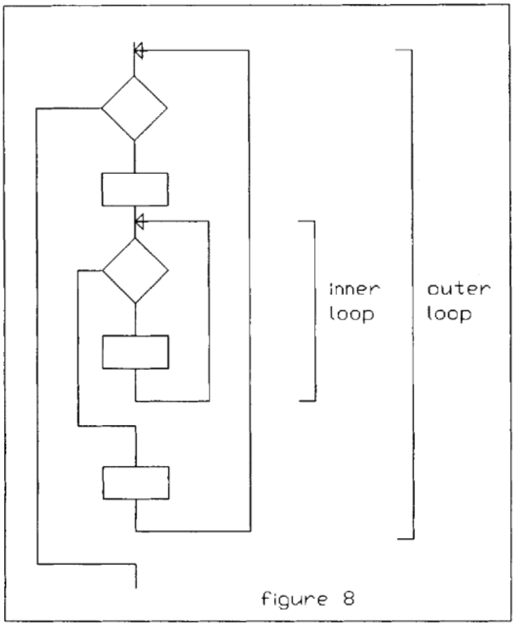


figure 8

from the start, then the rightmost structure will still execute the iteration body once before abandoning it. The important consideration is that only connective flow lines indicate the extension of an iteration or of a conditional action. And that is rather weak for such important execution profiles. Flow-charts express flow, and are not particularly concerned about *structure*. This becomes apparent when we try to draw nested iterations as in *figure 8*. Not very readable any more. The interesting thing to notice is that the connective flow lines (those that span one or more actions) correspond very precisely to Fortran statements such as IF and GOTO. Even more interesting is the fact that the DO statement is a shorthand for the combination of IF and GOTO.

An intriguing question is: can connective lines cross each other in a flow-chart? Of course they can! But is that a sound situation? I will let this question rest for the moment being. Still, because of such questions, some computer people started having doubts about the usability of flow-charts. It did not seem very wise to confuse execution flow and program meaning.

Assembler gets pregnant

Computers had begun their conquest of another realm: the business application world was being engulfed slowly but surely. But what happened here was very different to the Fortran world. Numerical analysts made their own programs: they considered the computer (with Fortran) as a tool to be used in their daily work. In business applications the situation was that although the computer was needed in order to speed up the business processes, most of the people concerned considered that accessing the computer was beyond their grasp. So, they quite naturally turned to the Data Processing specialists themselves and required them to work on business problems. A new breed emerged: the application programmer. He was recruited in the ranks of systems programmers and therefore he was allowed to stick to his ways: assembler language became the language of applications as well. Fortran and its world was another planet.

In the business field, problems were usually simple, but were burdened by huge volumes of data to be processed. This was very different to numerical analysis where there was not so much data but there were immense computations to be conducted. So, the applications world needed ways to efficiently access and produce the data. This gave birth to the **files** (for their story, see chapter 15) and all programs in this field were bent under the yoke of file management. Data could not reside in the program, on the one hand because it was too bulky and on the other hand because it had to survive the program's execution for re-usage in a later execution. In such cases, Fortran used punched cards. But this was much too impractical for the business data. So, magnetic tapes were used instead. These peripherals had until then served to hold the Monitor programs when the computer slept. Now it gained a new actuality. Programs were also going to use them. And, as expected, Monitor took over the responsibility of managing the tape drives and the access to them. But Monitor failed in one respect: it did not hide the peripherals from the programs. Instead, it defined a set of device-dependent commands for usage within the programs. In retrospect, this way of doing this has been the most effective of brakes upon program conception for years. It has also severely hampered the design of truly high-level languages.

As the number of application programs grew, it was felt by systems people that these programs could just not sit in with the Monitor and happily interfere with it. Systems programmers came up with the notion of **interface**: a small slot through which application programs could communicate with the Monitor and ask for services, mainly of file management scope. Other connections were totally banned. To this effect, hardware grew a new assembler instruction: the **monitor call** instruction (which I will designate by MC (IBM calls it SVC)), an instruction that allowed assembler programs to request a Monitor service. This type of requesting needed operands both for informing Monitor about the service required and for allowing it to return information to the requester. A discipline was needed, which was reflected by standard sequences of assembler code one had to use lest failure ensued. A great cause of mistakes and unpredictable program behaviour. Alleviation came, however. The standard code sequences were grouped under some more human *generic* name with some operands, a *macroscopic* assembler instruction as it were. Some ten to twenty such macroscopic instructions were added to assembler language in order to cover file and device management. This was the birth of the **macroprocessor**. Did the people who invented this new feature realize what they had unleashed? I am convinced they didn't. Even nowadays, programmers using conventional languages fail to grasp the importance of macroprocessors. Only those who took the other avenue, that of Artificial Intelligence, understood what it was all about. In its own, as yet very limited way, the macroprocessor was the basis of it all. Initially, the macroprocessor was used by systems people to create macroscopic instructions to cover file management with. Very soon, however, the set of such instructions was extended to contain other important services such as obtaining the date and time of day, or producing a binary listing of program contents (a *dump*).

Fiat macroprocessor

The macroprocessor was a remarkable being. It had been spawned by assembler, but soon it developed an independent life. At its birth, the macroprocessor was seen as the means of replacing a number of lines of (assembler) code by a more symbolic name. Usage of this name in a program would cause its replacement by the assembler lines. When such a name was used, this was called a **macro-call**. Thus, the macro-call **DATE** would, for instance, be replaced by the assembler code **LA 1,0 / MC 37**. Obviously, in using DATE, a program is more readable than when it uses the two assembler instructions that do not at all indicate what the monitor call (MC) intends to do (i.e. obtain a date). Increased program readability is, in my opinion, the most important achievement of macroprocessors. But it must be that no one yet had any use for readability: macroprocessors were never used to achieve this goal on a large enough scale.

On the side of the macroprocessor: how does it know that DATE must be replaced by the assembler instructions as indicated? Two possibilities were offered. Either (a) the program could contain in a kind of prologue the information that DATE was once and for all equivalent to **LA 1,0 / MC 37** . Or (b) this fact was kept somewhere in the macroprocessor. The *replacement rule* was called **macro-definition**, and the symbolic statement DATE is the **macro-statement** or **macro-call**. Both the definition and the call were soon called just **macros** and the context would clarify whatever was meant. So, the macro-definition could reside in the program. This was not very practical: if many program texts needed DATE, each of them had to contain the definition. Thus, in most cases, the macroprocessor had to keep this information elsewhere. It was decided to store definitions in a **file** instead of in the processor itself, thus giving the possibility of creating new macro-definitions easily. The **macro-library** was also born.

More was needed however. Consider the example of a device management read operation. The *record* must be delivered into a program area, a so called *buffer*. In assembler language this could, for instance, be done as follows:

XR 0,0 / LA 1,BUFFER / MC 89

Now one could define a macro called READS, containing the **XR 0,0 / MC 89** sequence so that a program could do **LA 1,BUFFER / READS**. This however is annoying. Why indeed, was the LA instruction not part of the macro-definition? Because the name of the buffer is program-dependent, i.e. variable. In other words, if the macro-definition was to contain the LA instruction it needed the possibility to use any name a program might need for a buffer. The solution was to equip the macro-definition with a **parameter** corresponding to an **argument** in the macro-call. In order to allow this, the macro-definition needed a header statement in which the expected parameter was to be written. Thus, the macro-definition could be as follows:

READS MACRO BUFFERNAME / XR 0,0 / LA 1,BUFFERNAME / MC 89

The macro-call could then be **READS BUFFER**, so that the replacement would be

XR 0,0 / LA 1,BUFFER / MC 89

In other words, the program-specified name BUFFER was used in place of the name BUFFERNAME of the macro-definition. The name used in the program is the *argument* while the name used in the macro-definition header is the *parameter*. At the moment the macro gets replaced, the argument substitutes for the parameter. The interesting aspect is that the macro-definition does not even have to know what BUFFER actually means. It doesn't see it as a name in program context but merely as a **word** in macro-context. Therefore, if the parameter was the **name** of a **word**, then the argument in the program-based macro-call was supplying the **value** of the **word**. This was very new: a word that had a name and a value! Of course, in macroprocessor context, the value of a word was usually the name of some program field. In Fortran something comparable was known. Fortran had variables that had a name and a value. ABC was a name and 743 was a value. This is clear. Not so in macroprocessor context. The **string** ABC could just as well be the name of a word as the value of it. And that is where the macroprocessor creators made a mistake. They considered that only *arguments* were values of words, so that they only appeared in macro-calls, whereas names of words were always *parameters*, so that these only appeared in macro-definitions. Therefore, there did not seem to be any ambiguity. The macroprocessor creators did nothing to distinguish between a **text constant** (a value) and a **text variable** (a name). Unfortunately, macro-definitions could also contain text constants. Suppose indeed that register 0 is not called 0, but R0. In that case, our macro-definition must contain the line **XR R0,R0** and here we have a problem: indeed, how do we know that R0 stands for itself and is not a name in macro context? So, the macroprocessor creators decided to distinguish the cases and they particularized the **name** by giving it a prefix, the ampersand. The macro-definition for READS thus becomes:

READS MACRO &BUFFERNAME
XR R0,0 / LA R1,&BUFFERNAME / MC 89

The &names were soon called **macro-variables.** In summary therefore, the macroprocessor is a program that replaces a macro-call by the lines present in the macro-definition while replacing the macro-variables this definition contains by values taken from the macro-call arguments. Thus, the macroprocessor was the first string processing program ever made.

It was soon obvious that the macroprocessor need not be committed to assembler language but could create any text in fact. The assembler world however was not aware of the value of this fact, so that the macroprocessor's syntax was condemned to an assembler-like syntax and to the exclusive generation of assembler code lines. Exceptions were however tolerated via an auxiliary statement that could be used only within a macro-definition: PUNCH. This statement would not appear in the replaced code. However, it has the effect of producing some text onto a file and does this during macroprocessing itself. This was the first of another new being: the **macro-statement.** So, what does **PUNCH ABC** mean? Due to the same ambiguity as explained earlier, how do we know whether ABC means the three characters A,B, and C or is a name which stands for a value? The macroprocessor designers introduced yet another distinction: on macro-statements a string without ampersand was taken to mean a name, whereas when a constant was needed one had to quote it. Why these people did not obey their own & convention within macro-statements is sheer mystery. That is not the end of the story. In many cases system functions covered by a macro-definition allowed minute variations. For instance, one way to read from a device is given by the assembler sequence

LA 0,1 / LA 1,buffer / LA 2,key / MC 89

while another way is given by

XR 0,0 / LA 1,buffer / MC 89

Obviously we need two macro-definitions to cover these cases, one with only one parameter for *buffer* and the other with two parameters for *buffer* and *key*. But in fact those two macro-definitions were not so different. So, it was desirable to have only one macro-definition able to **generate** both sequences. This meant that decisions had to be taken within the macro-definition. To that effect another macro-statement was introduced: AIF. Also, an AGO macro-statement was added which allowed branching within the macro-definition body. In order to decide what to generate it must be possible within the macro-definition to find out whether an argument is given for the *key* parameter or not, since that is essentially the difference between the two possibilities. Therefore, the macroprocessor introduced a number of internal **attribute functions** allowing for instance to know whether an argument exists or not (in the example below that is the K' function). Let us now write the macro-definition for a general READ macro (R0,R1 and R2 are the *symbolic* names of registers 0,1,2):

```
READ        MACRO &BUFFERNAME,&KEY
            AIF    (K'&KEY=0).LAB1
            LA     R0,1
            AGO    .LAB2
.LAB1       ANOP
            XR     R0,R0
.LAB2       ANOP
            LA     R1,&BUFFERNAME
            AIF    (K'&KEY=0).LAB3
            LA     R2,&KEY
.LAB3       ANOP
            MC     89
```

In this macro-definition the first AIF tests whether the &KEY argument exists and if not it *branches* to the label LAB1. The macroprocessor only allowed macro-statements to have labels. For that reason, it was required to introduce a macro-statement with no effect, ANOP. Notice also that labels are prefixed by a dot. The syntax and semantics of AIF and AGO were remarkably close to those of Fortran and this is no surprise since both languages originated at the same moment. In considering our macro-definition more closely, we make some interesting observations. It contains two types of lines: macro-statements which serve to govern the macro replacement process and ready assembler instructions with or without &variables which are generated as a result of the macroprocessing. The second observation is that a macro-definition looks very much like a program. Suppose indeed that we introduce a new macro-statement which we will call AGEN and which will generate the value of its operand, with &variables replaced, into the original text. If it has more than one operand, these are *concatenated* (strung together). Using AGEN we can re-write our macro-definition:

```
READ        MACRO&BUFFERNAME,&KEY
            AIF    (K'&KEY=0).LAB1
            AGEN   ('LA R0,1')
            AGO    .LAB2
.LAB1       AGEN   ('XR R0,R0')
.LAB2       AGEN   ('LA R1,' ,&BUFFERNAME)
            AIF    (K'&KEY=0).LAB3
            AGEN   ('LA R2,' ,&KEY)
.LAB3       AGEN   ('MC 89')
```

What we have now is very clearly a program of which the actions are the generation of assembler text lines. Macro-definitions *are* **string processing programs**, embedded in a gigantic outer string which is the receiving text. What I mean is: *a text contains macro-calls which invoke a macroprocessor that reads another text (macro-definitions) and interprets them as programs that cause the macro-definition to be re-*

placed by text. A question is: if a macro-definition is a program, then where are its variables? Well, we already saw some of them: the parameters. But what is to prevent us from defining more variables for usage within the macro-statements? This was indeed offered by means of the LCL declarative macro-statement allowing the writer of a macro-definition to declare variables internal to the macro-text. **LCL ABC** defined the macro-variable ABC. Such variables could be given a value by means of the SET macro-statement which had an incredibly garbled syntax: **ABC SET 'XYZ'** gives the value XYZ to the macro-variable ABC.

The Fortran spawn

Without realizing it, the oldies had given birth to a first construct that would become important in the structuring of programs. In fact, assembler had created the idea, but Fortran really formalized it: here comes the **subroutine**. The subroutine resulted from the consideration that writing a same piece of code more than once in a program text was uneconomical. In assembler, this situation could be cleaned up in two ways. One could create a macro-definition to generate the code, and use macro-calls at all places where this code was needed. A solution that was satisfying as far as the program text went, but in translated binary code, the whole thing was still present more than once. The advantage was however that the code had to be written only once, so that it could also be maintained in a coherent way. Another solution was to isolate the code in the program text, and wherever it was needed one could then **invoke** it. To that effect, IBM360 assembler gave the BAL (and BALR) instruction which branches to a certain address (its operand) and stores in a register *the address following the BAL*. By branching to that register, a *return* to the mainline coding could be achieved. The isolated piece of code was called **routine** or **subroutine**. Assembler did not force any discipline upon this new being. The piece of program that issued the BAL instruction was the **caller**. The relationship between caller and subroutine can be depicted as follows:

```
caller line                   subroutine line
...                                  ...
BAL R14,ABC                   ABC    first instruction
next instruction                     ...
...                                  ...
BAL R14,ABC                          BR R14  return
next instruction
...
```

An immediate extension of the subroutine was the **module**. A subroutine could be assembled separately (i.e. as an independent piece of program), and thus can constitute a separate entity, which is called module. The concept of subroutine raises a number of important questions such as:
(a) should a subroutine be technically and conceptually separate from the program text in which it is written? In assembler, one couldn't care less;
(b) Can a subroutine call another subroutine? The answer is yes, but in assembler it is up to the programmer to make sure that no confusion arises;
(c) Can a subroutine have "variables" of its own? In assembler, all variables are accessible by the whole program, therefore, there is no question of ownership. Assembler language tried to introduce some protection there by defining addressing ranges

(supported by the USING instruction), but this could be breached at any moment;
(d) Is a subroutine clearly delimited, i.e. does it have a **first** and a **last** instruction? In assembler, the answer is: not really. Subroutine texts can overlap, since the BAL(R) instruction branches to any valid label.
As a conclusion, assembler subroutines are not at all formalized. Programmers could (and did) misuse the subroutine in various devilishly tricked ways. Program readability decreased immensely.

Fortran, on the other hand, did all it could to preserve and formalize the subroutine. The subroutine is a piece of code, written in the program line. It is however clearly delimited by a SUBROUTINE and an END statement. The SUBROUTINE statement gives a name to the subroutine. The subroutine has to be invoked by means of its name in a CALL statement, and there is no other way to get at it. In other words, program execution flow will never enter a subroutine spontaneously. The subroutine can only be entered at its top (although later versions of Fortran would allow secondary entry points). Returning to the caller was to be done by means of a RETURN statement or by reaching the END statement. The subroutine body cannot access variables of the calling line. If it uses names already defined in the main line, this is still taken to mean another variable, local to the subroutine; accidents cannot happen. Conversely, the calling line cannot access any variables within the subroutine. Although a subroutine could not contain another subroutine, it could call any other subroutine of the same program. Subroutines could also be compiled separately. All this was nice and clean, and certainly gave programmers a well formed way to structure their programs. In fact, subroutines offered an important new philosophy: **procedural abstraction**, a mental process by which detailed code is replaced by a *name* that stands for the functionality of that code. There was more to it, though. Subroutines offered the all-important feature of parameters and arguments, in the same vein as what we have seen in the macroprocessor. The SUBROUTINE heading can contain a list of names, which are *parameters*, and the CALL statement must then contain a list of values, which are *arguments*. Arguments and parameters correspond one to one. Thus, if we have a subroutine **SUBROUTINE XYZ (K,L,M)** and a call **CALL XYZ (A,B,C)**, then, in the course of execution of the subroutine XYZ, the parameters K,L,M have the *values* of the mainline variables A,B,C. Fortran achieves this by first making a copy of the values of A,B,C into work fields. The parameters K,L,M are then sited over these copies. In other words, Fortran parameters have a storage piece of their own. Many subroutines need to give back values to their callers; this is done by the inverse mechanism: whenever a RETURN (or END) is executed, the copies of the arguments (i.e. the parameters, possibly modified by the subroutine's execution) are copied back onto the original variables. This looks acceptable, but contains danger. What indeed if we have a subroutine

SUBROUTINE XYZ(X) / X=4 / END

and call it by **CALL XYZ(3)**? The result of this situation is that the constant 3 is replaced by the constant 4. A nasty side effect. It would take Algol to clean this one up... As an important first, Fortran also allows parameters to be subroutine names. Consider the following example:

```
mainline              subroutine1               subroutine2
A=5                   SUBROUTINE ADD(X,Y,Z)     SUBROUTINE FCT(X,Y,Z,OP)
B=7                   Z=X+Y                     CALL OP(X,Y,Z)
CALL FCT(A,B,R,ADD)   END                       END
```

The call to FCT in the mainline gives the value *ADD* to OP, so that the call to OP in subroutine2 actually executes as a CALL ADD(...). For such cases to work correctly, Fortran wants the programmer to declare the ADD subroutine by means of the statement **EXTERNAL ADD.** A powerful mechanism had been created here, but it did not seem to have a huge success with programmers. Nevertheless, other languages such as LISP were going to continue in that line. Another aspect was that of passing *labels* to a subroutine. The subroutine could use those label parameters to return to some specific point of the calling program. Of all the nasty things this was really the worst! It allowed programs to become very abstruse, illegible and incomprehensible. Even the syntax was weird: in the call statement, labels needed a "&" prefix; in the subroutine header, label parameters had to be written as asterisks and had to be at the end of the list. Fortran implicitly numbered the asterisks from one upwards and the RETURN statement could use these implicit numbers. As an example consider a subroutine **SUBROUTINE SUBR (A,B,C,*,*)** and a call **CALL SUBR (X,Y,Z,&10,&20)**, then **RETURN 1** in the subroutine would actually go to the first asterisk, i.e. to label 10 of the mainline, and **RETURN 2** would go to the second asterisk, i.e. to label 20 of the mainline. A RETURN without a number will return to just after the subroutine call as usual.

A very essential new feature in Fortran was the **function subroutine**, or more briefly **function.** Such functions were subroutines but were special in the sense that the subroutine call had to be part of an *expression* and replaced itself by a computed value on return. This allowed for very powerful formulations. As an example consider the following subroutine:

FUNCTION MULT(X,Y) / MULT=X*Y / END

This can be used in a program as follows: **X=A+MULT(7,B)**. The statement **MULT=X*Y** in the subroutine assigns a value to the *function name*, and this is the value that replaces the call in the mainline. As far as terminology goes, one does not say that a function is *called*, but rather that it is *invoked*. Allowing for functions was certainly one of the most essential features introduced by Fortran. A whole new world opened, which was going to culminate, much later, in *functional languages*.

Chapter 2
THE YEARS OF SYSTEM AND COBOL

Towards the end of the 50's things became more complicated and Cobol hit the scene.

Monitor leaves the scene, System makes it

As Monitor kept growing, a number of new needs had risen. One of these was **interface calling.** What was that all about? Monitors grew bigger; they offered more functions such as device management. Monitors very soon became **systems.** And systems would be defined as the set of such functions, offered for better or worse. One might imagine that any one function was *implemented* as one (system) program. And in the beginning that was indeed the way it was done. Such a program, as part of a system, was called a **module.** A system was nothing other than a **set** of modules. Almost immediately it became clear that modules were going to need the functionality of other (co-resident) modules. They were going to **call** one another, so as to avoid repeated programming effort. But, a module called by another module, almost certainly had never been created to be called in such a manner. The functionality it appeared to offer was a fortunate by-product of the actions it performed. Moreover, the notion of **environment** started to appear. The machine had *registers*, the modules had *data.* The modules had to be able to preserve these objects when calling one another, otherwise the return would not be possible. And also the return point had to be remembered ... Part of the data was to be passed back. This set of **rules** constituted the **interface between two modules.** And that is where the problem suddenly appeared. The interface was particular to each **pair** of modules. Therefore, if n modules composed the system, and if calls were restricted to the obvious cases, there were $n(n\text{-}1)/2$ interfaces possible. It was soon admitted that system complexity, for increasing n, grew like n^2. Obviously, if a module had not been developed to receive a call from *one* caller, it certainly was not apt to accept calls from *many* ... The module was a hermit. When it was used on a call basis by its neighbours, it started losing grip and programmers tended to garble the module's code. The whole thing was heading towards catastrophe. Modules grew numerous **entry points**, for slightly different interfaces but for almost the same functionality. Modules behaved slightly differently for different callers. They became caller-anxious. And no one attempted to solve the situation. Programmers, if anything, started to cherish the complexity (a common streak in a specialized field), did all they could to protect it, to avoid simplicity. It made them indispensable. It called for tricks, not for skills. It made objective evaluation of the programming effort impossible. There was something rotten in that kingdom. A new invention, which claimed to be the solution, and indeed had possibilities, was immediately misused: **modular programming.** Modular programming came from the trivial observation that if P programs can associate to form a module, then a module can dissociate into P programs. Modular programming claimed that doing just this would clarify the programs since they became smaller and it would also alleviate the interface problems because of possible program (module) regrouping, at least in the long run. It was a first, timid, attempt at structured programming. But something very different happened. Modular programming had said that a system module A, receiving a number of calls and issuing another number of calls could be *fractioned* into submodules, say A1, A2 and A3, where A1 would receive the calls that were originally destined for A and all calls that A had to issue would come from somewhere within the submodules A2 and A3. The calls from A1 to A2 and A2 to A3 could be **standardized** as they were a purely internal and intentional situation. That is where it went wrong. Modular programming increased the number of modules, and programmers soon appreciated the **granular** functionality this brought. They issued calls to the submodules as well. The complexity of systems grew dramatically. No solution was yet in sight.

The overlay fanaticism

The formalization of subroutines brought new life to the concept of modular programming. Indeed, a module was a subroutine, or wasn't it? The step towards saying that *therefore a subroutine is a module* was very easily taken. A dangerous step this was ... But it all came about in such a pompous and emphatic way that everyone acclaimed it as the long-expected rescuer. Gradually, machines had become too small to hold a program's object code (another name for binary executable code). Machines of those days typically had 32 to 64 Kbytes of physical storage. And part of that was devoted to the *resident* system portion. Thus, not enough memory was left for the Fortran (later: Cobol) programs. Making programs smaller was certainly called for, and compilers did their best to achieve that goal. Also assembler programmers did their utmost, causing the appearance of a new breed of programmer: the bit splitter (that was the true specialist ... at a higher level we found the byte crunchers ...). Noble causes these were (or so one believed). But there is a limit to everything. So there were still programs that could not possibly fit in memory. The solution was, obviously, modular programming. It was argued that a program did not need to reside in core as a whole, but could rather be fetched into core piece by piece, as need arose. It remained to define the *piece*. It was called a **segment**. As a consequence we needed ways to segment programs. This was achieved by decomposing the programs into **control sections** (CSECTs) and by defining the segment as a sequence of such CSECTs. Now, a CSECT is an arbitrary grouping of statements, with some rules governing the addressing of data fields inside or outside the CSECT's text. The assembler programmer was free in this respect. Fortran (and later Cobol) however, defined CSECT by convention: a main program, a subroutine (or a Cobol segment, something that was to appear only much later). Suppose a program is composed of 8 consecutive CSECTs (this concerns the writing order, not the execution flow!) named A,B,C,D,E,F,G,H. We then could set up a plan for loading these CSECTs into memory, by defining **loadable segments** and by specifying how these can **overlay** one another in core. Consider the tree structure of *figure 1.* The tree contains 6 loadable

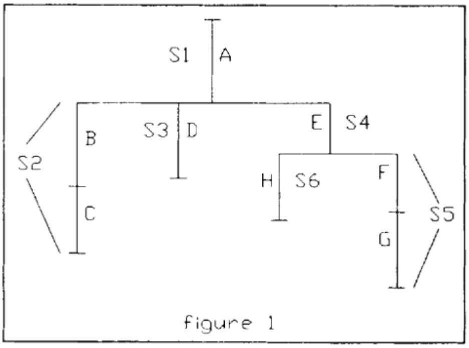


figure 1

segments (S1 to S6) which contain CSECTs and it also indicates an overlay policy: the **loader** (i.e. the system component that actually *fetches* objects into core) will start by loading only segment S1 (i.e. CSECT A). This is the **root segment**, which will remain in core at all times. Next, the loader will load only one of the segments S2 (CSECTs B and C) or S3 (CSECT D) or S4 (CSECT E). These segments can overlay one another on request basis, and it *is up to the program logic to make sure that this gets done in a safe way*, since the segment the loader overlays upon, is lost (well, it can be loaded again of course, but that is a fresh copy, with data fields in an initial state). In other words, the program's logical structure has to be bent into something that allows overlay to take place. A lot of features were brought into the overlay picture, so that CSECTs could be forced to be physically loaded in some segment even if they were actually written as source in another one. You see, when a segment gets overloaded, data is lost, without any hope of restoring it. This means that data one still needed after an overlay, has to be placed in segments of higher level in the overlay tree. The concept of **region** was also introduced, allowing independent overlay activity in different portions of physical core. An example is given in *figure 2.* The handling of segments S6 and S5 is unaffected by that of either S2 or S3 or S4. The programmer had two ways of *provoking* the actual overlay activity: either by demanding that a segment be loaded (by means of the *brute force* LPOV macro), or by *calling* a CSECT within another segment than the one currently loaded (done by the more *gentle* XCALL macro), a process which is different to the calling of a CSECT within the currently loaded segment. Tricks ... The result of it all was that a new specialist emerged: the overlay coder. Fanatics that considered it a sport to force big programs into very small core slots. *Rule by exception* was their motto. Incredible but true. On the other hand, overlay was the only way to get big programs to work. It was going to take very many years before overlay would evolve into more noble techniques ...

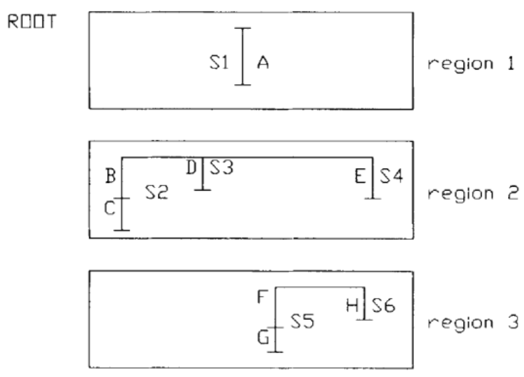


figure 2

What's in a name?

It is time now to bring up an important question. What indeed does a name stand for? This may sound like an obvious question but it isn't. *John Done* clearly designates a person. The name stands for the object resulting from the conglomerate of fair hair, brown eyes, bright complexion, slight scar on left cheek, big mouth, athletic arms, large torso, small hips, short legs that make up the *Done* person. This is rather obvious. Let us rule out once and for all the deviation created by duplicate names: there is no other John Done. Thus, the conglomerate (the person) described above, is the **value** of *Done*. Exactly what a Fortran variable is and an Assembler variable is not. But that is not all there is to it. In the early 50's Korzybsky claimed that Aristotelian logic was not a good way to describe reality. True logic was non-Aristotelian, he said. This statement was the foundation of general semantics. In Science and Sanity (1958), Korzybsky indicated that an object could be at the same time in the set A and in the set non-A (Aristotle had said A or non-A, but not both), from which followed that if $a=b$ and $b=c$, then $a=c$ and $a \neq c$ are both possible. Let us consider an example: is a cat a cat? Aristotle said yes, and he was right if the word cat really means the sleek animal we all know. Korzybsky said no, and he is just as right if the word cat does not always mean the animal. It amounts to saying that the cat is not necessarily a cat. During World War II, a general having been informed that taking the town X cost 15.000 casualties, said it was a lot to pay for a *mere name on the map*. Applying these views to *John Done* leads to the following statements: [*Done* is a (proper) noun], [*Done* is the person described as a short fair-haired athlete], [*Done* is the group of four letters D, O, N, E], [*Done* is the past participle of the verb *to do*]. The fourth possibility is purely incidental but the three first possibilities **always** exist. It is not all that simple anymore. Indeed, what do we mean when we use the word *done*?

What is the link with names in Data Processing? A name designates an object. In Assembler language the object itself comes to live only by its usage type. This is not true for Fortran nor for the macroprocessor. In these languages an object is clear by its mere definition. It is a variable. A variable has a value. It also has an address (i.e. it is located somewhere). Thus, a name stands for both aspects of the variable: value and address. And it also stands for itself as a group of characters. Note that a variable (a name) must have an address (must be sited) if it is ever to hold a value. Somehow one sees the address as a rather fixed object, while the value is more volatile. Better: a variable is important by its value and much less by its address. But what about the name of a parameter of a subroutine? The parameter is only a place-holder for the argument that it really is at the time of execution. Could one say that the parameter's name is an alias for the argument? Certainly one could. It would mean that the parameter is a name which has the same address and value as the corresponding argument when the subroutine containing the parameter is active. The parameter has neither address nor value when the subroutine is not active. This however creates technical problems in a compiled language: all statements using the parameter must be resolved at compile time. Technically therefore, the parameter is a named object having an address at all times. This location will contain a value only when the subroutine is active. The value will be the address of the argument. What we now have is the parameter as a truly existing object, no longer alias of the argument, but a *reference* to it. This kind of argument-parameter linkage is called **by reference**. Another way of seeing a parameter is by conceiving it as a *copy* of the argument. This means that there is no strict identity anymore between argument and parameter. If the parameter's (i.e. the copy's) value is changed, that of the argument stays unmodified. This kind of argument-parameter linkage is called **by value**. We have seen that Fortran uses a mixture of both policies: it creates the copy but moves it back onto the argument after the subroutine has terminated. Apparently, compiler makers had the choice of the policy. It was however not clear at all times. Indeed, what if an argument is an expression such as **a + b**? What is an expression? It is certainly an object, but has no name. Whenever the expression is written down somewhere it is immediately evaluated and is replaced by its value. Thus, an expression is an unnamed object with a value (and, of course, an address to hold the value). The value (and probably also the address) is essentially volatile. The expression can certainly not be assigned a value by means of an assignment statement. When an expression gets used as an argument, a link by reference is rather meaningless; in fact, it means exactly the same as a link by value. However, an expression can also be seen in another way: the expression's syntactic form can be taken as its name! And why not? Whenever this *name* appears anywhere, it stands for the computed value of it. Under this philosophy, an expression argument can be linked to its parameter by reference. Each time the parameter is used, this causes evaluation of the expression the parameter actually stands for. Passing by value, in this case, means the expression is evaluated just before entering the subroutine, and this value is passed once and for all. Various languages such as Algol and PL1 would have various policies in this area.

This situation is more or less the same for **constants.** What is a constant? It is an object that remains constant. It is also an object whose name is its value. Such objects are said to be **self-defining.** Any reference to a constant reproduces its value, which therefore cannot be changed. A constant cannot be assigned to. There is a major difference however between a constant and an expression. A constant's value is permanent while an expression's value is essentially volatile. The constant is sited once and for all (and in many cases it is not sited at all, because the binary instructions can contain constants within themselves). What happens when a constant is used as an argument? When it is linked by reference, it stands for itself. We have seen in Fortran that this could cause modification of a constant's value! When it passes by value, it is a copy of the constant's value that gets passed. Some authors have introduced the **named constant.** This is no longer a self-defining object, since it has an explicit name and an explicit, unmodifiable, value. Such constants are truly sited, but have the same behaviour as unnamed constants. They allow us to give the name *three* to the value *3*, and this may greatly improve the readability of our programs. On the other hand, we cannot distinguish such constants from variables anymore. Unnamed constants nicely indicated what they were, either because they were plain numbers (including a sign and a decimal point) or they were string constants, in which case they had to be surrounded by **quotes.** Thus, *'abc'* is a constant, whereas *abc* is the name of a variable or constant.

There are even stranger objects

Fortran had introduced another new object: the **array.** In fact, an array is a collection of many values which are somehow ordered. The ordering is a numbering: each value in an array is numbered by its *coordinates.* In other words, the values can be organized in a *line*, in which case they are numbered 1,2,3,... (or any equivalent scheme). The values can also be organized in a rectangle, in which case we speak of **rows** and **columns,** and the numbering is *two-dimensional,* e.g. (1,1),(1,2),...,(4,1),(4,2). Generally speaking, arrays can be n-dimensional. For arrays, the name stands for the conglomerate of all the values. The address is taken as that of the first element. But this does not suffice. The array object needs other **attributes** in its description: not only do we need to know how many values the array contains, but also how they are organized. In other words, we need to know how many **dimensions** and the numbering **bounds** in each dimension the array has. Compiler makers have always had problems in allowing arrays as arguments to subroutines. The problems were technical, not conceptual, however. Thus there is nothing that prevents us from passing an array to a subroutine, either by reference or by value. Most languages, however, do not or only partially cover these needs. A problem that is specific to conglomerates is that they are composed of lower level beings. An array contains **elements.** But it also contains rows and columns (this is generalized to the notion of **cross section** for n-dimensioned arrays). How do we *identify* those objects? What is their name? Obviously, part of their name must be that of the containing structure, i.e. of the array itself. The name must be further extended by the coordinates of the element (or the cross section). This is done by means of **indices.** In a two-dimensional array A, the notation (the name) A(3,4) designates the element at the crossing of row 3 and column 4. Such an element is called **scalar.** The name A(*,4) stands for the cross section composed of all values in column 4. This is itself an array, with only one dimension. Other authors have used the notation **isub(A,3,4)** to designate an array element. This is a strange notation, as we will see later. The important thing to remember for now is that A(x,y) acts as a name, which is not unlike an expression. Indeed, indices can be expressions, so that the array element coordinates are actually computed.

Subroutines and functions: what's in a name?

There are still other objects that have a name: subroutines and functions. And we have seen that Fortran allows us to pass them as arguments to subroutines. So what does the name of a subroutine stand for? Let us first consider non-functional subroutines. Does the name of such a subroutine have a value? Yes, of course. The value is the subroutine text itself (or better: its compiled equivalent). And what is its site? The first address in core where the executable code of the subroutine starts (the *entry point*). Does this mean that the name of a subroutine stands for a variable? Can we assign something to the name of a subroutine? If we can, then the assignment a = b where a and b are subroutine names, would *replace* the text (i.e. the code) of subroutine a by that of subroutine b. This however, is not what compiler makers have done. They have considered that the name of a subroutine is in fact the name of a constant, so that one cannot assign to such a name. Passing the name of a subroutine as an argument therefore cannot happen by reference. It only happens by value. Moreover, the actual value that gets passed is the address of the subroutine's code. This being said, there is a certain amount of ambiguity regarding the name of a subroutine. According to the way in which this name is used, the interpretation is different. When used as an argument, we mean the subroutine as a piece of code. However, when used in the call statement we actually mean execution of the subroutine.

As far as functional subroutines are concerned, we can see them in the same way as the other subroutines. But, since a functional subroutine's invocation can take the place of an expression, it is more usual to see the functional subroutine as a named expression with exactly the same semantics. This interpretation is true whenever a function is invoked, but it is the other interpretation that holds true when the name of a function is passed as an argument.

Let us come back to the assignment of one subroutine to another. If we allow an object to be a subroutine **without** text, then it must be possible to assign a subroutine text to such an object. What we have now defined is a *subroutine variable* as opposed to a *subroutine constant*. For technical reasons such an assignment is never done by moving the text or code, but rather by saying that in the statement a = b we actually mean that a is another name for b. Technically the value of a is not the code of b, but the address of the code of b. It appears that assignment can take place either by moving values, or by moving addresses of values.

A last question is: can there be an unnamed, self-defining, subroutine constant? Could we, for instance, do **A='T=M*2 X=T+M*3'** where A is a subroutine variable? Why not, provided we also *add the definition of the parameters*? In fact, some macroprocessors and also the LISP and PROLOG languages allow this. It is not at all common practice in current programming languages however (apart from the short-lived and controversial Algol68).

Those magnificent men with their sparkling Cobol

Meanwhile the world had not stood still. Dijkstra once said that Fortran was an infantile disease. What was to come next was catastrophic! Cobol was its name. A language meant for business application programmers. A language intended to be the epitome of all languages. A totally naive approach to writing programs in terms readable by human beings. The conceptors of Cobol decided to replace all operations by words and to rephrase the lot. A simple Fortran statement like **X=A+B** was replaced by **ADD A TO B GIVING X.** Many such verbs and phrases were spawned and the result was that although indeed *readable*, Cobol became very hard to master. As a result, this induced the appearance of fanatics of the *long live Cobol* species. They promoted the sophism that Cobol was easy. This misconception, added to the fact that the language was truly general-purpose and the sole available one, is the cause of the overwhelming enthusiasm that engulfed the application world and still exists today. Cobol kept some of the finer points of Fortran: it is a variable-based language and it wants all variables to be pre-defined. This of course is nice. Cobol also kept the conditional statement of Fortran and replaced numeric labels by more human-looking **paragraph** names. In inventing paragraphs, Cobol was the first language to offer program structuring. But it did more: it required the program to be divided into four **divisions** thus isolating the *identification*, the *environment* and the *data* of the program from its executable statements, the *procedure* division. Although the idea was not bad, the implementation left a lot to be desired: each division has different syntax rules; division contents are extremely verbose; each division can be present only once, thus not allowing the writing of a subroutine within a program. The environment and data divisions are further divided in fixed **sections** with very strict syntax and contents requirements. The claimed reason for it all was: documentation value and consistency of non-procedural declaratives. The procedure division itself can be divided freely into paragraphs (and in later versions also into **sections** and **segments**). What (if anything) is a paragraph? In reality a paragraph is a sequence of statements extending from the paragraph name up to, and not including, the name of the next paragraph (or to the end of the text). In essence, the paragraph name is a mere label one can branch to. Thus, the paragraph is the target of a GOTO statement. This means that the paragraph is not of very high semantic value *by itself.* But Cobol had a bag of tricks: it introduced another *connective*, a purely paragraph-bound one, the **PERFORM** statement. This statement takes a paragraph name as an operand and merely indicates that the paragraph must be executed completely after which control must come back to the statement following the PERFORM statement.

Somehow, this looks like calling a subroutine without parameters. However, it was much less clean than that:

(a) a performed paragraph is part of the procedure division and uses exactly the same variables as the program;

(b) a performed paragraph cannot be *abstracted* since it can have no parameters;

(c) a paragraph can also be branched to, in which case it does not return;

(d) a paragraph can also be flowed into sequentially in which case it has no special meaning;

(e) within a performed paragraph there can be other PERFORM statements or branches to other paragraphs: what happens then is rather involved.

The last point is the real troublemaker: it makes code truly unreadable. In Cobol the program flow did not at all follow the writing sequence anymore. People found it necessary to document the flow by using flow-charts. This mere fact indicates how badly adapted Cobol was. Last but not least, the Cobol makers concocted the **ALTER** statement. A very doubtful feature which allowed programmers to modify the interpretation of the target on a GOTO statement.

Cobol alternates

So Cobol had invented the not so very usual notion of paragraph. Nevertheless, this was an improvement, notwithstanding its awful semantics. Cobol did not stop there: it invented something else, something of which it certainly did not immediately realize the implications. Fortran had an IF statement which allowed the conditional execution or skipping of an action. Cobol used the same type of IF statement. In order to skip a group of actions, one could write **IF condition GOTO label.** But Cobol also allowed the IF statement to contain executable statements, for example: **IF condition ADD A TO B GIVING X.** The next step was to bring many statements under the rule of the IF. This could have been done by using the paragraph concept. Cobol however decided otherwise. The normal termination of a statement in Cobol is the period (•). When many statements are grouped under an IF, only the last one may have a period. The intermediate statements are separated by a mere blank or, if one so wishes, by a comma. So, the period was not really the *statement terminator* but rather the *action terminator*. Incidentally, the various uses of the period in divisions other than the procedure division is totally bewildering! Nevertheless, it was a nicety of Cobol that it invented the truly conditional statement, something Fortran did not have. An immediate consequence of this is that we don't need a label anymore for skipping and that we also have the possibility of *choosing* between two actions. If we have to execute either action A or action B depending on a condition being true or false, we can write **IF condition actionA ELSE actionB.** This structure is a true alternative. Its possibilities are enormous but in the Cobol world they were not that well used. Many Cobol programmers tend to avoid the ELSE part and prefer to do some branching instead. Moreover, something was lacking in the expression of the *condition*. Cobol allows a condition to be a comparison for equality and various inequalities. It does not really allow conditional *expressions* . In particular, boolean expressions cannot be written. On the other hand however, Cobol introduced testing for set inclusion and exclusion: the two tests $\mathbf{A} \in \{\mathbf{X,Y,..,Z}\}$ and $\mathbf{A} \notin \{\mathbf{X,Y,...,Z}\}$ can be written

A = X OR Y OR ... OR Z and **A ≠ X AND Y AND ... AND Z**

It is unfortunate that the conjunctions **and** and **or** were used in this context since they do not have their usual boolean meaning. Another nicety is that IF statements can be nested: the statement executed by an IF or by the ELSE can be another IF. In order to avoid ambiguities, each ELSE part was considered to belong to the nearest IF without ELSE. This introduces the requirement to write ELSEs on nested IFs even if there is no action needed. In this case, Cobol wants the programmer to code the standard statement **NEXT SENTENCE.** As an example, we want to execute actionA if $X \leq 3$ and actionB if $X > 3$ and $Y > 4$. We can write

IF X > 3 IF Y > 4 actionB ELSE NEXT SENTENCE ELSE actionA

Since Cobol cannot use AND in its boolean meaning, executing actionC if $x > 3$ and $y < 5$ and actionD otherwise, must be written:

IF X > 3 IF Y < 5 actionC ELSE actionD ELSE actionD

As a consequence actionD is written twice! Cobol programmers avoided this by splitting off actionD as a paragraph and PERFORMING it where needed:

IF X > 3 IF Y < 5 actionC ELSE PERFORM D ELSE PERFORM D

Because actionD is taken out of context, this gives it another *textual* weight, different to actionC. This is semantically unclean. Most programmers therefore will resort to branching:

```
IF X > 3 GOTO PAR1.
GOTO ACTD.
PAR1.
            IF Y < 5 actionC,GOTO PAR2.
ACTD.
            actionD.
PAR2.
            ...
```

Cobol tried to avoid boolean calculus. In doing so, it causes program writing to degenerate into extremely baroque and ad hoc solutions.

Cobol iterates

Cobol-based applications had a need for looping, in somewhat the same way as in Fortran, although usually Cobol loops contained input/output statements, rather than operations with a running variable. Of course, a Cobol program could achieve its loops (as in Fortran) by *branching back* on a given loop condition. However, the Cobol makers thought, since paragraphs had been created, why not use them for just the purpose of looping? As soon as said, as soon as done. The PERFORM statement received *variants* including the UNTIL and the VARYING clause, allowing the repeating of the named paragraph(s) any number of times. It is this feature, although at first sight quite appealing, that makes most Cobol programs totally unreadable. All the shortcomings of the paragraph are multiplied by the iteration feature. In particular, the mere fact that a paragraph may be "fallen" into is more than dangerous for program consistency. Cobolists will reply that programming discipline will avoid the traps. This may be so (although I doubt it), but what if the individual programmer breaks discipline? Is there any way of detecting it? On the other hand, the perform logic for loops has compelled Cobol programmers into the weirdest of tricks. Indeed, let us look somewhat closer at a loop . A loop is a sequence of actions (constituting its body), which are repeated for a given condition. We might represent it as in *figure 3*. The large circles are actions, and the black squares mere delimiters of the loop body. Within the body, there must be at least one action that changes the value of some variable, otherwise the loop body would have the same effect at each iteration step, making the step indistinguishable and thus forbidding any test for loop termination: the loop would become *endless*.

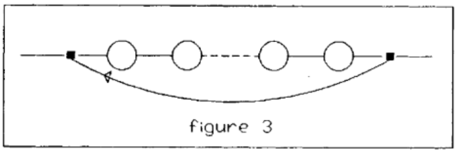


figure 3

The major question now is: where does the termination test take place? Cobol reply: in the PERFORM statement. Therefore, the loop is in reality represented as in *figure 4*. However, realistically speaking, it is true that the loop step condition could be tested before the step or after the step. This makes quite a difference as far as the first step (and thus the number of steps) is concerned. In other words, the test could be in the initial black dot or in the terminal one of the paragraph (or both...). Thus Cobol must clarify its PERFORM logic. *Figure 5* represents what it is all about. From this figure we observe that, the test is at the beginning of the loop step. As a consequence, if the test is to be done at the end of the body, there must be a **first** execution of the loop body **outside** the loop. This is generally done by doing a non-iterable PERFORM of the loop just before the iterable PERFORM. Let us generalize somewhat. Consider the case where the body of a loop contains some-

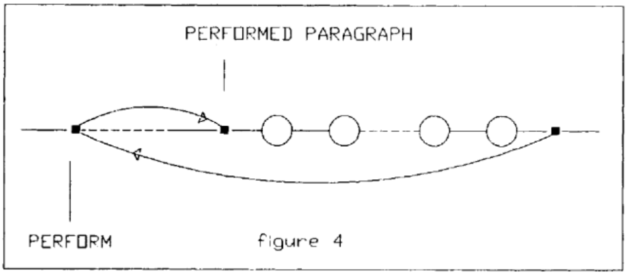


figure 4

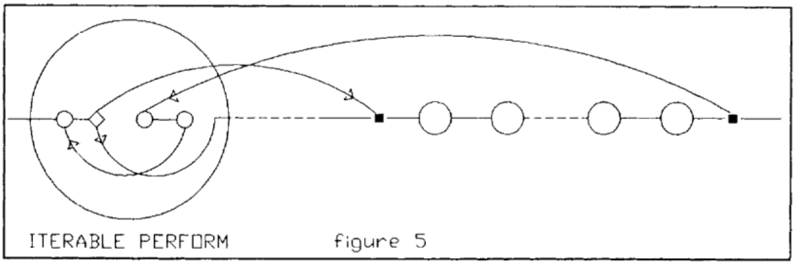


figure 5

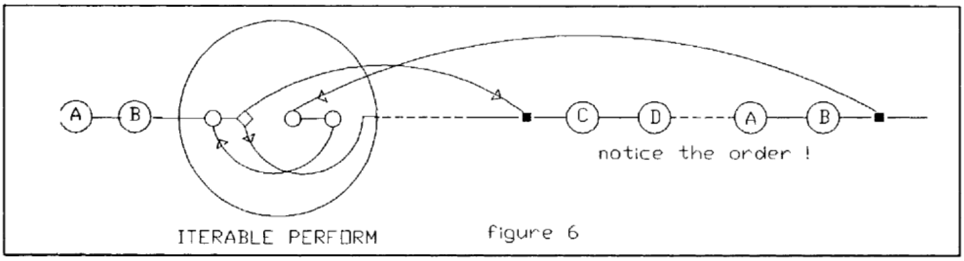


figure 6

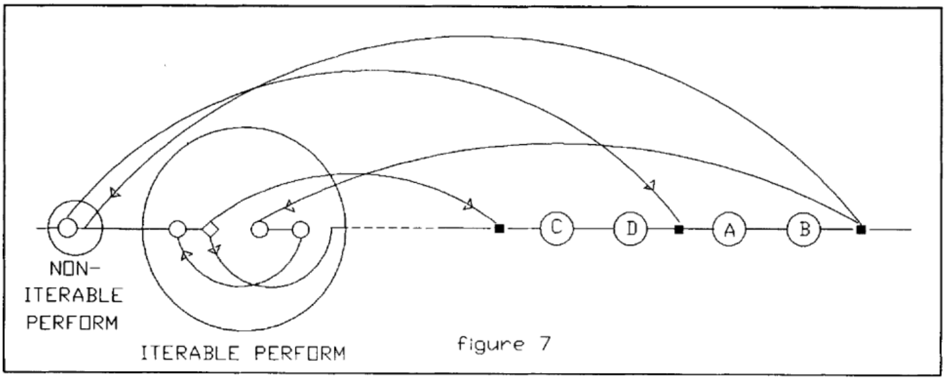


figure 7

where a *mandatory* termination condition which causes immediate end of the loop, a not uncommon situation. For Cobol, there is no obvious way out (with GOTO or something like it), so that the structure of *figure 6* comes into existence, or even that of *figure 7* which is much worse. These structures are daily bread for the Cobolist. One thing is clear: the semantics of the paragraph are *corrupted* by the existence of the PERFORM. Indeed, there are a number of ways to launch the execution of a paragraph:

(a) the paragraph may be *fallen into* by mere sequence of operation; in this case, when the last statement of the paragraph is executed, continuation is with the statement following the paragraph;

(b) the paragraph may be the target of a GOTO, in which case termination of the paragraph is as for the sequential case;

(c) the paragraph may be the object of a non-iterable PERFORM; in that case, when the paragraph ends, the control returns to *just after* the PERFORM;

(d) the paragraph may be the object of an iterable PERFORM, in that case, when the paragraph ends, return is to somewhere within the PERFORM and iteration may occur.

All paragraphs can be reached in these four ways. This means that the environment in which a paragraph is destined to operate is vague, to say the least. Error-prone is the qualifier I use to describe it. It also raises the question: what if a performed paragraph contains a GOTO to a place outside its text? Is that allowed? And if so, does it ever come back to the PERFORM statement? Possibly when it reaches the end of some (any?) paragraph it encounters during execution? Mystery. Bewilderment. And there is more: a PERFORM may name a list of paragraph names. This means that all paragraphs of the list are executed in sequence, and only the last one returns to the PERFORM. Of course, any of these paragraphs may be reached by GOTO as well. A variation of this type of PERFORM is **PERFORM name-1 THRU name-2** meaning: execute in sequence all paragraphs, *as they appear in the program text*, starting at name-1 and ending with name-2 (included). This sounds comfortable. It allows programmers to add paragraphs in the list, change their order, etc. Nice for maintaining the program ... But it is not comfortable at all: it is utterly dangerous! The result of the THRU notation is that in no way the PERFORM statement indicates what it is really intended for: it becomes context-dependent (maybe I should call this *abtext-dependency*!). Self-documenting they said about Cobol! There is worse to come: Cobol also allowed the following: **PERFORM name THRU EXIT** which means: go and perform at the named paragraph and go on in sequence, over paragraph boundaries, until an EXIT statement is met (by the way, this statement must be written in a paragraph that contains only the EXIT: so one can branch to it and achieve a *break* effect). And if one falls into an EXIT paragraph without being in a perform environment, then what? Mystery. The whole thing means that the paragraph is really a *two-headed* and *two-tailed* being. Depending on the way it is entered, there is a different way of getting out of it. Thus, at object code level, there are two (if not three!) entries into the paragraph, if only to set some flag allowing the decision of how to get out again. Due to the **THRU EXIT** form of the PERFORM, the paragraph is in fact a three-headed/three-tailed being, since, when entered in *that* fashion, the continuation after paragraph end is to the next paragraph. And don't believe that the whole thing is a matter purely internal, to be seen by the compiler only. The truth is that the paragraph is a being with a three-way split personality, a situation the programmer must take very seriously into account when writing his program. This appalling semantics fascinated programmers who used it in an almost mystic way to do things one would never have deemed possible. In those days, trickery hit the scene. Prestidigitation was more appraised than real craftsmanship. And it was only the beginning.

What's in a Cobol name?

Let us now come back to the concept of variable. As I already stated, a variable is an object with a definite meaning of its own, and a name that uniquely identifies it. The variable is able to receive a value (or has one unmodifiable value, in which case it is a named constant) and therefore it needs to be sited. Siting can be permanent or variable ... How does Cobol stand here? Something new (not that new: Fortran had it implicitly) was offered for variables: the **usage mode**. The variables now had an explicit **type**. Types existed in two major categories: numeric (COMPUTATIONAL) and non-numeric (DISPLAY), i.e. character strings. Finer types were offered within the numeric ones, and these were (and still are) all machine-dependent: floating, packed,... Along with type, some idea of precision and length was given. This resulted in the **picture clause** (essentially indicating the number of digits or bytes, decimal comma position, sign ...). As from Cobol, a variable has the following *attributes*: a name, a type, a precision, a value, a site. The type implied the operations that a variable could undergo. So far, so good. But Cobol offered more. The intention was to formalize the notion of record (from a file), or in fact any *user form*. Such a record is an agglomerate of Data. Fortran had already introduced the array as a dimensioned agglomerate (i.e. a variable with one name and many simultaneously existing ordered occurrences of values). Cobol wanted more. It wanted the agglomerate to be made up of a fixed list of **named** variables. The **structure** was born. A major breakthrough that was! All subsequent languages would contain it, with semantics almost exactly copied from Cobol. Via Cobol, the structure was synonymous with a record: the operations it could undergo were essentially read/write operations of the file management system. But, the structure was also a way to regroup data into objects with a well-defined semantic profile, objects that could be moved by means of the assignment statement, which in Cobol was called MOVE. Moreover, the structure could be made up from other (sub)structures, to any depth (well: Cobol set some limit there, but it was high enough). The structure is only one way of representing data. The greatest advantage of the Cobol structure is its fixed layout. But this fixed layout prevents it from representing more dynamic data types. At the time Cobol's coming about (early 60's), there were no other needs though. In other words, the structure was a *recursive refinement* of the variable. As a consequence, a name could now stand for a list of names, of which each name could also mean a list of names and so on. And still Cobol did more. First of all, it allowed for two or more variables to have the same site. Formally, this was already an established possibility for the parameter-argument pair in a subroutine call. Cobol extended on this by allowing any variable to be a **redefinition** of any other variable. Fortunately, this was for usage only within a structure. *Redefinition* in Cobol means that two variables will have exactly the **same site**. But that would be none other than creating a mere alias? Not so: the redefinition could span variables of different types ... In the vast majority of cases, this feature would be used to allow actions upon data that Cobol would otherwise forbid. Breach of discipline! The common usage of this feature was to allow string handling. But Cobol really offered it for another reason: since a structure represented a record, and since the contents of a record could be of different layout for each (of a group of) occurrences, the redefinition was a must. The major question was: why should there be such unpredictable records? (Incidentally this question has been investigated more formally, and the consequence has been the data-base. Another story, told in chapter 15). In the early 60's such wild records existed, and programmers had to cope with them... It was not all Cobol's fault. In fact Cobol had the merit of at least trying to discipline the weird cases. Unfortunately, misusage immediately appeared... Another new situation Cobol introduced was the *88-level*. A remarkable typographic shortcut. In fact, the 88-level was a list of names of values that a variable (part of a structure) could take, and with the mere intention of making the testing of such a variable more readable, thus more significant. In fact, the 88-level was a mere **compiler trick**, and came rather close to a macroprocessor variable name, for substitution during compilation but with no effect (**nor existence**) at program execution time. It was not a bad thing, as it increased program readability, but it confused two semantic environments. On the other hand, it could have been extended much further (e.g. value validation). Apparently, Cobol was somewhat unclear in its naming formalisms. This appears very much in its writing scheme: non-structure variables must have a level number 77! Names of values must have a level number 88! Moreover, the introduction of type for variables increased the semantic possibilities, but also caused them to become complicated. Indeed, in considering the two questions: (a) what operations are possible on which data types? (a non-trivial question, since Cobol has some 100 operations upon variables); (b) do data-types convert into one another? (important since Cobol programs do a lot of editing of paper listings for human consumption), Cobol has preferred to state the answer, operation per operation, thus creating the highest set of semantic rules ever (and thirty years of Cobol have made it even worse!). It was a wilderness. Trends in languages to come would aim at avoiding such a jungle. At the price of other difficulties though.

The conclusion is obvious

A Cobol program is certainly readable. But is it understandable? How future-friendly is such a program: can it be easily extended? Over twenty five years of experience with Cobol all over the world has given the answer: NO!! Nevertheless, Cobol still has the widest of acclaims: almost the whole application world uses Cobol on an exclusive basis. Why is that? Why don't programmers switch to more adequate means and techniques? In my experience, people find changes awkward because of the effort of changing, causing turbulence; the possible inability to cope with the novelty; some inertia: everybody uses Cobol; a misplaced feeling of challenge: mastering Cobol is the game; a fondness towards the cause of problems... On the other hand, for EDP managers, Cobol is comforting because Cobolists are readily available, a Cobolist needs no training (a true misconception!), all Cobol problems can be solved with discipline (another complete misconception!), Cobol, being a subset of English, the program can easily be transferred to other persons (the third misconception!), Cobol being ignored by all theoriticians there is no danger of seeing the formalism overwhelm the practice, there is no obsolescence of means, there are no theoretical fashions: the Cobol world is stable (this is true, but it is counterbalanced by the fact that so-called work groups introduce unbelievable idiosyncrasies in the language). The major question: *what is the price companies have paid for Cobol* has never really been answered, even if one is aware of the high cost of maintenance. Maintenance costs have universally been considered to be problem-bound and not method-bound (yet another misconception!), thus unavoidable. Well, unavoidable does not mean irreducible!

This being said, Cobol brought a number of important new ideas to the world of languages. It was the first language that tackled the business application realm; this brought the recognition of data structures and more types than Fortran had. Cobol also incorporated a more flexible file management syatem than Fortran. Clear separation of program and data (by means of divisions) is a sound idea (implemented in a non-uniform syntax). Cobol also introduced a full if-then-else construct and it generalized the loop construct. Furthermore, it created the notion of performable paragraph which is an interesting way to give structure to programs (unfortunately, it went off the rails). Without Cobol, there would have been no PL1 and Pascal would not have been able to introduce its record type. Without Cobol, file management and data-base management systems might not have become the strong systems they are nowadays. So, despite its shortcomings, Cobol is a language that has an important pioneering value.

Fortran, the infantile disease?

Cobol, apart from some good points, is not a very good language. But what about Fortran? Certainly, Fortran is a good *early* language. It is also the first high level language, one to which the Data Processing community owes a lot. Experience and positive criticism of Fortran have led directly to Algol60. Moreover, Backus' endeavours to provide a formal syntactic description of Fortran has led to BNF syntax (revised by Naur and used extensively for Algol60) on the one hand, and constituted one of the initial sparks that would start the theory of formal languages (by Chomsky) on the other hand.

But Fortran had also negative aspects. So, what was wrong? A number of things. Fortran introduced the *variable* with the *assignment* statement as the major action. This has set the philosophy of all later languages. Concurrently other computer scientists were inventing the *applicative* style (explained in chapter 14) which did not at all use assignments. Nowadays, this style is considered formally cleaner. However, due to Fortran's overwhelming success, the applicative style is still confined to only the academic circles. If Fortran deserves a criticism this is probably the most serious one! Apart from that, Fortran also had abusive implicit variable declaration and typing (type of a variable was REAL or INTEGER according to the first letter of its name, based upon a tacit convention of mathematics). The later attempt to create explicit types by allowing the specification of which initial characters corresponded to which type is totally ludicrous. Full explicit typing was enforced only for LOGICAL, DOUBLE PRECISION and COMPLEX (it was optional for REAL and INTEGER), but did not include the CHARACTER type, although this type was made available in input/output operations (via a FORMAT statement). Fortran showed a lack of uniformity in the usage of dimensioned variables (i.e. arrays); an element of an array (i.e. an indiced variable) cannot be used in all places where a non-dimensioned variable can. This causes the necessity to write more assignments and also more GOTO statements. There was also a confusing handling of labels, based upon the atrocious computed and assigned GOTO statements which tend to create hard to follow program flows. Especially nasty is the passage of label parameters to a subroutine (of course, one may wonder: is such a mechanism really needed?). The way of handling parameter passage, was rather conspicuous since it contained the possibility of destroying data. The later introduction of *by name* parameter linkage (see Algol in chapter 3) did not really clean up the mess. Another drawback of Fortran was the existence of the *storage classes* COMMON (allowing the sharing of data areas between separately compiled program portions) and EQUIVALENCE (instructing the compiler to assign the same memory addresses to different variables) with rather complicated semantics and usage (not further described in this book). These features induced programming tricks that were deemed *clever*, but are in fact heavily errorprone.

But let me state it again, in the 50's, FORTRAN really hit the scene. It had many good points, important firsts:

(1) the *expression*, both arithmetic and logical, allowing involved computations;

(2) the *iterated loop*, with a control variable and nesting, even though the construct had holes in it (present day computer scientists condemn that construct, but when it came about, it was much better than what existed in Assembler);

(3) the first successful attempt at setting up a machine-independent I/O system (a topic covered in chapter 15); it is not perfect and is limited to formatted I/O only, but it serves its purpose amazingly well; in fact no language has really done better (in the realm of formatted I/O);

(4) a formalization of the subroutine -especially the function- concept, allowing *procedural abstraction*; that was a real achievement;

(5) the pragmatic proof that a high-level language was possible, efficient and usable; in achieving this, Fortran became an essential milestone in computer science.

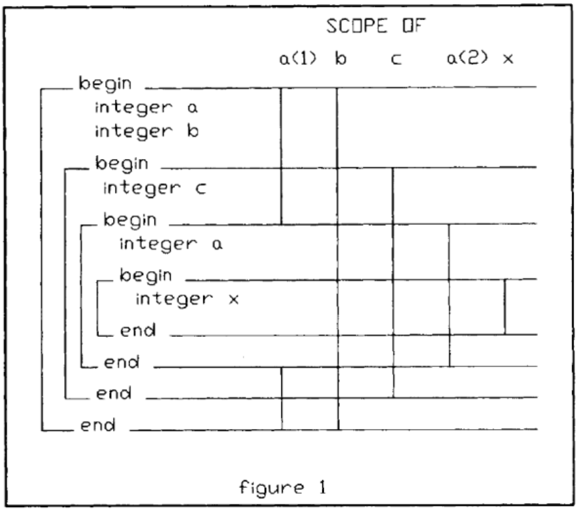
This page intentionally left blank

Chapter 3
THE YEARS OF ALGOL AND PL1

The roaring 60's with its line of new languages and constructs, including Algol and PL1 as well as the "what's in a name?" question.

Meanwhile, in some dark recess ...

While some were brewing, others were thinking. They were thinkers by nature, those first computer scientists... They knew what they wanted: programming (*program composition*) should not be allowed to degenerate. It had to become an art, reach heights heretofore unattained. It was reserved for those who had the true spirit and the good blood... Sunday programmers could go on smearing paper with their approximate charts... The real artists, the brotherhood, were going to receive facilities never before available. The result was Algol60. An *algorithmic* language. A language apt to express thought, able to describe *algorithms*. Algol60 did what Cobol was very soon going to undo... It delivered the largest package of *structural* items ever concocted for program writing. Algol had the *block structure* no other language had had before. Block structure means that a program text is composed of a list of grouped statements (*blocks*), which can themselves be composed of yet other (nested) blocks. The blocks are delimited by (special) *parentheses* written as **begin** and **end.** The statements within the blocks are **separated** by a semicolon (**;**). Algol had semantics strictly reduced to what its syntax implied at first sight. Compared to Cobol, this is amazingly fresh, but it is in fact the inheritance of Fortran. A deviation from Fortran is that all variables **must** be declared. As a novelty (or progression), Algol also introduced *scope* in a formal way. This works as follows: variables must be declared (there is a statement to that effect). Such declaratives normally follow the **begin** bracket of a block. The declarative assigns a type to a name, and implicitly sites it (or better: fixes the way in which it will eventually be sited). But, the declarative also delimits the part of the program text in which the variable **name** is known (and thus its **value** is usable): this knowledge is established throughout the **begin-end** block where the declaration is written, and *all contained blocks*. The knowledge is however suspended by another declarative of the *same name* in a deeper block, and the rule now applies to this new declarative. *Figure 1* illustrates the scheme. A remarkable situation: for the first time the *what's-*



```
begin
  integer a
  integer b
  begin
    integer c
    begin
      integer a
      begin
        integer x
      end
    end
  end
end
```

figure 1

in-a-name question became really preoccupying. It also implied some new vistas as far as siting was concerned: variables were sited **only** when the block that contained their declaration became active (by reaching **begin**) and as long as the block stayed active (until reaching the corresponding **end**). The variable became un-sited (and its value **lost**) when the block was deactivated. This was going to have more annoying consequences than helpful ones. Scope was an invention which allowed subroutines (*procedures*) to have *free* variables with a meaningful value. Thus it solved some problems to do with nesting depth. But it created huge deviation possibilities. I'll cover this aspect later.

For *functions*, Algol did still more: it also formalized the returning of a value, thus creating a *functional* sub-language. The way in which Algol introduced *functions* was an extension of the built-in functions of Fortran. Fortran had something like **A=MIN(X,Y)**, clearly a functional notation, meaning A was assigned the lowest value of X and Y. X and Y were arguments of the function, and MIN was its **name**. Any reference to MIN meant execution of the code that realized the necessary computations: no permanent value, no (apparent) site, no way to assign to the name. On the other hand, the *writing* of the MIN(X,Y) largely resembled a subroutine call. This aspect was used by Algol to allow user-defined functions. In fact, the programmer was enabled to define the expression that realized the computation of the function, and to do so in subroutine syntax (a *procedure*) including argument-parameter usage. The subroutine text had to contain an assignment of a value to the function name: a somewhat remarkable statement, but quite normal, given the fact that a function could only have a value when invoked (i.e. when referred to in an expression) and the invocation resulted (*internally*) in the computation and equation of value with function name! Algol did not call itself a functional language, and indeed it is not. However, function calls can and may appear wherever a constant may appear. This means that an expression may be composed of function call(s), whose argument(s) may be other function calls, to any depth. Algol allows functions to have *side-effects*. By this is meant the fact that the function has more effects than just returning a value; such effects could be the modification of a variable in the function's outer scope, the output of a parameter by name, input/output operations, modification of data structures or the usage of variables internal to the function body but whose value is remembered from one invocation to another (so-called **own** variables). Side-effects are desirable in a number of cases (see for instance the *sum* function below which has two parameters by name). But they are always (potentially) dangerous and must be considered with care (see the procedure *incr* below).

Algol also formalized the *parameter-argument linkage* by allowing the receiving subroutine (*procedure*) to specify either **by value** or **by name** linkage, for each of its formal parameters. When a parameter is passed *by name*, any usage of the parameter name within the procedure body causes *re-evaluation* of the corresponding argument, in the current scope environment! When a parameter is passed *by value*, the argument is evaluated at the moment of the call and the value is stored in a temporary memory slot which holds the value of the parameter. Passage by value protects the argument against assignments from within the procedure's body: such parameters can only be used as *input* parameters. Of course, in many cases procedures must output values to their parameters also. It is exactly here that passage by name comes in. But this kind of parameters is more powerful. Imagine that we would like to create a function that computes the result of $\Sigma_{i=a}^{b} V[i]$. But, we have the further requirement that nesting must be allowed so that we can program $\Sigma_{i=a}^{b}\Sigma_{j=c}^{d}V[i,j]$ in a natural way. The following function will do the trick:

```
integer procedure sum(i,l,u,v);value l,u;integer i,l,u,v;
begin integer s;s: = 0;for i: = l step 1 until u do s: = s + v;sum: = s end
```

(parameters that are not declared by value are by name). We can now write the program for our problem as *sum(i,a,b,sum(j,c,d,V[i,j]))*. When sum is invoked, the *i* and *v* are not evaluated. They are evaluated at the first reference during the execution of the function body. In other words, v is evaluated repeatedly within the for loop. Each of these evaluations concerns the inner sum: *sum(j,c,d,V[i,j])* which restitutes the sum of *V[*,j]* for * being the current *i* of the outer sum, and this works since *i* is a parameter by name.

Although passage by name seems interesting, it is rather dangerous: the argument (which can be an expression) may very well deliver **new** values each time, since some of its operand values might have been changed by the procedure as they are external variables of the procedure, and therefore modifiable! Moreover, it is impossible to write a generally valid procedure that increments a variable by 1 (or any other amount). Consider indeed **integer procedure** incr(x); **integer** x; **begin** x: = x + 1 **end**. Parameter *x* must be by name, otherwise the procedure cannot output the new value. Suppose we call *incr(V[func(i)])* and *func* is a function that has side-effects in such a way that it does not necessarily evaluate to the same value for the same argument. In that case, the two references to *x* (which mean *V[func(i)]*) yield a different array element so that our procedure has not the desired result.

choice 1 :action 1	**if** choice1 **then** action1
choice 2 :action 2	**else if** choice2 **then** action2
choice 3 :action 3	**else if** choice3 **then** action3
...	...
choice n :action n	**else if** choice n **then** action n
other cases :action m	**else** action m

figure 2

Algol introduced new (or extended) *connectives*. These were specifically the *right alternation* and various *iteration* connectives. The right alternation corresponds to the ordered *multi-case* alternative (i.e. the ordered *choice*) as indicated below left. Algol writes this as indicated in the right column of *figure 2*. As such, the ordered choice is a not uninteresting extension of the normal **if ... then ... else ....** The iterative connectives are of three types:

(a) conditional with test at top: **while** condition **do** statement;
(b) conditional with test at end: **repeat** block **until** condition;
(c) controlled (with a so-called **current** variable):
for control-variable: = initial-value **step** increment **until** end-value **do** statement.

The **for** connective has caused much concern. Indeed, Algol creators stated that the control variable was valued only **within** the execution of the **for** connective. The variable takes its initial value when the **for** is reached from outside. The statement is executed repeatedly, and at each step of the loop, the control variable is increased by *increment*, until it becomes greater than *end*, in which case the loop terminates. When this occurs, the control variable is **un-valued.** One might say that the control variable is a variable of scope local to the **for** *block*. One might just as well say that it is not a problem-solving variable, but a mere *connective* counter. The result is that the variable cannot be used outside the **for** block. This is perfectly acceptable, were it not that compiler builders mostly leave the variable to its last attained value... thus causing programmers to rely on it after leaving the **for** loop. Apart from this, it looked as if Algol took the decision of turning its connectives into statements (superstatements as it were). This apparently increased the block nature of the language since statements could be *grouped* into an alternation or iteration. But in fact Algol merely created 6 types of statements:

(1) the indicative statement (assignment, procedure call);
(2) the conditional statement (**if**);
(3) the alternative statement (**if ...then... else**);
(4) the repetitive statement (**while ..., repeat ... until**);
(5) the iterative statement (**for ...**);
(6) the declarative statement.

Of course, Algol contains a **goto** statement also, which allows the transfer of control to any other statement by using its label as the target of the **goto**. The additional note that a **begin-end** block (which, if it does not contain any declaratives is called a *compound statement*) can always replace any statement, thus allowing statements controlled by connectives to be quite large, complements this survey and indicates the full writing power of the language and the possibilities, hitherto unheard of, it offered its educated writers. Algol also introduced another notion. It has already been said that all variables had to be declared (quite amazingly, a **for**-control variable needed declaration outside the **for**!). There was more: Algol introduced the **procedure** declarative which *declared* a name to be a *subroutine* (or function text) and gave the subroutine text as its *value*. This had a number of consequences, the principal one being that a procedure name had a scope, just like any other variable (but was it a variable?), thus making it callable from places within the scope only. The procedure text consisted of a **begin-end** block, containing declaratives, thus participating in the scope rules. Notice in particular that a procedure name is part of the containing block, thus is known in the containing block. But since it is (by writing) also known within the **begin-end** block that defines the procedure, the procedure can call itself. *Recursive* calling all of a sudden became possible. Was mankind too young for it? *Recursion* never really made it outside of academic groups... Anyhow, the scope rules proved to be consistent. Some final remarks (for this section): Algol allowed the full boolean conditional expression with the ∧ and ∨ operations, as well as the boolean negation. It also allowed *boolean* expressions to be assigned to boolean variables. This created an ambiguity in the meaning of the =, used by Fortran to express assignment. Algol got out of it by imposing := for assignment (an excellent choice) and = for comparison. Algol also had a **goto** statement. The target of the statement had to be a label. Labels were *words* prefixing a statement. The paragraph notion of Cobol was (fortunately) completely avoided. Nevertheless, it was the existence of the **goto** statement that would culminate 10 years later in the Structured Programming War.

Algol's predicament

Algol, as a new language, had everything which could have avoided the war of Structured Programming. It contained all the required artefacts for clean program writing. Nevertheless, Algol did not make it. Why was that? In fact, there was a lack of *sponsoring*. Algol was the achievement of scientists, and for quite a while, the product remained confined to those circles. No practical compiler was readily available.

In the meantime, numerical analysts got engulfed by Fortran, system programmers stuck more than ever to assembler, as did application programmers (waiting for the exodus to Cobol). And also, because no language of those days had sufficient string handling features, the compilers themselves were written in assembler (which created the rather strange situation that those who wrote the compilers did not use the language and tended to offer involuntary deviations from it...). When an Algol compiler became available (mid 60's), it had to cope with a lot of antagonism. Indeed, the Algol creators had had the nice idea of distinguishing the Algol verbs (in bold type) from the user variable names and constants. But this was in effect an ill-inspired decision. Machines were still card-based in those days and card punches had no bold type. In some compilers, the Algol verbs were bracketised (in /* and */)! This had the most unfortunate effect on the readability of an Algol text... A detail, but a rather important one... Immensely restrictive is the fact that Algol has no provision whatsoever for file operations.

And of course, since Algol was the first language to achieve a degree of sophistication, it suffered from childhood diseases. An important one is that Algol is intended to program numerical processes and remains deficient in the area of symbolic processes (characters for instance). This, of course, was also the case for Fortran. The only success Algol encountered was as a *publication* language. It was ideally suited to convey algorithms through literature. And indeed, for some 10 years, all formal publications were going to use Algol (with extensions and deviations) and this was a tidal wave... With adverse side effects. Cobolists could not read the language and therefore segregated: they ruled out all formal literature and were doomed to stick to the early 60's spirit forever. Systems people, though capable of formal thinking, were more than suspicious. They had resilience to changes coming from non-systems groups such as the Algol creators. Plus, they said, whatever Algol could do, they could do better, since they had Assembler, the most powerful of languages. As a result, also the Systems people became illiterate in Algol.

Algol was going to be successful in one aspect though. It was going to become a *reference*. Most languages to come were going to be block-structured languages and have easily recognizable Algol constructs in their syntax. This would be the case for PL1, Pascal, C, ADA and even (to some degree) the preoccupying BASIC. But the Algol makers would refuse to recognize the filiation... Instead, they started arguing about their child and found that it was not general enough. They wanted even more generality. This culminated in the creation of Algol68, a much too general language which nevertheless contained a number of interesting features to be found back in later languages. There were also those who loathed the new complexity. They made simplified languages: AlgolW and SIMULA (which was Algol60, increased with *classes*, something that came very close to object-oriented programming *avant la lettre* and led, much later, to SMALLTALK)... It was a funny world.

Meanwhile, in Vienna, in the early 60's

Why ever IBM (the SHARE committee, in fact) undertook that piece of work is a mystery. Admittedly, they had the intention of setting up the next version of Fortran (version V and VI), but it turned out differently and, in 1964, they made PL1. PL1 is an Algol-like language. Many will disagree with this statement, but it is a fact that PL1 took over many of the structural ideas of Algol, in particular: the block structure, the nesting of constructs, the scope rules and recursion. In truth, PL1 attempted to merge the features and semantics of both Fortran and Cobol with the structures of Algol. It was the first language that undertook honestly the task of offering resource management as well as program flexibility. PL1 was going to contain file management in a "portable" way, as well as memory management and task management. To ensure the portability, the PL1 makers defined the language on an *abstract machine* . In fact, the entire semantics of PL1 are the exact description of the abstract machine. By doing so, the makers allowed any compiler builder to create PL1, by allowing him to emulate the abstract machine to the physical one. The result is that PL1 needs a so-called **program manager**, which is none other than the run-time package that represents the abstract machine.

Let us start with a syntactic overview, in the light of Algol. Like Algol, PL1 has the *block structure*. A statement can be of the simple type, or be a block of statements (to any depth), surrounded by the **BEGIN;/END;** brackets (notice the semicolon). Algol made a distinction between blocks containing declaratives (a true block) and those not containing declaratives (a compound statement). PL1 calls compound statements *groups* and surrounds them by **DO;/END;** brackets. Such groups are not affected by the scope rules.

The **procedure declarations** are seen as blocks and are contained in **PROCEDURE...;/END;** statements, without the need for **BEGIN/END** but with the same scope rules as in Algol. The linkage between arguments and parameters is ruled by the argument type: for a constant, an expression or a function call, passage is *by value*; otherwise, it is *by reference*. This is a deviation from Algol's by name: in fact it is the same as by name as long as the argument is a variable or a constant in the sense that any reference to it from within the body of the procedure is to the actual variable or constant. Passage by reference is impossible for expressions. PL1 does not allow the programmer to specify the passage mode; however, if passage by value is needed, this can be forced by writing the argument between parentheses, which turns it into an expression. On the other hand, passage by reference is really only interesting for output parameters, and these must be variables anyhow. Of course, due to the absence of by name passage, one cannot create the *sum* function that was explained in Algol. Instead, one creates a single function that effectuates the nested sum (e.g. *sumxy(a,b,c,d,V)*). Just like Algol, PL1 allows the writing of function subroutines. These are procedures that have a **RETURNS(type)** clause on their PROCEDURE header and contain at least one **RETURN(expression)** statement in their body, by which the function actually returns a value. Any procedure can be exited by executing a RETURN statement or by *falling through* the end.

The PL1 *connectives* are similar to those of Algol, but there are differences. The major difference is that PL1 considers most of its connectives as statements in their own right. And since PL1 uses the semi-colon as a statement terminator and not as a

In PL1:	**In Algol:**
IF condition THEN action1;ELSE action2;	**if** condition **then** action1 **else** action2
DO WHILE(condition); action; END;	**while** condition **do** action
DO I = begin BY increment TO end;action;END;	**for** i: = begin **step** increment **until** end **do** action

figure 3

separator (like Algol), the syntax is different. Merely compare the PL1 and Algol syntax in *figure 3.* The semantics are identical, apart from the fact that the variable of a controlled loop keeps its last value after the loop has ended. Algol needs a **begin/end** bracket if the action contains more than one statement. PL1 only needs brackets on the THEN and ELSE, and uses **DO;/END;** instead (notice the semicolon!). PL1 also allows the two forms of the iterative DO construct to be combined and alternatively allows an **UNTIL(condition)** or **UNLESS(condition)** instead of the WHILE. This is only partly similar to ALGOL's **repeat-until** construct. Nesting rules of the iterative constructs are as for Algol. Later versions of PL1 have a **SELECT/WHEN** construct which is similar to Pascal's **case** structure (to be discussed later). In the THEN and ELSE branches of the IF, PL1 allows unlimited nesting of IFs (unlike Algol).

Major differences between PL1 and Algol appear in the area of variable definition. Variables need not be declared, in which case contextual or implicit declarations take place (somewhat like in Fortran) with rather bizarre scope rules. Something I consider an immense minus point. Variables are defined by means of the DECLARE statement and have Cobol-like semantics. PL1 variables have a number of attributes, which are associated with the variable name, such as: the **type** and **base**, the **length** and **precision**, the **scope** (an explicit deviation possibility from the regular scope rules), the **storage class** (the way in which a variable is sited, see further), the **initial value** or **initial expression.** PL1 also defines a number of new variable types, given here for completeness: the *label* type, the *entry* type (a subroutine), the *bit string* type (very different from Algol's boolean type), the *pointer* and *offset type, the area* type and the *file* type. These types are interesting, especially when building systems applications, but they cause a lot of semantics to appear in the language. They are also dangerous (especially the pointers which may point anything without any protection) Finally, PL1 allows data declaratives to be nested in Cobol-like leveled structures. These structures can contain arrays, and there may also be arrays of structures. A LIKE clause allows the programmer to declare variables so that they have the same *structure* as some pre-declared structured variable (this is a small step, which Pascal was going to take much further).

Other major differences between PL1 and Algol reside in the procedural package: PL1 covers all needs by offering an immense set of **built-in** functions. *String handling*, in particular, is covered in this way by (amongst others) the very important SUBSTR function. This function is rather peculiar: it is a *naming* function (a notion which will be dealt with later).

Another aspect of PL1 (much debated and rather controversial) is the freedom in type intermixing. Most data types are compatible with one another, in that PL1 uses an almost unbelievable apparatus of conversion semantics. To say the least, this may cause rather awkward situations, as a result of programmers abandoning discipline. It is also possible to exercise explicit conversion via the string editing features, a rather nice asset.

A last point: PL1 offers a GOTO statement, similar to Algol's, which uses a label as a target. A dramatic oddity of the GOTO is that one can exit from a procedure merely by GOing TO a label that is external to the procedure. This is a negative consequence of the scope rules: labels also have a scope, and therefore, a procedure knows the labels of the surrounding text.

PL1's misfortune

The makers of PL1 had a number of objectives: anything that makes sense and is unambiguous should be legal; there should be no need to escape into machine (assembler) language; machine characteristics should not be visible (abstract machine); it should be possible to use the language in a novice way as well as a specialized way (reduced training); the accent had to be upon the ease of programming. These were completely new and commendable objectives: PL1 definitely aimed at providing comfort and coherence to the programmer. However, PL1 shared Algol's fate: it did not succeed in hitting the market place. The reasons were not the same, though. PL1 was immensely superior to Cobol, and should have made it. However, it came too late: Cobol was already a firmly established language, and was supported by the US Ministry of Defense. As a consequence, PL1 enjoyed no initial thrust whatsoever. Apart from these aspects, PL1 was a core devourer: the abstract machine could cost up to 150K of memory, and the machines of those days could not always offer so much physical core. Moreover, computer scientists largely condemned PL1, and even though the application world was not really concerned with formal literature, some echos came down and were damaging. The opposition of formalists, though understandable, was regrettable: a language that obeyed a number of sound principles was condemned without mercy, only because its vast semantics contained some potential contradictions. The bells were tolling for block-structured languages: none of them would ever be a real success in the application world. The systems people would not use PL1, because they found that the abstract machine stood in their way. Still, and this is remarkable, a lot of systems development languages to be concocted later, were going to have a PL1 look about them!

The objections to PL1 include the following: oversized object code; a difficult to learn language (but how easy is Cobol?); unreadable source code because of the nesting of blocks (but how readable are Cobol's cascades of PERFORMs?); unreadable *expressions* (a strange argument: since when is ADD A TO B GIVING C more readable than C = A + B ?); overwhelming semantics which are very context-dependent and rather error-prone; the semi-colon as a delimiter rather than a separator (a rather formal argument, which in practice has no importance at all); the "overwhelming" GOTO; the existence of POINTER variables allowing the implicit manipulation of variables causing deviations from the semantics with high error-proneness; grotesque implicit conversion from one type into another; all brackets (DO, END, BEGIN,...) are *statements* rather than *delimiters* in that they must be ended with a semi-colon (a little bit superfluous, maybe, but nothing to get anxious about!).

A new question: what's in a label?

PL1 and Algol introduced a new object: the **label**. And what is a label? An obvious definition is: a label is a *name* given to a statement or a procedure. Does this name have a value? Or a site? One could see the value of a label as the address of the statement that carries the label, since a label is the target of a GOTO or a CALL. Obviously, the label is *constant*: its value cannot be changed. A strange object, which exists only because it appears in front of a statement. Is a label *variable* conceivable?

It is. As a named object, whose value is a label constant. A label variable is therefore sited quite normally and may be assigned to. The label variable can be used instead of the label constant as the target of a GOTO.

Let us come back to Fortran. In that language, labels are numbers. But can they act as a number? In particular, can a label be computed? Is something like this possible: **GOTO 2*X+32**? In Fortran, it is not. But in some other languages, such as BASIC, it is! Is it still allowed to consider a label as a mere name? More generally: can a name be *computed*? This is a very general and important question with many implications, which I will deal with in the section on macroprocessors.

Another new object that PL1 introduces is the *procedure*. Of course it exists also in Algol and as a *subroutine* in Fortran. However, PL1 considers the name in front of a procedure to be an object in its own right, somewhat akin to a label. PL1 sees such a name as the name of an *address* (the address of the procedure, in fact, or better: its *entry point address*). This is again a constant, and its value cannot be changed. But PL1 also gives the possibility to declare **entry** variables, which are regularly sited and can receive as value the name of a procedure. Such a variable can be used as target in a CALL statement. As a consequence, there is no way in PL1 to distinguish from the mere typography whether a name is a label/entry constant or variable. Semantics have to come into the game: if such a name appears in front of a statement, and is followed by a colon, it is a constant. Otherwise it may be a variable or a constant, and only a *dictionary* of the names used in the program with their type (the so-called *cross-reference*) will indicate which is which. This is a cause of confusion. Since constants are the common situation in this area, it would have been well-advised to distinguish label/entry variables by means of a typography, such as for instance a special first character to the name of the variable.

PL1 has got something else that is totally new: *pointers*. Pointers are *variables*, thus they are sited and valued. Their value is an *address*. Any address? PL1 was careful: the semantics only allow the address of another known variable to be the value of a pointer variable. The only exception is the special pointer constant value NULL, which can be used to indicate that a pointer variable has in fact no value. A built-in function that comes in handy is the ADDR function, which returns the address of its argument (itself a variable, not a constant nor an expression). The question is: why do we need pointer variables? Surely, in a high-level language one can expect the compiler to take care about all siting operations? Pointers are needed when one has to manipulate non-static data structures, such as list structures. Such structures are constructed out of a normal structured variable, which exists in more than one occurrence. The different occurrences are *linked together* by pointers: each occurrence contains the *address* of the next occurrence, and, of course, the last occurrence has a NULL value in that field. Using such structures can only be done with pointers. There is an inherent danger: a pointer field in a list structure may very well contain an impossible value, or even a wild value (a *dangling* pointer). There is no way to validate such pointer values: everything is up to the programmer who must exercise the required discipline. List structures abound in systems programming: all the *control blocks* used within a system are linked by addresses. Thus, if PL1 is used for systems-like applications, this cannot be conceived of without pointers. It looks like pointers are an unavoidable evil. That this is not true was being proved in the same years by the LISP language. But no one really cared.

Storage classes: an evolution?

In PL1 it is possible to declare a variable so that it is sited at an address which is given by the programmer as a pointer. Such a variable is said to have the **storage class** *based*. For instance: **DECLARE XYZ FIXED BASED(PT1);** where PT1 is a pointer variable. The variable XYZ is taken to reside at the address in PT1. This is true at each moment, so that the site of XYZ actually "follows" the changes of the value of PT1. The value of the variable is the value that exists at that moment at the given address. Worse still, even a pointer variable can be based upon another pointer, and this can be done to any depth. The whole pointer mechanism can give rise to a true jungle.

PL1 has four more storage class possibilities. Their intention is to define the way in which a variable gets sited.

(a) The **automatic** storage class: the siting of a variable in this class takes place implicitly at the moment that the block (BEGIN or PROCEDURE) that contains the variable is activated. The variable is un-sited when the block is terminated. The siting remains in effect, however, when the concerned block activates another block. Variables in this class have no permanent site, and can therefore not be used to preserve values across different activations of the containing block.

(b) The **static** storage class: the siting of such variables is effectuated once and for all by the compiler itself and is permanent. Values of such variables are preserved. Notice that constants are static by definition.

(c) The **controlled** storage class: the siting of a variable is effectuated by means of a special function, ALLOCATE, which must be invoked by the programmer prior to using the variable. The ALLOCATE function returns the address of the variable consecutive to siting. This way, a program may allocate the same variable more than once: different sites will be allocated by the system; this is the way to create list structures. Un-siting is done by the converse function FREE. If many sitings of the same variable have been effected, FREE always undoes the most recent one (a so-called *Last-in First-out* allocation policy).

(d) The **defined** storage class: this is not really a storage class; it is rather the indication that the declared variable is a kind of alias of another variable, and therefore it has the storage class of that other variable. It is absolutely identical to Cobol's REDEFINES clause. This possibility is therefore just as bad as it is in Cobol, if not more so because scope rules may complicate matters (indeed, the defined variable may have another scope -a more local one- than the variable it is defined upon).

A notion that is strongly connected with siting is initial valuing. PL1 makes the not unreasonable claim that once a variable is sited, it is valued, because in a physical machine there always exists a value at any address. This is the *initial* value of the variable. Of course, an un-sited variable has no value and cannot be used at all. PL1 also allows the programmer to define an initial value for a variable. This is indicated by means of the INITIAL clause in the DECLARE statement. The initial value is assigned automatically to the variable at the moment it gets sited. In other words, the initial value is assigned to an automatic variable each time the containing block is activated. The initial value is pre-assigned for static variables. Other storage classes do not allow such initial valuing. This is rather normal for *based* and *defined* variables, but one may seriously wonder why *controlled* variables cannot be initialized (at the moment the ALLOCATE function creates them). On the other hand, most formalists would like variables to have a status such as *valued/un-valued* and *sited/un-sited* kept as a **tag** with the variable definition. Reference to an un-valued variable could thus be recognized and prohibited, thereby increasing the reliability of programs. No classic programming language however has this feature.

Scope, the great pretender

I have indicated earlier that scope rules are an almost inevitable consequence of block structures. But I also hinted at problems with this new notion. Scope rules in languages such as Algol, PL1, Pascal, C, ADA are *lexical*. In fact the scope of a variable is the portion of text in which the variable is known (i.e. usable). The portion of text is the block in which the variable declaration is actually written and all blocks contained in this block (unless a superseding declaration of the same variable name occurs). The *place* of the definition is the DECLARE statement for a variable, or the labelled statement itself for a label or entry name. By this rule, the label of a procedure is taken to be part not of the procedure but of the surrounding text, and this is normal, since otherwise the surrounding block could never call the procedure! It all sounds very disciplined. But is this true? A deeply nested block (such as a procedure) has access to all variables of the higher level (containing) blocks. Thus, in the low-level block one can assign values to variables of a higher level block. In other words, subroutines, apart from predictable side effects (such as Input/Output) and modification of arguments, can also have much less visible effects. This is called modification of *free* variables (variables that a block can use, but which are defined in a higher level block; this is in contradistinction with *bound* variables, i.e. variables defined in the block itself, either as regular variables or as parameters of the procedure). Allowing the modification of free variables is clearly a negative discipline, one that will cause programs to be more error-prone. Scope rules would have been more acceptable if there had been a way to restrict the usage of free variables (for instance: variables usable only in read mode, but not in modify mode). PL1 allows the programmer to restrict the scope of a variable to only the text in which it is declared and not the contained texts. By default however, scope is as large as it can be. Another problem appears with the GOTO statement: it can actually branch to any label in the current scope, i.e. to a label outside the block. This causes termination of the current block and un-siting of all of the block's automatic variables. Of course, if the GOTO is in a very deep block and the target label in a very high block, then this un-siting and termination is effectuated for all intermediate blocks as well: quite a cascade! And, a subroutine that exits in such a way does not come back to its calling point, but to some label. How acceptable is this? In my opinion, it is not acceptable at all! One can also create very serious inconsistencies. Consider the following example (*figure 4*): in statement (1), a variable E1 is declared as an entry (i.e. a subroutine address). E1 has an outer scope. Statement (2) defines a procedure, A, while statement (4) defines another proce-

```
DECLARE E1 ENTRY;          (1)
   A:PROCEDURE(X);         (2)
   DECLARE X...;
   ...
   E1=P2;                  (3)
   RETURN;
         P2:PROCEDURE(...);     (4)
         ...
         END P2;
   END A;
...
CALL A(...);               (5)
CALL E1;                   (6)
```

figure 4

dure, P2, nested in A. Statement (5), of the outer scope, calls A. Procedure A assigns E1 in statement (3), and it can do so since both E1 and P2 are known in A. However, when, at the end of A, control comes back to the mainline, the call to E1 (i.e. to P2) in statement (6) will fail, because P2 is not a label that can be called from the mainline. If an activation of P2 were possible from here, this would mean that P2 could go and access variables of A (since they are known lexically within P2), but these variables were never sited! Scope rules are in fact more cumbersome than useful. If we take a look at functions (also subroutines), the situation may become critical. Functions are beings that replace expressions and therefore stand for a value, re-computed at each invocation. The *environment* of functions is composed primarily of their parameters. An extension to this environment is given by the outer scope. But it is in conflict with the formal definition of the concept of a function that the execution of a function could change anything apart from creating a return value. Thus, the *pure* function should not output any argument value nor modify any free variable. Indeed, say some, a function should not even perform any visible action (such as input/output) apart from computing a return value. But this is the purist's view. PL1 does not at all protect functions. To PL1 a function is a procedure like any other one. Thus, the programmers must exercise discipline, and we all know how reliable that is. An unfortunate aspect is that functions in PL1 cannot return more than one value: they cannot return structures nor arrays (and in some implementations not even a character string); when programmers are in need of such returned values they use free variables instead. An unfortunate situation. PL1 is not by any measure a functional language. And yet, it would have been easy to protect functions: pass all arguments by value, access outer scope in read mode only (or not at all), allow the RETURN of agglomerate values, do not allow the RETURN of a pointer or label (since these could refer to items in the function body itself which creates at least a conceptual conflict).

This being said, the idea of allowing scope to penetrate blocks, but restricting the operations in a lower block to only reading the outer block's variables, introduces a very rich notion. Indeed, an object may be a variable in a given context and a constant in another (deeper) context. Thus, the modifiability of an object may range from constant, over *more or less constant* to *not at all constant* (i.e. truly variable). No classic language offers anything like this, however.

Dijkstra has made an interesting proposal, which is based upon the strict discipline that all variables used in a block *must* be declared. In fact, he stipulates that also outer variables should be declared in an inner scope that desires to use them. In the absence of such a declaration, the variable, though known, may not be used. Thus, each block must commence with variable definitions, each such variable having a scope attribute (which I take one step further than Dijkstra did) of *either* **global** for a variable *inherited* from an outer scope, *or* **local** for a variable declared for the first time here, *or* **private** for a variable declared for the first time here and which cannot be inherited at a deeper scope level. When a name is re-declared by means of **global**, it can also have the supplementary attribute **variable** or **constant**. It is also a good idea to require that a variable be initialized in all cases, rather than make this optional. Only **local** and **private** variables can be initialized, of course. Initialization could be achieved by means of a clause in the DECLARE statement that contains a value or a call to an initialization routine (a procedure), which could expect as a parameter the variable itself, always passed by reference (otherwise it is meaningless).

Such an initialization routine, if written in the scope of the variable to be initialized, can obviously access the variable. However, there is not yet a value. Dijkstra proposes to declare the variable in the routine, using the *pseudo*-scope **virtual** rather than **global**. The initialization routine could in fact also be a BEGIN/END block written in the DECLARE statement. The whole thing is an attractive proposal, which no single language has implemented.

Not really new, but renovated: the expression

Fortran (and autocoders) introduced expressions. Cobol, on the other hand, did all it could to banish them. So, what is an expression, and why are people afraid of it? Obviously, an expression is a recipe for the calculation of a value. **a + b** is an expression indicating that the value is obtained by adding the values of **a** and **b**. The simplest expressions are single variables or constants. The most general form of an expression is a *list* of variables and/or constants separated (connected, one could say) by *operators*. Fortran offered the operators **+ - * / ****. Semantically it is not that simple, however. Indeed, the world is composed of two classes of people: there are those who think that 7 + 4*5 is 55, and those who claim it is 27. One can find pocket calculators that deliver 55 and others that deliver 27. *All* desk calculators of the 60's gave 55. So what is up? There were two ways of interpreting an expression: the mere left to right way (result 55) on one hand, and taking into account operator precedence (result 27) on the other hand. Indeed, for people who have taken algebra, it is a known fact that multiplication has a higher priority than addition so that 4*5 is done before adding 7 to it. And that is the problem: the syntax does not indicate which order of evaluation is meant. A further complication arises when priorities are considered: how indeed could the result 55 be achieved if one wanted it? This implies a deviation from the operator priorities, which can only be indicated by using parentheses: **(7 + 4)*5**. As a result, expressions may become rather complicated. The philosophers of the left-to-right way will therefore prefer the way of Cobol:

MULTIPLY 4 BY 5 GIVING X. ADD X TO 7 GIVING RESULT.

In fact, this is a poor way of doing things: the programmer must anyhow decide about effectuation order, so why can't he write 7 + 4*5 and remember the precedence rule? True: Cobol has a COMPUTE statement which swallows expressions, but no Cobolist uses it. In the remainder of this section, I will use the precedence philosophy. But even so, isn't there another possibility of writing expressions, preserving priority and avoiding ambiguity? Yes, there is. Let us make a detour first. Suppose we have a function called **add** which takes two arguments and adds them together, and another function **mult** which multiplies two arguments. We can rewrite our expression as **add(7,mult(4,5))**. The writing order of the expression elements (the 7,4,5) is preserved, but the delimiters are gone. There are no operators anymore. Instead we have functional notation which replaces priority by *nesting*. Now let us make some typographical changes. First, drop the parentheses. Next, use the **+** sign instead of **add** and the * sign instead of **mult**. Our expression becomes: **+7:*4:5** (the colon serves as a separator), and this is strictly equivalent with the functional notation and therefore unambiguous. It suffices to remember that both **+** and * mean an operation to be performed upon the *two* operands that follow it left to right, and if such an operand is itself an operator, first execute this one and so forth to any depth. The notation

here presented is called *Polish notation* or *prefix notation*. It never needs any parentheses.
As an example:

(a + b*(3 + 7))*((x + y)*4 + k) is equivalent to * + a*b + 3:7 + * + x:y:4:k

Nothing of course prevents us from writing the operator after its operands. In this case, we obtain the functional notation (7,(4,5)mult)add or with the same typographic shorthand as before: 7:4:5* +, and the more complicated expression would be written

a:b:3:7 + * + x:y + 4*k + *

This notation is called *reverse Polish notation* or *postfix notation*. Our more human way of writing expressions is called *infix notation*, and this can never be parentheses-free. Polish notation is very much in favour with compiler builders. There are pocket calculators even that only accept this notation, just imagine! Notwithstanding the fact that Polish notation is unambiguous and therefore to be preferred, it is never offered. No language (apart from LISP and its clones) contains it. So infix notation it is. This seems natural enough for so-called *binary* operators that have two operands (in which case the operator separates the operands). There are also operators that have only one operand (*unary* operators); these are written before their operand. Operators with three operands (*ternary* operators) have special syntax rules and other *n-ary* operators are written in functional notation. The rather special operators that have no operands (*0-ary* operators) are written as such. In Fortran the operator priority was established in conformity with algebra. Priority 1 (highest): **unary** -, +; priority 2: **; priority 3: *,/; priority 4: +,-. Operators of equal priority evaluate from left to right. Care must be exercised regarding division. Find out for yourself what is the difference between the following expressions:

a*b/(c*d), a*b/c/d, a*b/(c/d), a*b*d/c

PL1 and Algol extended upon the expression, most notably by introducing the *conditional expression* which can be used as the operand of the **if** or **while** connectives. PL1 allowed very complicated expressions in this respect for instance:

(A + B*(7 + 5) > 45*X & TUV < = 15) | XYZ

which contains a number of new operators. The = >= <= > < are comparison (or *relational*) operations which actually compare two operands and yield the boolean values *true* or *false*. The & | ¬ are boolean operators which perform the logical **and** or logical **or** of two operands or the logical **not** of one operand (incidentally, Algol uses ∧ for **and**, ∨ for **or** and merely **not** for inversion). In order to understand the conditional expression given above, we also need to know what the priority of relational and logical operators is with respect to arithmetic operators. Since comparisons are *propositions* in boolean calculus, the relational operators must have a higher priority than the logical operators (the latter connect propositions). In a comparison the two operands are arithmetic (or similar) expressions. Therefore, the relational operators must have a lower priority than the arithmetic operators. This settles the question, with the following additional remark: all relational operators have the same priority, the logical **and** and **or** also have the same priority, but the logical **not** has the highest priority of all operators. Unary operators still have some ambiguity when more than one of them occurs in sequence. Some languages consider them *outside in* (left to right) while others consider them *inside out* (right to left). As can be seen, for a compiler maker the choice of operator priority is slightly subjective and, indeed, various languages have various rules. An expression may also contain function invocations. These are in fact already in prefix form. Therefore, they also have the highest priority. And remember that any argument of a function can also be an expression. Algol has a supplementary operator, a *ternary* one: the **if** expression, which is not the same as the **if** statement. It is written as follows:

if (condition)**then**(expression)**else**(expression)

It can be used to write the following statement:

xyz:= if(a > b)then(4*x)else(75) + 43

and this has an obvious meaning. There is some confusion in Algol as to the priority of the **if** expression: how far indeed does the **else** reach? This is not well described. My proposal is to use the mandatory parentheses as indicated. Then the whole **if** construct can be considered similar to a function and given the highest priority.

As can be seen, expressions are truly powerful beings and by inversion of this argument we have yet another sadly weak point of Cobol.

What's in a name: the question recurs

The built-in functions of PL1 contain a number of special beings. One of them is the SUBSTR function: **SUBSTR(string,position,length)**. This function returns that part of the *string* (argument 1) which starts at *position* and extends over *length* characters. So far, so good: obviously a useful function. But the PL1 makers invented something: they allowed the SUBSTR function to be used as the receiver of an assignment (provided the *string* is a *variable* and not an *expression* or a *constant*). A rather funny situation: SUBSTR is a function, but instead of delivering a value, it now receives one. And how is this achieved? In fact, SUBSTR *designates* a portion of the receiving string. And this is also a useful feature. PL1 calls such beings *pseudo-variables* since they can indeed replace a variable anywhere, even as receiver in an assignment. I prefer to call them *designator* functions since they actually designate a site into which an assignment is to be done (or from which an "extraction" must occur). We could also call them *naming* functions. In effect, they are unnamed objects that have a site and have or can receive a value. Other such designator functions exist. One of them is UNSPEC, which allows the programmer to see the argument as a pure bit string (and thus permits the modification of any bit in the argument). Another one is STRING, which takes two character variables as arguments, and sees the concatenation of them as a new modifiable object. A last, very important one, is the array reference, which I discussed in the context of Fortran. Indeed, if A is an array, then what is the meaning of **A(i)**? Merely the designation of the i-th element of A, and this can be used as a value or as a receiver of a value. Some people actually use the notation **isub(A,i)** to *designate* an element of an array, and **isub** is clearly a designator function. The important thing to see is that a designator function actually computes a *site* by **refering** to at least one really existing variable, using a number of refining arguments.

An interesting question is: can one construct such beings? PL1 says a hard "no" here. And all other languages as well. Nevertheless, such functions are conceivable enough. I can't resist the pleasure of elaborating somewhat on this idea. So, let us assume we want to create a designator function that designates the last character of a string. We shall write it LAST(string). As a regular function that returns a value, it is easy enough to define it (*figure 5*). Can LAST be used as the receiver of an assignment? Well, since the RETURN is itself a designator function, nothing would really prevent us from using the function in this *reversed* way. This is not always possible however. Suppose we now want a function that designates the new string formed of all even-ordered characters of an object string. We will write it as EVEN(string). Again, as a regular function it is not hard to define (*figure 6*). This time, however, the RETURN is not composed of a designator function, so we cannot use EVEN in a reversed way. A solution is possible, though. We are free to consider that a designator function is an *operator* that establishes the siting mechanics of the designation, rather than merely returning a site. Thus, a designator function used as a receiver (and I will call that an *acceptor*) must be able to *accept* a value and move it into some site, which may designate contiguous or even *linked* cells. *Figure 7* is a plausible syntax for defining such an acceptor. This function will actually accept a value (as indicated by the ACCEPTS clause that has taken the place of the RETURNS clause) and move it into some site(s) defined by its parameter. Within the function body, the ACCEPT statement serves to move into a local variable the value that is assigned to the function at its invocation point. Notice the rather obvious symmetry between the *returner* version and the *acceptor* version. I have not investigated whether such a systematic approach is always possible, but it is an intriguing thought to pursue.

```
LAST:PROCEDURE(X)RETURNS(CHAR(*));
DECLARE X CHAR(*);
RETURN(SUBSTR(X,LENGTH(X),1));
END;
```

figure 5

```
EVEN:PROCEDURE(X)RETURNS(CHAR(*));
DECLARE X CHAR(*), I FIXED;
DECLARE R CHAR VARYING;
DO I=1 BY 1 WHILE(I*2<LENGTH(x));
SUBSTR(R,I)=SUBSTR(X,I*2,1);
END;
RETURN(R);
END;
```

figure 6

A last observation is that some algolists tend to use the **if** expression as an acceptor as well (at least when they use Algol as a publication language). Indeed, they write:

(**if** condition **then** name1 **else** name2):=expression

Depending on the *condition*, the value of the *expression* is assigned to either *name1* or *name2*. The parentheses are required in order to avoid ambiguity.

```
EVEN:PROCEDURE(X)ACCEPTS(CHAR(*));
DECLARE X CHAR(*), I FIXED;
DECLARE R CHAR(*);
ACCEPT(R);    /* value assigned to */
              /* function at its  */
              /* invocation point  */
              /* is moved into R   */
DO I=1 BY 1 WHILE(I*2<LENGTH(X));
SUBSTR(X,I*2,1)=SUBSTR(R,I);
END;
END;
```

figure 7

Stacking: a new-looking oldie

Let us come back now to the mechanics that are used to effectuate a subroutine call. The problem is the preservation of the environment of the caller, since when control comes back, this environment must be put back in effect. A first annoying thing is that the machine keeps some data of the caller in *registers*, which may very well be used to other purposes in the subroutines. The easy way out, is to *save* the registers into some compiler-created save area, and link all these areas together (in order to cope with subroutines that call subroutines that call subroutines,...). The last save area of the list holds the registers that are in effect upon entry into the currently active subroutine. When the subroutine returns control, the previous save area is used to refresh the registers, and all is well. Such a list is therefore used in a last-in/first-out way, and it is usually called *return stack* because it holds the return address of a subroutine caller. These dynamics sufficed for Assembler, Cobol and Fortran. But what about Algol and PL1? There is a difference here. Indeed, consider the automatic variables (in Algol, all variables are automatic). They are sited at the moment a subroutine (or block) becomes active and un-sited when the block terminates. Siting (allocation) is dynamic, and must therefore be done *somewhere*. What better place than in the stack, as an extension of the save area? The dynamics of the stack imply siting and un-siting at the right moment. So, the *machine register* that points the current stack area is also the register that will allow addressing of the automatic variables. The stack has become a real *data stack*. PL1 and Algol are called *stack languages*. The stack mechanism creates some overhead. So, people tended to circumvent it by making variables static (these are allocated definitely, without stack). The question was asked: since Fortran and Cobol live quite happily without a data stack, why should we bother in PL1? Well, what if a routine calls itself? This is allowed in Algol and PL1 (and indeed, it is a powerful mechanism of extreme value, as I will attempt to prove in chapter 7). If we use this feature, this means that a subroutine can be active many times simultaneously. Each of these simultaneous activations must have its own *incarnation* of the routine's local variables, otherwise the whole thing will become unusable. The only way to do this is by having a data stack, which *nests* the multiple incarnations of the automatic variables. Remarkably enough, most computer centres of the business platform rejected, prohibited, the usage of recursion. Why? For no other reason than fright. A grotesque twist of minds, even of the best inclined ones. Still, it must be admitted that the existence of a stack is not without problems. Indeed, consider a program that contains two procedures, A and B, which are not nested in one another. Subroutine A can call subroutine B, since B is in the scope of the whole program and is therefore known in A. When the program calls A which in turn calls B, the stack comprises the *environment* (the automatic data) of the program, followed by the environment of A, followed by the environment of B. This creates the impression that B can access data from the environment of A. However, this is false: since B is not nested in A, it has no knowledge of the variables defined in A. In other words, the scope rules are not reflected in the topology of the stack. Our scope rules are *lexical* (they pertain to writing order). The stack on the other hand reflects a dynamic scope, a *call scope*. The two are not necessarily the same. Some languages, such as LISP, use call scope only. Others like Algol, PL1, Pascal, C, ADA, use lexical scope only. I think this is another indication of the potential dangers that lie in the notion of scope. There is more. What about the return value of a function? Where is it held? It cannot be held in the environment of the function itself, since that is deactivated (abandoned) when the function terminates, and the return value would be lost. So the only place is the environment of the function's caller. In fact, the return value is associated to the name of the function, and that is part of the caller's environment (or some higher scope). A consequence is that one must be very careful in creating functions that return addresses (pointers) or labels or even entry values: these should never point at something that is part of the function's internal data.

Exceptional Exceptions

PL1 was the first language to have **exceptions.** In other words, PL1 defines a number of errors or exceptional situations which it is able to detect. In PL1 terminology these are called *conditions*. When such a condition occurs, the program manager is then able to take some action upon it. The nice thing is that the action can be user-defined. The programmer writes a so-called ON-UNIT, which is really a subroutine, defined as a BEGIN-END block with normal scope rules. The whole problem is to decide whether PL1 should take standard error action or should execute such an ON- UNIT. To this effect PL1 defines 3 notions:

(a) the condition is **defined** if there exists an ON-UNIT for it (the ON-UNIT names the error situation, using a standard set of PL1 names or some name of its own, which then is a user-defined condition);

(b) the condition is **activated** as soon as the execution flow passes **normally** over the ON-UNIT. It is **deactivated** by means of a REVERT statement. Different ON-UNITS for a same condition may be activated. They are stacked, and REVERT deletes the most recent one;

(c) the condition is **enabled** by the appearance of a prefix **(condition- name):** somewhere in the text. It is disabled by the appearance of a prefix **(NO condition-name):**. The whole portion of code comprised between the two prefixes is **enabled** for detection of the error.

Now, when one of the standard errors occurs in a portion of code **enabled** for that error, then, if the corresponding ON-UNIT is in an activated state, the ON-UNIT will be executed. Otherwise the standard PL1 action is taken (which varies by type of error). If such an error occurs in a disabled portion of code, no action is taken. However, for some errors, there is enabling by **default** anyhow, so that such errors are always treated. A user-defined condition does not correspond to a PL1-known case. Thus, the user has to raise the condition by a SIGNAL statement. This is interesting when PL1 programs are used to interface with non-standard resource handlers that return error-codes of their own and for which some user-conventional action has to be taken. The whole thing is rather nice and allows very interesting programming avenues (of the event-driven type). However, there are severe shortcomings. Indeed, where is control transferred to after execution of an ON-UNIT? Is the failing statement retried? Does execution proceed with the next statement? Is the program terminated? Is the containing procedure terminated? Or what? Can the programmer decide? PL1 left this in the open. The programmer can do a GOTO to some place he knows about. And I must admit, it is not very clear what a *continue* or *retry* may mean at the object code level ...

Chapter 4
THE YEARS OF MULTI AND PASCAL

When System becomes overwhelming, things get locked. Macroprocessors reach a top. Pascal comes in strong.

The roaring 60's : SYSTEM went multi

While all these linguistic developments took place, something insidious, but with enormous possibilities, was going on. SYSTEM came of age. The whole thing came from performance requirements. Conventional runs (reading the program card deck, execution, printing the listing) were lengthy, and mostly so because of the printing. The printer was (and still is) a slow device, and while it was doing its job, the CPU nearly idled, losing time. So, a shrewd systems designer came up with the idea: why don't we *spoolout* the print output to a more rapid device (a tape, and later a disk), thus gaining time? Printing could thus occur later (from the tape) by means of a system-embedded routine. The question was: what is later? At the end of the day? A possibility. But there was a better way. Why not perform the printing anyhow, and let the CPU do normal work in the idle time of each printing line? All it needed was a device by which CPU power could be switched between sending a print line and executing some user programs (or some system service). That was the beginning. Not much to speak about, is it? But it was the first multi-acting system ever. And it contained another novelty: a **wait queue**. Indeed, the outputs that were waiting for printing were *queued* onto the help device. The waiting requests were serviced in order of decreasing age: a *first-in/first-out (FIFO)* policy. The system went *multi* and found all the queueing problems that it still has today. Quite soon, the *multi* approach extended from mere spooling to generalized *multitasking* (I'll stick to that word for quite a while). Multitasking was the capability of the system to execute more than one *task* at the same time (macroscopically speaking). And a task was *some* amount of sequentially contiguous work. The way it was set up was: use the I/O idle times of any task to do some other task in (or at least a portion thereof). The mechanics involved were apparently obvious. Assume two tasks, Task1 and Task2. Assume furthermore that task1 is executing, and task2 is not, as in *figure 1.* At a certain moment task1 launches some I/O, by virtue of which it stops executing and waits for completion of the I/O. Task2 now gains control. This task may also do I/O, and idle. Since there are no other tasks, now the machine idles. When task1's I/O is finished, this task regains control. Suppose that when task2's I/O is finished, task1 is still in control. Task2 is

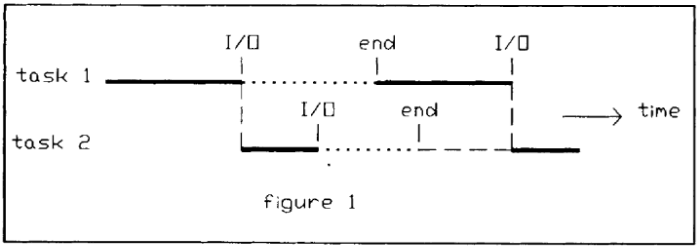


figure 1

runnable but cannot run before task1 abandons control once more, and so forth. Thus, a task can be in any of 3 states: **active, waiting** or **ready** (i.e. able to become active). The diagram in *figure 2* depicts the possible transitions. The situation complicates by the introduction of *preemption* and *priorities*. Tasks could have priorities, so that a high-priority job would never have to wait in a ready state but rather takes control and pushes the currently active task into a ready state. On top of it all the I/O management system became a bottleneck. How was that possible? Merely because I/O devices were slow (compared to the CPU) and thus unable to serve concurrently all requests coming from simultaneously executing programs. Some serving task was therefore required, call it task 3. The serving task was going to *schedule* the demands for I/O against the device (assume only 1 device, for simplicity), which introduced a queue of *I/O requests* . Task 1 and task 2, when requiring I/O, would hang their request on the I/O request queue. Task 3, a never ending (cyclic) task inspected the queue and effectuated the request by sending it to the device. Task 3, unable to do its own job in a zero time interval, needed to be inserted in the give/take machine time mechanics, and probably needed a higher priority than any user program task, since it helped in the effectuation of these user tasks. I/O had become a *resource*. And that was only the beginning. But an important notion was born: a *resource manager* that was to schedule the resource against the requests of a request queue, in some order. The queue was an inevitable plague of the situation: the queue accounts for speed differences; only in a system with *all components* acting at identical speed, can queues be avoided (although even then, there may be queueing because of asymmetrical peaks). For a queueing system to hold the road, it must be guaranteed that queues never grow uncontrollably, choking the system: if the system comes to a standstill, this is a catastrophe! The CPU in fact, was the major resource. All those tasks, active, waiting, ready, needed to be followed; they needed a **dispatcher**, yet another *scheduler*, scheduling the CPU to the list (request queue) of ready tasks. This task dispatcher was the heart of it all. It was a permanent, cyclic "program", either idling or scanning the task list for a ready task that it could turn active and thus allow to execute. In case of preemption, a signal would be given to the dispatcher so that it could interrupt the currently active task and give control to the preempting task. As a consequence, systems became synonymous with queuing systems. Other resources were going to be scheduled (i.e. waited for), either because they were slow (and exclusive) or because there was a limited amount of them... Early systems only scheduled I/O access and acted dramatically when other temporarily unavailable resources were applied for (by aborting the requestor); other systems were more lenient in that they caused the requestor to wait for resources when unavailable. In these systems, a new problem appeared: the **deadly embrace**.

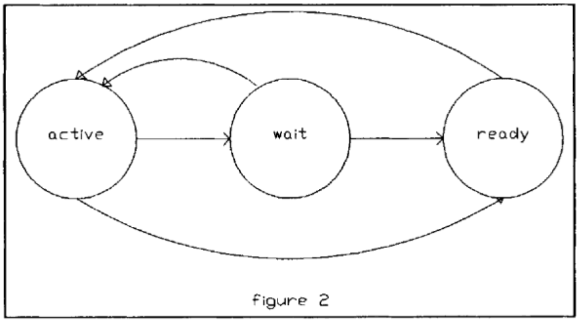


figure 2

The good old days were gone forever

By the mere fact that system had gone *multi*, programs could run simultaneously. They were called *concurrent*. All programs that executed concurrently in the system had to be physically present in core, on top of the system itself. *Core management* services were needed to place a program in core. As a result, a program could reside at different places in core for different executions, at different moments in time or even simultaneously. This forced programs to use either relative or base/displacement addressing instead of absolute addressing, since address 0 of the program no longer corresponded to address 0 of the machine. As a result, the assembly (or compilation) listing of a program would give addresses starting at 0, but a program *dump* would give real addresses, starting at the true load point. Since some fields of a program could not be anything but addresses of other fields, these had to be *relocated* by the loader, i.e. recomputed to their real value. All programs present in core could access all core, since the whole machine was one single range of accessible addresses: an *address space*. This was dangerous; security was enforced by means of so-called *storage keys* that prevented programs from accessing core not belonging to them. Somehow, each program (and also the system) had its own portion of core, constituting an individual *address space*. Routines *common* to programs could not exist therefore. Either they had to be copied into each program or they had to be made part of the system, so that they became indeed shareable (with some protected calling mechanism, obviously, otherwise anyone would have been able to go and do things in the system portion...). SYSTEM started spawning quite a number of resource *managers*. The primary resource had been I/O. Management of I/O led systems to the concept of concurrence, so that the immediately next resource to be managed was the CPU itself. **Dispatcher** handled that piece of fine craftsmanship. Core was another important resource that received management. It was needed for loading programs, but it was also needed for the allocation of the automatic storage (stacks) of Algol and PL1 programs. The core manager decreed that a program would be loaded into a *region* or *partition* big enough to contain it and also contain some free space for stack allocation. The system had also needs for allocating volatile data structures, if it were only because somehow all the *managers* needed to keep track of who used which resources. This was done in lists kept in system space core, and these lists grew and shrank as demands were satisfied. The *managers* needed their own space to handle their data structures in. Early system versions would do it in pre-allocated areas, but this proved rather inefficient as a lot of unused core might exist in some such areas and still would not be usable elsewhere. So later versions had a gigantic single area from which core blocks were allocated (on demand basis) to the managers. Core management, once more... Still: the whole thing could get full. What would happen in that case? Easy: SYSTEM would collapse. Users would first weep, next get angry, finally request action. A temporary solution was that only the unsatisfiable requesting program would be aborted. But this was not so acceptable either. So, another new solution: the demander would have to wait until the resource was free. As a consequence, the manager of the resource now had to maintain a list of waiting demands, and *ready* (or *post*) the topmost demander when resources became available. This in turn called for the task dispatcher to take such *wait-post* situations into account. Such situations sound easy. They are not. Let us take a look at things. A first program (called Liza) is running in the machine, while a second one (shall we say Robby?) also happily uses the CPU. Now, imagine that for some reason core management is "called" in, and runs *somewhat as a subroutine* for the calling task. The time-frame can very well be as in *figure 3.* Notice that *core management* when invoked (by MC or SVC for instance) runs under control of the demander. That is what I mean by drawing it in the left column or the right one.

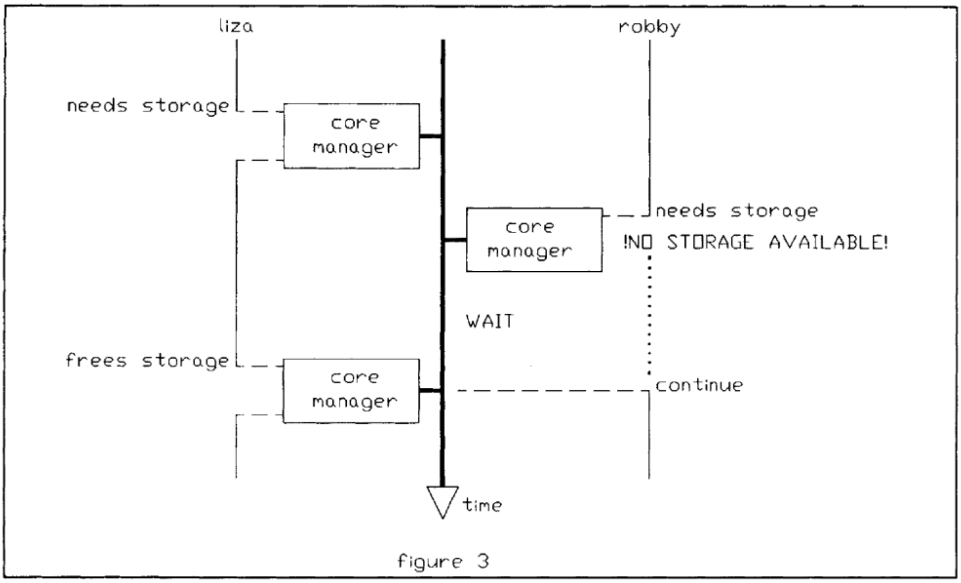


figure 3

It is obvious that a problem will occur in some cases, namely when the core management executions of one single module *overlap*. Is this allowed? Is it possible? The whole thing raises the suspicion that a program is also a *resource*, and might very well need to be scheduled. But couldn't such programs, that might be executed concurrently by many tasks, be *arranged*? A program, after all, is a set of instructions and some data. The set of instructions is absolutely stable, it is only read (at least if program execution does not modify the set of instructions, and unfortunately some programs do, but let us assume for the moment being that this is not the case). The data might also be classified into *read-only* and *not-read-only* fields. Clearly, concurrent execution of programs that only have read-only data is allowed without further ado, that is, no *manager* is needed. If programs have modifiable data as well, the question then is: can these data *values* be particularized per *user* (calling task) of the program? If yes, such data might be *allocated* in (an extension of) the task's address space, and the program might use an address scheme to access the various incarnations of the data. Again: core management is needed. Programs that do not modify their own instructions and have read-only data and use task-based incarnations of their modifiable data make up the very important class of *reentrant programs*. Reentrancy is a desirable attribute of programs in a multi environment, that much is clear. But not all programmers were able to see this: it is somewhat like the difference between plane-geometry and space-geometry. Allocation of data areas to programs so as to achieve reentrancy, added a perspective view to the program flow... Now, if programs could not be made reentrant, maybe there was another way out. Was the program *reu-*

sable? In other words, could it be used for one execution *after* it had completely ended the previous execution? For most programs this would certainly be the case. If this was guaranteed, one could imagine that a manager might come into the game, so as to *schedule* program executions to its demanders. As an example suppose that our core management program is not reentrant but is reusable. One way to use this program is to create a "core management" task with it, running on its own, and normally asleep. This task is woken up by a storage demand from any other task and proceeds to satisfy it. When in this interval, another storage demand occurs, the demander has to wait. At the end of execution of the first demand, the next demand (the waiting one) is serviced, and if there is none, the core management task goes to sleep again. So that problem is solved. But bear with me for a moment, as I'll show yet another problem. Suppose that two resources are to be managed, and that the situation of *figure 4* comes into existence. The situation is that no task gets out of this situation anymore! They wait forever. The situation is the now familiar *deadly embrace* or *fatal osculation*. The system world had discovered a severe disease, and was going to suffer from it. Lots of solutions were proposed and found unsatisfactory. But in due time the vaccine was found by Dijkstra: *Semaphores*. A very important notion, that will be widely dealt with in chapter 5.

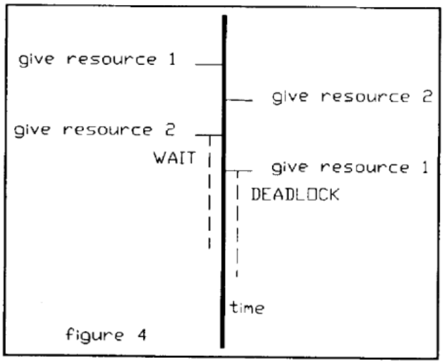


figure 4

Macroprocessor pops up again

New languages such as Algol and PL1 had come up, and although they did not really invade the everyday field, they did cause linguistic views to evolve. The Macroprocessor was not left untouched. One of my favourites is the MP1000 macroprocessor (a Philips product) which I'll use here as an example, with extensions of my own (the product doesn't exist anymore in any case...). No doubt, you remember what a macroprocessor does: it reads a text, containing *macro-calls* and replaces these by text, as generated by the corresponding *macro-definition* which is the procedure actually invoked by the macro-call. Let me formalize the definition just slightly. A *macro-call* is a being with the *type* **text** and stands for "inserted lines" (indeed a **string!**). A macro-call is clearly very similar to a function invocation. The macro-definition is the *function definition*. This brings us rather close to the Algol/PL1 functions. Are other properties of block-structured language also present? There are data types and scope rules. But there is no **if-then-else**, nor statement "grouping". But the macroprocessor has improved its syntax. There is a **MASSIGN** statement (assignment of macro-variables), a **MIF** statement, a **MGOTO** statement. There are declaratives : **MINT** (define an integer) and **MCHAR** (define a character string). There are built-in functions, most notably **MSUBSTR** (designate a portion of a string) and **MSUBLIST** (designate an element of a list). The pure macro statements are enclosed between *markers* (**$** and **;**). This makes the macro language nestable in any other language (especially if the **$** can be freely changed to some other special character). The true novelty however was the functional approach. Let me concentrate somewhat upon the macro-definition, in that light. Consider the left-hand Macro-definition in *figure 5*. Now this means exactly the

```
$MACRO  abc(parameters)   ;$MACRO abc (parameters);
text-line-1               $MRETURN ('text-line 1'/
text-line-2                       'text-line 2');
$MEND;                    $MEND;
```

figure 5

same as the definition on the right, provided we define the $MRETURN statement as the statement that assigns the return value of the function name and also introduce the *slash* operator for concatenation of variables (with an inserted line feed). We already know that macroprocessors do not want quotes around lines to be returned, as we can observe on the non-RETURN version. But we also need a way to set up conditional concatenation. The Philips macroprocessor does it by using the MIF statement in-line with the concatenation. The macro definition on the left corresponds to the MRETURN version on the right in *figure 6*. A rather important point appears: the *operand* of a macro-statement (here MRETURN) *could* be a text that looks like a macro-definition body or portion thereof. This is called a macro-expression. No macroprocessor actually has an MRETURN. But a feature like it is needed. Suppose we want a macro that creates a line made up of the concatenation of two parameters (say ABC and XYZ), with one *blank* separating them. It will not be sufficient to write something like **ABC XYZ** as this would be taken at *face value* to mean the characters A and B and C and the blank and X and Y and Z. We need to indicate that ABC and XYZ are macro-variables. So let us stipulate that such a name must be prefixed by a $; our *statement* becomes **$ABC $XYZ**. This is fine. But what if the generated text must be followed (without intervening blank) by the characters TUV? Can we write **$ABC $XYZTUV**? Certainly we can't. In fact we need some kind of separator. For reasons that will become clear somewhat later, the dot was used here. Thus we write: **$ABC.$XYZ.TUV**. The **$** and **.** brackets were said to delimit a replacement statement. Let us investigate it somewhat further. Whatever is between **$** and **.** must be any valid macro-expression. Thus if we introduce the **comma** as a concatenation operator (without generation of an intervening blank), we might also have written: **$ABC,' ',XYZ,'TUV'.** (and notice that we need quotes in the expression to delimit constants). We have already seen that a macro-instruction can also act as a macro-expression. We are allowed to write **$MASSIGN A=B+C.** (instead of **MASSIGN A=B+C;**). The effect is peculiar though. In fact, the **$.** brackets cause whatever is contained between them to be *evaluated*. The result should be a string, and this string *replaces* the whole statement. So, MASSIGN A=B+C will be evaluated. The evaluation of a macro-statement results in its effectuation (causing the effect of A acquiring a new value) but also results in the actual *value* **empty** (this is the case for any macro-statement evaluated as an expression). So the replacement statement has no value by itself, but provokes a side-effect. Replacement statements can also be nested. The statement **$$ABC..** means that the *inner* **$ABC.** is replaced by its value, say XYZ (a string). This means that we now have **$XYZ.** to be evaluated, and this will work, if XYZ is itself a macro-

```
$MACRO abc (...);              $MACRO abc (...);
text-line-1                    $MRETURN('text-line-1'/
$MIF condition GOTO lab1;      $MIF condition GOTO lab1;
text-line-2                           'text-line-2' $lab1:;);
lab1:MEND;                     $MEND;
```

figure 6

expression (textually) or, at the least, the name of a macro-variable. Another example is

$MASSIGN X='$MIF condition GOTO LAB1;GLITCH $LAB1:;FLUKE'; $X.

The replacement statement for X will actually execute the text contained in X and leave as a result the string FLUKE or GLITCHFLUKE. Replacement statements can also be used within macro-statements:

$MASSIGN ABC='LAB1';$MIF condition GOTO $ABC.;

will result in conditional branching to LAB1. Finally, also notice the very important fact that string variables (defined by **MCHAR**) have no limited length: the macroprocessor considers them to have *varying* length; in other words, the length is adapted to what is needed at any given moment. This allows us to perform any kind of concatenation such as **$MASSIGN X=Y,Z;** or even **MASSIGN X=X,X;** which merely replicates X behind itself.

Let us recapitulate: the operand of a replacement statement can be either a macro-variable or a constant string or a macro-expression. If it is a variable, the value of the variable is fetched. If it is an expression, this is evaluated first. Variable, constant and expression thus yield a string. This string is scanned to see if it doesn't contain any macro-statement. If so, this is considered a nested replacement and evaluated, and so on. A macro-call is a function invocation, as we have seen. Thus, a macro-call yields a string, and therefore we could just as well write **$macro-call.** instead of **$macro-call;**. The latter notation considers the macro-call as a statement. A striking example is given by the three texts following. We will admit that the *value* of a MACRO/MEND block is **empty**. We will also require that quotes contained in a quoted string should be doubled. The texts, with *body* standing for one valid macro-definition body are given in *figure 7.* I leave it to you to find out how these three texts achieve exactly the same net end result: that of setting up the macro-definition for abc.

```
$MACRO abc;  $                 $$
$            '$MACRO abc;      '$MACRO abc;
'body'       body              ''body'''
•            $MEND;'           •
$MEND;       •                 $MEND;'
                               ••
```

figure 7

The importance of macroprocessors cannot be underestimated: they were (and still are) the tool which allowed programmers to *extend* any given language. Languages could be given the essential property of *open-endedness*. Alas, not many bothered.

Macroprocessors were the first string processors ever. They did it in a *functional* way: macros are *functions* that are used for their replacement value. They opened the world of *string processing*, a very important domain which would be further explored by SNOBOL. One of the interesting applications of macro-processors was in creating two-level compilers for high-level languages: the compiler itself translated the language lines into macro-calls. A macro-library was then used to further translate the macro-calls into assembler code (which was processed by an assembler). The advantage to this approach was in the fact that for a given language only one compiler had to be developed and a specific macro-library was delivered for each target machine. long the same lines, macro-processing was sometimes used to help translate one language into another or to create full jargons of a language.

ened out. Pascal brings the usual Algol types such as integer and real. It also allows the programmer to regroup variables in agglomerate structures (like PL1) and calls these **records.** It has (multi-dimensioned) arrays. But Pascal does more. First of all, it makes a sharp distinction between **constants** and **variables**, and introduces the named constant (protected against a change of value). **Label** constants (for use as a target in a goto statement) must also be pre-declared at the top of the program, otherwise they can not be used. This is a timid attempt at imposing discipline upon the usage of the goto. Pascal introduces also many new data types. The most important one is the **scalar** type. To allow declaration of such beings, Pascal offers a new declarative: the **type** declarative. This permits the programmer to define new data types of his own. For instance, suppose a program needs to work with colours. Then one can define the **scalar** notion of colour: **type colour = (white,yellow,red,blue,violet,black)**. The type declaration establishes the name **colour** and also establishes the possible **constant** values this name stands for. Mind you, a **type** declarative does **not** create a variable. Instead, it now allows one to declare variables that have the type colour. The variable "paint" can be declared as follows: **var paint:colour.** From now on, the variable *paint* can be manipulated in the program. It can receive values, but only those allowed in the type colour. And of course, it is not possible to perform arithmetic upon such variables. They can however be tested in comparison operations. The definition set of colour is also considered **ordered**. Thus yellow is **higher** than white. An important aspect is that the Pascal **for** statement allows only the use of such scalar types. **for paint:=yellow to blue do action** will execute the action repeatedly, while the variable paint takes the values yellow, red, blue in succession. Many of the standard types are also scalar: integers have a definition set that is the range of integers; booleans range over **false** and **true** (in that order); the new type **char** ranges over a standardized representation of one-character codes (the so-called ASCII set).

Another important data type is the **subrange** type. This is in fact a subset of a scalar type. If we keep our definition of colour, we can define a new type as a subrange: **type dark=blue..black** and declare a variable to have the type dark: **var morkpaint:dark.** Morkpaint can only take the values between blue and black. An interesting feature. One can also define a subrange over the standard types: **type letter='a'..'z'** and **type onedigit=0..9**. Scalar types and Subranges are also used as bounds for arrays:

var x:array[1..25]of integer and **var y:array[colour]of integer**

Pascal also allows one to define a type as a structure (i.e. a record):

type mystr=record name:array[1..30]of char;id:integer;status:(married,single) end

A variable can now be declared to have that type: **var employee: mystr**. This means that employee is a record and contains the sub-variables employee.name (which is an array of 30 characters), employee.id (an integer) and employee.status (a scalar with possible values *married* and *single*). PL1 and Cobol allow the siting of one variable over another one (DEFINED and REDEFINES). This is a dangerous feature. Pascal decided to clean it up. First of all such identical sitings are allowed only within records. Next, they can only be done over the tail of the record (only the last portion -possibly the whole- of a record can be redefined as something else). The notion is that of **variant**. In fact, says Pascal, a record (or portion thereof) can have more than

proc([]int,real)struct(int,int). A very important feature of Algol68 was that procedures could have procedures as parameters. Suppose an integer array-returning procedure that takes another procedure with an integer parameter and an integer result as a parameter, it would have the type **proc(proc(int)int)[]int**. Features like these would certainly have been welcome in Pascal.

Pascal's misfortune

Notwithstanding its shortcomings, Pascal is a truly excellent language. It is clean and simple. It is much more powerful than Algol (especially in its objects). But it was not a (commercial) success. Not at all. Everyone spoke about Pascal, but no one, apart from the academic circles, used it. Cobol continued its unchallenged reign in the applications world. Assembler held the throne in the systems world. Of course, Pascal had no file management (or very little). But had file management been available, I do not think it would have made any difference: it was too late. Thus Pascal went the same way as Algol: it became a *publication* language for formalists. And it also became a model for later languages such as MODULA, C (to an extent), ADA. Pascal had its revenge, though: it is possibly the most widespread micro-computer language of today (after BASIC, though). The 60's were roaring indeed. But most people were deaf to the noise.

Is there still something in a name?

Pascal is a fantastic language, full of marvellous beauties. Especially the new object: the *type*. Pascal allows one to define and name a type, which becomes usable in *variable* declarations and in other type declarations (so types are *nested*). But, one may rightly ask, what kind of object is a type? Can it be assigned? The answer is no. Thus a type is a (named) *constant*. So far so good. Next question: does it have a value? It does: the value is structure, since there are recognizable tokens in the string (type within type). It sounds somewhat like a procedure definition, where the body is a piece of text. A type is a *declaration subroutine*... But a type of object cannot be manipulated at all. There is no single *executable* statement that uses a type name as operand... The only usage is within declaratives. And at that level there are *type expressions*, limited to *self defining* type constants or type names. One can indeed write : **type xyz = 1..25; var abc : xyz** where we use a type name or **var abc : 1..25** where we use a type "*constant*". The idea that type names (and type self-definers) are the beings with which you *program* declaratives, is somewhat new. And intriguing enough. The *case* variant within a record type looks a lot like programming within the declaratives... Is a type-name sited? An awkward question. Since a type-name cannot be used within an executable statement, there seems to be no need for siting. And this is true, for execution time. But the fact is that type names are most definitely sited during *compiler time*. The location is in the symbol dictionary that the compiler maintains while compiling (some compilers append the dictionary to the object code, which allows symbolic tracing, but that is another, rather marginal, matter). The type definition is the first real emergence of compile-time objects... (in a way, macro features are also compile-time objects) programs had to be complicated, otherwise they were wrong. Programs were not explainable. Programs could only be made by the upper class, the intelligentsia. Thus anyone making programs was part of it. A specialist. And very soon, each programmer started considering his fellow programmer as particularly inept. Then, each programmer felt entitled to dictate the word to his colleagues and called himself an Analyst. The Analyst was the knower of the Truth. His own truth, obviously. The only possible truth. Negating all other truths claimed by... well, by whom? ... just look at the other paupers. So they all were analysts. They spoke the truth(s). They were however not applying it anymore and needed workers to do just that. Data Processing had found its first elite. Now, the elite wanted slaves. So they started hiring "coders": people, generally untrained, able to rewrite the analysts dogma's in true programming languages. It mattered not how they did it. And very soon the coders wanted to be gods too. So they became programmers (the job had been vacated, remember?) and analyst-programmers or programmer-analysts or senior-programmers or expert programmers or whatever they pleased to call themselves. They (the coders) started speaking truths about the truths. The successful ones indicated what *not* to do if you had to implement a truth. So they said *don't use anything but assembler for systems programming; don't use anything but Cobol for applications; do not nest IF statements; do not make programs of more than so many lines; do not use macroprocessors; do not use this fine construct; do not use that other fine technique; do not be smart about your job!* The 60's gave all the tools needed to really make a great job of it. No one used them.

Decision tables: A matter of style?

With such a psychology oozing into the job, what could programming *style* amount to? Right: chaos! Everyone decided to garble programs as much as possible, so as to create the impression of utmost difficulty and thus of specialized work. Program authors made themselves indispensable. Analysts confused agitation and action and therefore couldn't care less. Programs, as a result, became huge, illegible, incomprehensible. Moreover, they were untestable, unamendable. Maintenance became an overwhelming and self-justified business. Managers started really worrying: programming costs rocketed sky-high. Performance (in terms of program quality) was poor. It was in those days that the myth was born: *programming was unavoidably complicated and expensive*. Darkness triumphed over clarity. But at the same time, the search for light was begun. When things are complicated, so managers said, *methods* are needed to keep things under control. It was in the realm of application programming that methods started appearing. In those days, an interesting one was *decision tables*. A decision table is a graphical way to map conditions against actions: it expresses the fact that certain actions are to be performed only if certain conditions (or combination thereof) are satisfied.

minterm	x	y	z	Fa
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	0

figure 11

Consider an action **A**. Let us also define a boolean function $\mathbf{F_A(x,y,z)}$ which if true indicates that **A** must be executed, and if false prohibits such execution. x,y,z are three conditions which

We assume that the above registers take information when they receive a *take* signal.

There are also some hard-coded constants:

- a hard-coded constant of value 1 (C1); in fact a read-only register containing the value 1; it is used to add or subtract *one* to or from other registers when needed;

- a hard coded constant of value 4 (C4).

All these hardware elements are connected to a 32-bit large *bus-bar.* This is a *cable* of 32 wires, which allows the machine to transport 32 (or less) signals from one hardware element to another. The connection of an element to the bus-bar is *disabled* (i.e. the signals present in an element do not flow into the bus-bar), but can be *enabled* electronically by means of a *signal* combined with a *clock pulse.* It remains enabled for only a very short while. Thus the connections are like sophisticated *points* in a railway network. There also exists a secondary 4-bit large bus-bar. We can now draw our complete simplified machine, see *figure 19.* Each dot on the bus-bar is a connection (which can be enabled). The instruction register (IR) is given a 32-bit instruction. However, it is seen as a set of four 4-byte registers as well, of which IR1 is separately wired for read-out; IR2 is decomposed in two separately readable 4-bit sub-registers: L and H; IR3 and IR4 together form a 16-bit register (IR34), also wired for read-out. We now assume that the assembler language for which the μ-program is intended is composed of only 4-byte instructions, having a one-byte opcode, two 4-bit index registers and a two-byte displacement. Typically, we could have the instruction **ADD R1,R2(D).** This instruction means: add to R1 the contents of the cell whose address is obtained by adding the contents of R2 to D. I will note such an operation as **[R1]←[[R2] + D] + [R1]**, where the square brackets mean *contents of.* The instruction, when loaded in the IR register, fills it as follows: IR1 holds the op-code, H holds R1, L holds R2, IR34 holds the displacement (D). How would our ADD object instruction actually be executed by the Simple Machine? It would take place according to the following steps:

(1) get the instruction into IR:

- place [PC] into AR (by enabling the two connections and delivering a take signal to AR, along with a clock pulse)

- get memory contents into MR (by delivering a read signal to the store, along with a clock pulse)

- place [MR] into IR (by enabling the two connections and delivering a take signal to IR, along with a clock pulse)

(2) *analyze* [IR1], to see what the operation is; in our case it is an ADD (say code 78)

(3) *select* in the μ-program the appropriate sequence of actions for code 78, and execute them (a μ-program subroutine), which implies the following actions:

- place [L] (i.e. R2) in the addressing selector of the index store, via bus-bar 2 (by enabling the L connection of bus 2; L acts as address register for the index store)

- get index contents into the A-register of the ALU (by delivering a read signal to the index store and enabling the connection to A: the contents are put immediately into A, which acts as memory register for the index store)

- add [IR34] to [R2] into the C-register (by enabling the connection from IR34 to the left-side (LS) of the ALU, and giving an *add* signal to the ALU); this gives in C the memory location of the first operand to be processed by the ADD instruction.

- the real μ-program starts at the 257th word
- there is a μ-program compiler which allows us to write the μ-instructions in a *friendly* way; in particular, it allows us to write addresses of μ-words as their sequence number, and will translate these to the true addresses; it also allows symbolic addressing.

The syntax of a μ-instruction is:

[symbolic address] command [B address] [:comments]

where the *B address* part indicates a branch to a constant address. Otherwise, the address of the next sequential μ-instruction is taken. The μ-program portion that interprets our ADD instruction is given in *figure 21*. The formats of the various commands are explained in flight.

As can be seen, μ-programming is programming. However, the objects one manipulates are not the usual ones. They are the raw hardware components. The language of μ-programming is a command language, as could have been expected. This command-language has also evolved: Algol-like connectives are used (if then else) and macroscopic functions also became available (in fact, they refer either to subroutines which are standard or to more evolved built-in hardware functions).

Lots of things can be achieved with micro-programming. Powerful and concise assembler languages can be created. Operating system functions could even be implemented as micro-programs. A fascinating domain. But one we will now abandon and will not take up again.

INDEX OF CONCEPTS

the numbers indicated refer to the page having the section where the concept is discussed